Шнейвайс А.Б.

ОСНОВЫ ПРОГРАММИРОВАНИЯ

(ФОРТРАН, СИ)
Первый семестр

для студентов астрономического отделения
математико-механического факультета СПбГУ

2016

Шнейвайс А.Б.

Учебное пособие «Азы программирования на ФОРТРАНе и СИ» (первый семестр) содержит информацию по дисциплине «Программирование», излагаемую студентам первого курса астрономического отделения математико-механического факультета СПбГУ. Основное внимание уделяется не только правильности использования синтаксических языковых конструкций, структурности записи исходных текстов и модульному подходу при разработке проектов программ, но и выработке навыков простейшей оценки программируемых формул (с точки зрения их пригодности для ведения расчётов на ЭВМ), а также умения приводить их к виду удобному для расчёта посредством простейших аналитических приёмов. Пособие содержит множество примеров исходных текстов программ (с подробными пояснениями), что (при добросовестном отношении к делу) позволит вчерашнему школьнику за относительно короткий срок приобрести навыки поиска синтаксических ошибок, использования операторов форматного ввода/вывода, процедурно-ориентрованного и модульного программирования. Почти каждая тема, излагаемая в пособии, завершается кратким перечнем её основных моментов и списком задач соответствующего домашнего задания, которые необходимо выполнить в среде операционной системы LINUX. Пособие завершается приложением, содержащим набор зачётных задач. Результаты программ, приведённых в пособии, получены в основном на компиляторах gfortran и gcc

Рекомендовано учебно-методической комиссией
математико-механического факультета СПбГУ
для студентов, обучающихся по специальности
«АСТРОНОМИЯ».

Содержание

1 Вводная.

На сайте **http://www.astro.spbu.ru/astro/win/index.html** (Астрономичский институт им. В.В.Соболева), в частности, имеются

1. Лекционный курс **ВТОРОГО** семестра *"Программирование и математическое обеспечение ЭВМ для астрономов"* (В.Б.Титов, А.Д. Положенцев, В.Б. Ильин).

2. Практические пособия *"Азы программирования на ФОРТРАНе и СИ"*, *"Азы GNUPLOTa"* (А.Б. Шнейвайс), которые можно использовать в **ПЕРВОМ** семестре для начального освоения программирования на упомянутых языках.

Классическое изложение международного стандарта языка ФОРТРАН-95 дается в книге *"Программирование на современном ФОРТРАНе"* (А.М. Горелик,2006) и языка **C++** в книге *Самоучитель C++* (Шилдт Г. 2002).

Для серьёзного знакомства с основами программирования в среде **Linux** полезны руководства Даниела Роббинса (перевод В.Б.Титова)

Обширная инфомация по темам **LINUX**, **ФОРТРАН**, **C**, **C++** имеется по адресу **http://gamma.math.spbu.ru/user/rus/cluster/progr_1.shtml**

1.1 Элементарное знакомство с UNIX.

1. Основные понятия.
2. Вход в систему.
3. Выход из системы.
4. Системная и виртуальные консоли.
5. Команда – первое слово после промпта.
6. Аргументы и опции команды.
7. Примеры некоторых простых команд.

1.1.1 Основные понятия.

UNIX – многозадачная, многопользовательская операционная система.

Операционная система (ОС) – набор команд и программ, обеспечивающих:

• наиболее оптимальное использование всех устройств ЭВМ;

• наибольшее удобство работы с ЭВМ человеку (**пользователю**).

Главные части UNIX:

Ядро (kernel) – основная часть ОС, выполняющая большую часть работы: управляет распределением памяти, разделением времени, центральным процессором, собственно выполняет вводимые команды. При этом **ядро**, как управляющая программа, изолирует нас (то есть пользователя) от сложности аппаратуры.

Файловая система (file system) – отвечает за организацию, хранение, использование и защиту **файлов**.

Файл (операционной системы UNIX) поименованная область памяти, предназначенная для хранения данных. **Файл** состоит просто из последовательности литер, которая может быть и пустой. Совокупность литер, завершающаяся литерой „новая строка“, образует одну **текстовую запись**.

Оболочка (shell) – программа общения операционной системы с пользователем; один из интерфейсов UNIX. Оболочка изолирует нас от сложностей ядра. Средство общения, предоставляемое оболочкой на экране,– **командная строка**, в которой набираются нужные инструкции (**команды**). Есть разные варианты оболочек: командные процессоры языка СИ, Борна (Baurne) и др. Оболочка – аналог интерпретатора командной строки **command.com** в **MS DOS**. Точнее, последний сильно обедненный вариант командного процессора **UNIX**.

1.1.2 Вход в систему.

После загрузки монитор обычно находиться в **графическом** режиме работы (возможен и **текстовый** режим; о переключении режимов см. пункт **1.1.4**). В графическом режиме, видна **панель приглашения**, в которой есть поле команды **login** входа в систему (в случае текстового видна командная строка с **login**-приглашением). В качестве аргумента команды следует набрать свое **регистрационное имя** (то есть информировать ОС систему о своем желании пообщаться с ней).

Регистрационное имя уникально для каждого пользователя. Оно узнается у администратора системы. После набора регистрационного имени нажимаем клавишу ввода **Enter**. Тогда команда **login** запросит **пароль**. В графическом режиме можно и, не нажимая клавишу ввода, перевести **курсор** в поле набора пароля клавишей **Tab** (или посредством указателя мыши). Пароль, как и **login**-имя, узнается у администратора или оператора дисплейного класса и при наборе либо высвечивается символами “*”, либо не высвечивается вообще.

При верном наборе **login**-имени и пароля на экране появляется **приглашение командной строки** (несколько символов), которое иногда называют **промптом** (prompt – подсказка), поскольку оно указывает строку экрана для набора команд (**командную строку**). В случае неверного набора пароля UNIX сообщает об этом и предлагает повторить вход в систему. Существует команда **passwd** смены пароля (о ней позже). Пароль должен состоять не менее чем из 7 символов, среди которых должны встречаться не только буквы, но и другие литеры.

Ни в коем случае никому не сообщаем свой пароль!!!

1.1.3 Выход из системы.

При работе в **текстовом режиме** набираем в командной строке команду **exit** и нажимаем на клавишу **Enter**. Выход достигается и совместным нажатием клавиш **Ctrl** и **d**. В некоторых модификациях системы UNIX используется и команда **logout**.
При работе в **графической оболочке** выход необходимо осуществлять через виртуальную кнопку **завершить сеанс**, которая помещается в нижней строке меню, вызываемого на экран посредством соответствующей иконки.

Выход из системы всегда доводим до login–приглашения!!!

В противном случае человек, подошедший к терминалу, получает возможность не только за ваш счет *"пошастать" по Интернету*, но и стереть все ваши наработки, а при знании вашего пароля даже изменить его.

1.1.4 Системная и виртуальные консоли.

Системная консоль – монитор и клавиатура, которые непосредственно связаны с операционной системой.

Виртуальная консоль – действующая программная модель системной консоли. Позволяет нескольким пользователям (с разными **login**-именами) работать независимо через одну системную консоль. Один **login**-пользователь, тоже может под своим единственным **login**-именем общаться с UNIX через разные виртуальные консоли, правда, проинформировав UNIX посредством ввода **login**-имени столько раз, сколько виртуальных консолей он предполагает использовать. При работе монитора под управлением графической оболочки, войдя в систему под своим **login**-именем, можно вызывать сколько-угодно **терминалов** – еще один способ общения с оболочкой – без необходимости ввода соответствующего количества **login**-имен.

Режим монитора	Клавиши переключения консолей
Текстовый	**Alt** + $\mathbf{F_k}$
Графический	**Ctrl** + **Alt** + $\mathbf{F_k}$

Здесь $\mathbf{F_k}$ – функциональная клавиша клавиатуры под номером **k**.

Виртуальная консоль с **k=7** по **k=12** отведена под работу в графическом режиме.

1.1.5 Команда – первое слово после промпта.

Например, **$ fox kolobok**. Здесь **$** – промпт, **fox** – имя команды, **kolobok** – аргумент команды. Аргумент отделяется от команды не менее чем одним пробелом.

При верном наборе команды и нажатии на клавишу **enter** команда выполняется и работает со своим аргументом (если это возможно). Если же набор ошибочен (допущена опечатка или не указан путь к месту расположения команды), то на экран, после строки с введенной командой выведется текст **command not found**. Все набираемые команды запоминаются в некотором **файле** так, что любую из них можно вызвать на экран клавишами управления курсором ↑, ↓.

1.1.6 Аргументы и опции команды.

После команды могут следовать **аргументы** и **опции** (вместе, порознь или вообще не следовать; все зависит от команды и/или от нашего желания).

Аргумент – это то с чем команда работает.

Опции – это переключатели, настраивающие команду на тот или иной режимы работы. Каждая команда имеет свои **синтаксис** (правила записи) и **семантику** (смысловую нагрузку), которая в зависимости от аргументов и опций может видоизменяться). Помнить все опции каждой команды человеку не надо – по командам UNIX существуют справочники (например, см. [5]). Кроме того, всегда можно воспользоваться командой **man** (от **manual** – руководство) или командой **info**.

1.1.7 Примеры некоторых простых команд.

date - вывод текущей даты и времени.

```
bash-2.05b$ date
Птн Сен 24 13:03:12 MSD 2004
bash-2.05b$
```

who – кто на момент запроса работает в системе.

```
bash-2.05b$ who
aw       :0           Sep 24 12:53
aw       pts/1        Sep 24 12:54 (:0.0)
aw       pts/2        Sep 24 13:02 (:0.0)
nvv      pts/2        Sep 23 13:09 (dust.astro.spbu.ru)
seger    pts/1        Sep 23 21:11 (sis.dorms.spbu.ru)
bash-2.05b$
```

Первая колонка – **login**-имя пользователя; вторая – имя используемого терминала; третья – дата и время входа в систему; в скобках имя удаленного узла, с которого произведена регистрация X-консоли.

whoami – регистрационное имя пользователя.

```
bash-2.05b$ whoami
aw
```

cal - календарь на текущий месяц.

```
bash-2.05b$ cal
    Сентября 2004
Вс Пн Вт Ср Чт Пт Сб
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

man cal – справка по команде календарь.

Возврат из обзора справки в режим работы командной строки – нажатие клавиши **q**.

1.2 Немного о файлах и каталогах UNIX.

1. Понятия файла и каталога.
2. Пример дерева каталогов.
3. Домашний, текущий и корневой каталоги и их псевдонимы.
4. О чем узнали из первых двух параграфов (конспективно)?

1.2.1 Понятия файла и каталога

Файл – именованное место для хранения данных (набор данных, которому дано имя; **file** – папка, скоросшиватель). Выгодно, чтобы имя файла напоминало то, ради чего файл создается. Например,

Имя файла	напоминает, что в файле находится текст
myfirst.for	моей первой программы на ФОРТРАНе в фиксированном формате;
myfirst.f95	в свободном формате ФОРТРАНа-95;
myfirst.c	на языке C;
myfirst.cpp	на языке C++.

Условимся, пока, не включать в имена файлов символы *** , / , ? , ‘ , ’** .

Каталог (директория) – именованная совокупность файлов.
Файлы выгодно группировать в каталоги по какому-то (удобному нам) признаку – это ускоряет и поиск, и работу с ними. Например, лекции по матанализу, астрономии и физике пишутся все-таки в разных тетрадях. И в программировании удобно, чтобы среди файлов, касающихся одной конкретной задачи, не было ненужных.

Каталог, если надо, может содержать внутри себя не только файлы, но и другие каталоги (подкаталоги, поддиректории): файловая система UNIX – **иерархична**. Поэтому полное имя файла должно включать в себя и имена всех содержащих его каталогов, например, **/home/m06avs/task1/for77/myfirst.for**.
Первый символ “/” обозначает **корневой каталог** – каталог самого верхнего уровня, т.е. тот, который содержит все каталоги. Остальные символы “/” служат просто для отделения имен каталогов друг от друга и от имени файла.

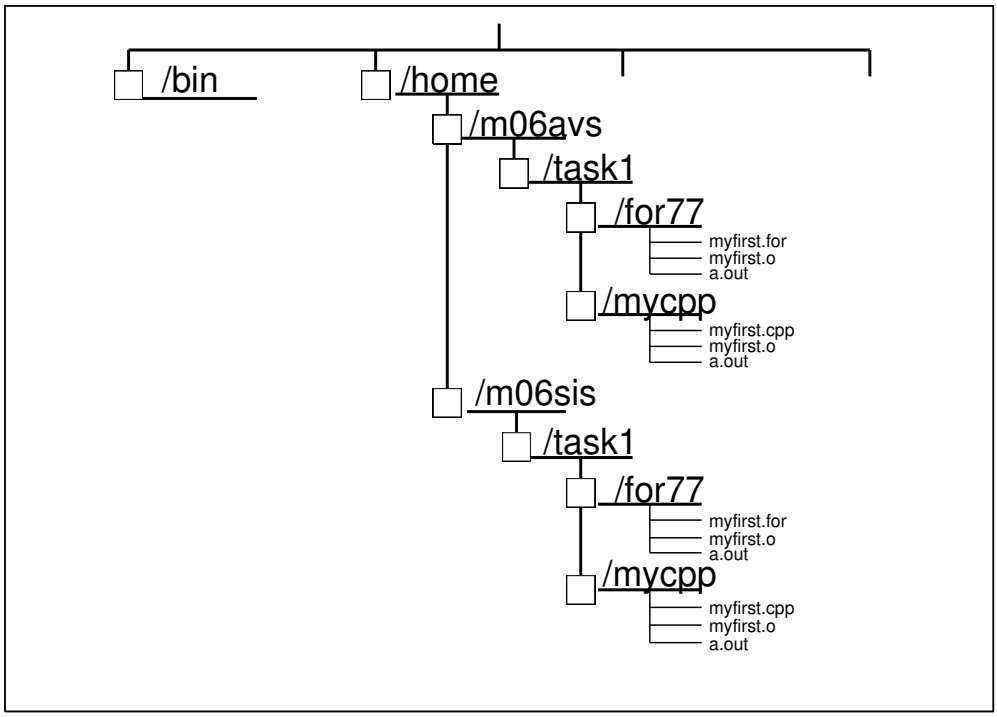
Путь – это маршрут, который надо проделать по иерархии подкаталогов, чтобы получить доступ к конкретному файлу. Каталог, ближайший по иерархии, и содержащий данный называют **родительским**. Так каталог **m06avs** – родительский по отношению к каталогу **task1**, а каталог **home** – родительский для каталога **m06avs**.

Имена разных файлов в одном каталоге не могут быть одинаковыми, но разные имена в одном каталоге могут указывать на один и тот же файл (см.,например, команду **ln**).

1.2.2 Пример дерева каталогов.

Подкаталог **home** содержит имена подкаталогов, каждый из которых закреплен за конкретным пользователем. В дисплейных классах математико-механического факультета имя такого отдельного подкаталога формируется из литеры **m**, двух литер, обозначающих последние две цифры года поступления в университет, и трех букв, с каждой из которых начинаются соответственно фамилия, имя и отчество пользователя.

Так, если **Анна Валерьевна Синицына** и **Сергей Иванович Соколов** стали студентами в 2006 году, то первой будет предоставлен подкаталог **m06sav**, а второму – **m06sis**. Такими же окажутся и соответствующие **login**-имена. Ниже дан фрагмент дерева каталогов упомянутых пользователей.



Решение о структуре дерева подкаталогов принимает лично пользователь. Например, если знания по языкам **ФОРТРАН** и **СИ++** до поступления на факультет у пользователя отсутствовали, то вполне обосновано для него создание подкаталогов **for95** и **mycpp**, в каждом из которых разместить подкаталоги зачетных задач **task1**, **task2** и т.д. Если же налицо свободное владение обоими языками, то, вероятно, не менее объективно подкаталоги задач **task1** и **task2** оформить родительскими по отношению к подкаталогам **for95** и **mycpp**, хранящих исходный, объектный и исполнимый коды конкретного языка программирования.

1.2.3 Домашний, текущий и корневой каталоги и их псевдонимы.

После входа в систему попадаем в свой **домашний** (базовый) каталог, который выделен системным администратором. Имя домашнего каталога совпадает с вашим **login**-именем. Так что Анна Валерьевна при удачном входе в систему сразу попадает в свой **домашний** каталог **m06avs**, а Сергей Иванович в **m06sis**.

Домашний каталог имеет псевдоним – значок "$\sim$" (тильда) или же "$\sim/$", что удобно при указании пути без явного перечисления имён всех каталогов от корневого.

Команда командной строки всегда относится к тому каталогу, в котором находимся в данный момент. Этот каталог обычно называют **текущим** или **рабочим**. Так после входа в систему **текущим** становится **домашний** каталог.

Текущий каталог имеет псевдонимы – значок "**.**" (точка) или же "**./**"
Когда уровень погружения подкаталогов велик, то полный маршрут к текущему каталогу человек часто забывает. Напомнить себе абсолютное имя маршрута можно командой **pwd** (print working directory):

```
$ pwd
/home/m06avs
```

Новый подкаталог в текущем создаётся командой **mkdir** (make directory):

```
$ mkdir task1
```

По дереву каталогов перемещаемся посредством команды **cd** (change directory):

```
$ cd task1
```

Убедимся, что погружение в каталог **task1** произошло:

```
$ pwd
/home/m06avs/task1
```

Высвечивание на экран имен файлов и поддиректорий текущего каталога реализуется командой **ls** (сокращение от list – список):

```
$ ls
```

Сейчас в этом списке ничего нет (каталог **task1** только что создан и он пуст).
Создадим в нем два каталога **for95** и **mytemp**. В первом разместим (не сейчас, а в дальнейшем, см. подраздел **1.4**) первую программу на ФОРТРАНе. Второй используем чуть ниже для знакомства с командой уничтожения пустого каталога.

```
$ mkdir for95 mytemp
```

Теперь дадим команду **ls**:

```
$ ls
for95 mytemp
```

Видим, что в каталоге **task1** появились имена подкаталогов **for95** и **mytemp**. Уничтожим пустой каталог **mytemp** командой **rmdir**:

```
$ rmdir mytemp
$ ls
for95
```

Возврат в **родительский** для каталога **task1** каталог **m06avs** путем

```
$ cd m06avs
bash: cd: m06avs: No such file or directory
```

невозможен. Дело в том, что команда **cd m06avs**, выданная из текущего каталога **task1** "пытается войти" в каталог **m06avs**, который *"по ее мнению"* должен находиться внутри **task1**. Отсутствие такового и привело к высветке наблюдаемого сообщения. Вернуться в **родительский** каталог из **текущего** можно, указывая весь маршрут к нему от **корневого**.

```
$ cd /home/m06avs
```

Проще, однако, использовать неизменный псевдоним родительского каталога с любым именем – "`..`" (две следующие друг за другом без пробела точки):

```
$ cd task1      вошли еще раз в task1
$
$ cd ..         используем псевдоним родительского
$
$ pwd           убеждаемся в правильности выхода в него
/home/m06avs
$
```

В данном случае каталог **m06avs** – домашний (у него есть особый псевдоним – значок "тильда" (~). Поэтому, если надо вернуться в **m06avs** из подкаталога **task1** (или вообще из любого подкаталога, находяшегося на глубоком уровне иерархии), то достаточно из последнего дать команду **cd ~** или даже просто **cd** без парметров. Проверим это

```
$ cd ~/task1/for95      проникли в for95
$ cd ~                  вернулись в домашний.
$ pwd                   убедились, что все в порядке.
/home/m06avs
$ cd ~/m06avs/task1     вернулись опять в task1
$ cd
$ pwd
/home/m06avs
```

Ценность значка ~ не в том, что с ним упрощается выход в домашний каталог (как сейчас видели, есть более простой способ), а в том что тильду можно использовать в программах, написанных на языке оболочки (так называемых **скриптах**), в качестве отправной точки при составлении маршрутов к любому подкаталогу внутри домашнего.

1.2.4 О чем узнали из первых двух параграфов (конспективно).

1. UNIX – операционная система.
2. Назначение операционных систем.
3. Три основные части UNIX: ядро, файловая система, оболочка. и их назначение.
4. Вход в систему: (**login**-имя и пароль).
5. Выход из системы (**exit** и кнопка “Завершить сеанс”).
6. Системная консоль. Виртуальная консоль. Переключение консолей.
7. Командная строка.
8. Команда – инструкция для UNIX. Аргументы и опции.
9. Синтаксис и семантика команды.
10. Команды **who**, **whoami**, **cal**,
11. Команда **man** (выход из нее – литера **q**).
12. Файл UNIX– неструктурированная последовательность байт.
13. Файлы удобно объединять в каталоги.
14. Пока не использовать в именах файлов ***** , / , **?** , ‘ , ’ .
15. Символ / - псевдоним корневого каталога
16. Символ / - разделитель имен каталогов и файла.
17. Символ $\sim$ - псевдоним домашнего каталога.
18. Символ . - псевдоним текущего.
19. Символ .. - родительского каталога.
20. Команда **pwd** сообщает маршрут к текущему каталогу.
21. Команда **mkdir** создает каталог.
22. Команда **rmdir** уничтожает пустой.
23. Команда **cd** с аргументом позволяет перемещаться по дереву каталогов.
24. Команда **cd** без аргументов перемещает в домашний каталог.
25. Команда **pwd** сообщает маршрут к текущему каталогу.
26. Команда **ls** список подкаталогов и файлов в текущем каталоге

1.2.5 Нулевое домашнее задание.

На освоенном в школе языке программирования

1. Разработать алгоритм расчёта суммы десятичных цифр введенного неотрицательного целого, **НЕ ИСПОЛЬЗУЯ** встроенные функции обработки значений строкового типа.

2. Разработать алгоритм расчёта суммы десятичных цифр введенного вещественного значения из диапазона [0.1, 1), **НЕ ИСПОЛЬЗУЯ** встроенные функции обработки значений строкового типа.

3. Письменно дать прогноз результата, получаемого ЭВМ, при работе программы

```
program testreal;
var a : real;
    i : integer;
begin a:=1.3; for i:=1 to 20 do begin writeln(i,'...',a);
                                      a:=11*a-13
                                 end
end.
```

   и обоснование своего прогноза. Во сколько раз значение двадцатого элемента получаемой последовательности будет отличаться от правильного?

   Программа вычисляет первые 20 элементов последовательности, у которой первый элемент равен **1.3**, а каждый последующий равен разности, полученной вычитанием из предыдущего, умноженного на **11** числа **13**.

4. Массив из 1000 элементов содержит целые числа из отрезка **[0,127]**. Разработать алгоритм подсчета числа элементов, содержащих каждое из чисел диапазона соответственно (выводить значения, встречающиеся более 0 раз, например)

```
  0 встречается  13 раз
 13 встречается  15 раз
 ... ........... ... ...
100 встречается  55 раз
```

5. Оцените правильность результатов получаемых программой

```
program tstsin;
var i : integer;
begin
for i:=1 to 17 do  writeln(i:5, sqr(sin(pi*i))/sqr(i*sin(pi)):20)
end.
```

   и напишите небольшое сочинение по темам, затрагиваемым в ней.

6. Приведённую ниже программу предполагалось использовать для расчёта функции

$$\frac{\mathbf{sin^2(n * pi * x)}}{\mathbf{n^2 * sin(pi * x)^2}}$$

при целых значениях **n**:

```
program test
implicit none
real(4) r, pi
integer n
pi=4*atan(1.0)
write(*,'(10x," n",7x,"r")')
do n=3,17
  r=sin(n*pi)**2/(n*sin(pi))**2
  write(*,*) n, r
enddo
end
```

В процессе тестирования было обнаружено, что при **x=1** получились следующие результаты:

```
 n       r
 3  8.2694441E-03
 4  0.9999999
 5   2.388379
 6  8.2694441E-03
 7   4.703676
 8  0.9999999
 9  8.2694441E-03
10   2.388379
11   7.482878
12  8.2694441E-03
13   1.463577
14   4.703676
15  8.2694450E-03
16  0.9999999
17   3.247618
```

Оцените насколько они правильны и ПОЧЕМУ.

Письменно постарайтесь сформулировать, какие эффекты машинной арифметики демонстрирует эта задача и какие вопросы могут возникнуть после анализа результатов.

Пожалуйста, заполните и сдайте небольшую анкетку.
Она потребуется для получения доступа в астрономический дисплейный класс.

1. Фамилия

   Имя

   Отчество.

2. Год окончания школы.

3. Страна, город, поселок.

4. Номер школы и профиль класса.

5. Операционные системы, с которыми довелось работать.

6. Языки программирования, с которыми довелось работать.

7. Самая простая задача по программированию, решенная Вами полностью.

8. Самая сложная задача по программированию, решенная Вами полностью.

1.3 О перенаправлении ввода/вывода.

1. Стандартные устройства ввода и вывода.
2. Уяснение ситуации.
3. Перенаправление вывода. Программный канал. Конвейер.
4. Немного о въюерах **more** *и* **less**.
5. Команда **cat**.
6. Операции перенаправления. 7. Немного о команде **ls**. *8. Глобальные символы.*

1.3.1 Стандартные устройства ввода и вывода.

Команды **UNIX** связаны со стандартными устройствами:
Stdin (Standard Input) – клавиатура;
Stdout (Standard Output) – экран.
Stderr (Standard Error) – экран.

Их открытие и назначение имён делается автоматически операционной системой. В предыдущих пунктах ими активно пользовались, даже не зная этих имён. Например, после команды **fox kolobok** сообщение **bash: fox: command not found** выведено на экран через устройство **Stderr**, а после команды **cal** фрагмент календаря на текущий месяц выводится через **Stdout**.

1.3.2 Уяснение ситуации.

Наличие двух отдельных устройств вывода даёт возможность по разному организовать обработку **запланированного** и **незапланированного** завершения работы команды.

Запланированное завершение (следствие грамотного задания аргументов и опций при отсутствии небрежности пользователя) означает, что программа получила в выходном файле (то есть, по умолчанию, на экране) результат, в форме ожидаемой пользователем, и никаких неожиданностей (неверной работы или аварийной остановки) при работе команды не произошло.

Незапланированное завершение (следствие небрежности или компьютерной неграмотности) означает, что программа завершила работу по причине неожиданной для пользователя, то есть такой, которая, или неизвестна ему, или, по его мнению, не должна была реализоваться в принципе.

Так после вызова команды **cal 12 2006** получим на экране календарь на декабрь через устройство стандартного вывода **stdout**. Если по невнимательности вызовем команду **cal 11 2006**, то получим календарь на ноябрь. Результат *с точки зрения команды* **cal** – **не аварийный** (месяц с номером **11** в году имеется). Откуда оболочке знать, что правильно набран не тот месяц случайно. Если не заметим опечатку сразу, то возможно слишком поздно обнаружим, что нужная дата, полученная через **stdout**, приходится не на тот день недели. Однако, если печать календаря запрашивается командой **cal 13 2006**, то вызов завершится аварийно выводом на экран фразы:

```
cal: illegal month value: use 1-12
```

через устройство **stderr**. Важно понимать, что весь вывод (и аварийных, и запланированных сообщений) иногда выгодно помещать в один файл, а иногда в разные. Оболочка именно для этого и предоставляет два стандартных устройства вывода. Например, напишем в командной строке

```
cal 11 2006; cal 13 2006; cal 12 2006
```

Поскольку и **stdout**, и **stderr** связаны с одним и тем же физическим устройством – экраном, то результат на экране будет выглядеть так:

```
     Ноября 2006                          !
Вс Пн Вт Ср Чт Пт Сб                      !   Вывод
          1  2  3  4                      !
 5  6  7  8  9 10 11                      !      через
12 13 14 15 16 17 18                      !
19 20 21 22 23 24 25                      !         stdout
26 27 28 29 30                            !
cal: illegal month value: use 1-12    //    <--=   Вывод через stderr
    Декабря 2006                          !
Вс Пн Вт Ср Чт Пт Сб                      !   Вывод
                1  2                      !
 3  4  5  6  7  8  9                      !      через
10 11 12 13 14 15 16                      !
17 18 19 20 21 22 23                      !         stdout
24 25 26 27 28 29 30                      !
31                                        !
```

Вывод информации на экран удобен, когда результат (и запланированный, и аварийный) виден целиком. Однако расшифровка причины аварии может оказаться не столь простой: возникнет желание подумать, например, дома. А тогда не избежать переноса результата на "*флешку*". Короче,

Вывод: Необходимо уметь

1. Переводить стрелку вывода **stdout** с экрана в файл на диске.
2. Переводить стрелку вывода диагностики **stderr** в другой файл на диске.
3. Объединять поток диагностики **stderr** с потоком **stdout**
4. Использовать в качестве стандартного ввода **stdin** файл на диске.
5. Переадресовывать стандартный вывод одной команды на стандартный ввод другой, минуя посредника в виде файла на диске.

Языки программирования типа **ФОРТРАН** или **СИ** позволяют не использовать перечисленные команды переадресации в явном виде. Однако, знание последних в ряде случаев выгодно упрощает тексты исходных файлов и экономит время.

1.3.3 Перенаправление вывода. Программный канал. Конвейер.

Часто результат работы одной команды надо подать на обработку другой. Например, команда получения календаря на весь 2004 год **cal 2004** не оставит на экране данные по первым трем месяцам, а лишь следующую за ними часть календаря. Решить проблему можно двояко:

1. записать результат не на экран, а в файл и потом воспользоваться любым доступным редактором или иным средством обзора содержимого файла, например, командами **more** или **less** (иногда их называют **вьюерами**).

2. перенаправить результат вьюерам **more** или **less** сразу, **минуя посредника** в виде файла.

Каждый из способов хорош по своему: первый позволит хранить результат в файле; второй – не засоряя каталог лишними файлами, тем не менее, даст полный обзор результата. В первом случае результат команды

```
$ cal 2004 > calendar.2004
```

посредством значка $>$ (интерпретируемого оболочкой как приказ перенаправить стандартный поток вывода) окажется выведенным в файл **calendar.2004** текущей директории. Факт появления файла проверим командой **ls**:

```
$ ls
calendar.2004  myf95
```

Если файла не было, то он будет создан. Если же файл существовал, то его содержимое заменится новым результатом. Во втором случае все оказывается еще проще, ведь файл создавать не надо. Достаточно одной строки:

```
$ cal 2004 | more
```

Здесь, после номера года встречается значок (| – вертикальная черта), который интерпретируется оболочкой как приказ перенаправить результат первой команды **cal**, не на экран, а на стандартный вход **stdin** команды **more**, следующей за вертикальной чертой. Возможность такого перенаправления (без какого-либо посредника типа временного файла) дает средство связи (**программный канал**) между выходным потоком одной программы и входным потоком другой.

Соединение **программным каналом** двух или более команд называют **конвейером, или транспортером, или трубой, трубопроводом**. Команды, которые образуют конвейер, работают **одновременно**, а не последовательно. Команды, которые способны воспринимать входные данные непосредственно из программного канала иногда называют **фильтрами**.

В данном параграфе познакомились с примерами реализации возможностей, о которых говорилось в выводах **1** и **5** предыдущего параграфа **1.3.2**.

1.3.4 Немного о вьюерах more и less.

Для обзора содержимого файла проще всего вызвать команду

```
more  calendar.2004
```

(аргумент — имя файла). Нажатие на ввод выведет очередную порцию текста или (если она последняя) — завершит работу команды. Для досрочного прекращения работы **more** нажимаем клавиши **Ctrl** + **d**. Для подвижки по файлу в любом направлении (**more** работает только в одном), применяем команду **less**.

К простейшему вьюеру можно отнести и команду **cat**, которая полезна для вывода текстов, лишь полностью умещающихся на экране (в этом контексте **cat** не конкурент **more** и , тем более, **less**).

1.3.5 Команда cat.

Имя команды происходит от **concatenate** – конкатенация (сцепление). Её простейшее действие: копирование на экран того, что набрано на клавиатуре, после нажатия на клавишу **enter**. Вот счастье–то! Потренируемся для навыка:

```
$ cat
Кто стучится на Руси    ! набрали на клавиатуре и нажали на  enter:
Кто стучится на Руси
Длинным файлом по PC?   ! набрали на клавиатуре и нажали на  enter:
Длинным файлом по PC?
Это он! Это он!         ! набрали на клавиатуре и нажали на  enter:
Это он! Это он!
Электронный почтальон!  ! набрали на клавиатуре и нажали на  enter:
Электронный почтальон!  ! нажали Ctrl + d для выхода из cat.
```

Сама по себе команда **cat** при условии ввода и вывода данных с экрана и на экран абсолютно бесполезна, если не считать радости по поводу дублирования. Однако, в сочетании **cat** с символом $>$ получаем средство создания файла, наполненного данными, которые набирали на клавиатуре. Пробуем:

```
$ cat > mypoem.inp
Кто стучится на Руси          !  Нажали < enter >
Длинным файлом по PC?         !  Нажали < enter >
Это он! Это он!               !  Нажали < enter >
Электронный почтальон!        !  Нажали < enter > и затем  Ctrl + d
```

Образно говоря, здесь команда **cat** совместно с командой перенаправления $>$ выступает в роли простейшего текстового редактора. Проверим – появился ли файл?

```
$ ls                                !
calendar.2004  myf95  mypoem.inp    !       Да!
```

```
$  cat mypoem.inp                    ! Выведем его содержимое
Кто стучится на Руси                 ! (в данной ситуации используем
Длинным файлом по PC?                !  cat как вьюер).
Это он! Это он!                      ! Вывести можно и так:
Электронный почтальон!               !     cat < mypoem.inp
```

Результат тот же. Разница – в способе обработки команд **cat mypoem.inp** и **cat** $<$ **mypoem.inp** оболочкой. В первом случае имя файла **mypoem.inp** – аргумент команды **cat**; во втором, оболочка **переназначает стандартный ввод** команды **cat**. Комбинируя **cat** с переназначением стандартных ввода и вывода, можно копировать один файл в другой. Например,

```
$ cat < mypoem.inp > mypoem2.inp
$ ls
calendar.2004  myf95  mypoem2.inp  mypoem.inp
```

Команда **cat** позволяет копировать и несколько файлов в один. Создадим посредством **cat** еще один короткий файл с именем **mypoem3**:

```
$ cat > mypoem3.inp
И международная
Почта электронная,
Скорая, метельная,
Истинно E-мельная,
Предоставит наслажденье,
Стоит только пожелать,
Напечатать выраженье
И кому-нибудь послать.
```

Объединим файлы **mypoem3.inp** и **mypoem2.inp**

```
$ cat mypoem3.inp mypoem2.inp > mypoem.inp
$ cat < mypoem.inp
И международная
Почта электронная,
Скорая, метельная,
Истинно E-мельная,
Предоставит наслажденье,
Стоит только пожелать,
Напечатать выраженье
И кому-нибудь послать.

Кто стучится на Руси
Длинным файлом по PC?
Это он! Это он!
Электронный почтальон!
```

Таким образом, познакомились не только с командой **cat**, но и научились реализовывать пункт **4** из выводов параграфа **1.3.2**

1.3.6 Операции перенаправления.

На самом деле операция переключения вывода, обозначаемая значком > может быть записана чуть иначе:

```
cal 09 2006 1> mysept     ! раньше использовали:  cal 09 2006 > mysept
```

Единица, стоящая перед значком > (без пробела!) – это частное значение так называемого **дескриптора файла** (небольшое целое число), которое используется для идентификации файла. По умолчанию при запуске программа получает три (уже упоминавшихся) открытых файла: **stdin**, **stdout** и **stderr**, которым сопоставляются дескрипторы: **0**, **1** и **2** соответственно. Запись **cal 09 2006 1> mysept** *с точки зрения* оболочки означает переадресацию файла с дескриптором **1** на файл с именем **mysept**. В записи справа идексный дескриптор **1** подразумевается по умолчанию.

Аналогично, **cal 09 2006 2> mydia** означает перенаправление стандартного потока диагностики в файл с именем **mydia**, который хоть и будет создан, но окажется пустым, поскольку никакая диагностика не генерируется, а календарь на сентябрь выведется на экран: ведь переназначение касалось исключительно **stderr**.

Если же запущена команда **cal 13 2006 2> mydia**, то на экран через **stdout** не выведется ничего, а в файле **mydia** появится текст: **cal: illegal month value: use 1-12**, поскольку оболочка переадресовала вывод диагностики.

Направление диагностических сообщения в файл на диске при диалоге с оболочкой создаёт иллюзию нормального завершения работы команды, что опасно.

Однако, как уже говорилось в параграфе **1.3.2**, бывают ситуации, когда сразу понять причину авариного завершения программы не удается. Для уяснения происходящего выгодно содержимое **stderr** направить не на экран, а в **stdout**, с тем, чтобы вся информация (и запланированная, и аварийная) была выведена из **stdout** в единый файл. Указание интерпретатору о слиянии потока стандартной диагностики с потоком стандартного вывода записывается посредством обозначения **2>&1**. Например,

```
  cal 09 2006 > mytest 2>&1      ! или cal 09 2006 1> mytest 2>&1
```

Написанное означает, что стандартный вывод команды **cal** необходимо направить в файл **mytest**. При этом поток стандартного вывода может заменится на поток стандартной диагностики, если таковая появится. Таким образом, в случае правильного задания номера месяца, как в последнем примере, содержимое файла **mytest** окажется требуемым календарем:

```
    Сентября 2006
Вс Пн Вт Ср Чт Пт Сб
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Если же даем одну из команд

```
cal 13 2006 > mytest 2>&1        ! или cal 13 2006 1> mytest 2>&1
```

то файл **mytest** будет содержать диагностическое сообщение, перенаправленное из **stderr** в **stdout**:

```
cal: illegal month value: use 1-12
```

Заметим, что запись

```
cal 13 2006 2>&1 > mytest   НЕ ЭКВИВАЛЕНТНА !!! cal 13 2006 > mytest 2>&1
```

В первой записи команда узнает о перенаправлении потока диагностики в стандартный выходной поток, когда последний еще не переадресован на файл **mytest**. Иначе говоря, до момента перенаправления в файл **stderr** связано с экраном. Поэтому, содержимое диагностики через **stderr** выводится на экран, а в файл **mytest** ничего не посылается. Во второй записи команда узнает о переадресации стандартного вывода раньше чем о перенаправлении потока диагностики. Поэтому к моменту последнего стандартный вывод уже переадресован.

```
cal 04 2006 >& mytest   эквивалентно cal 04 2006 > mytest 2>&1
```

При необходимости дополнять файл результата достаточно вместо значка $>$ перед именем файла использовать $>>$

```
$ cal 09 2006 >> myerr 2>&1
$ cal 13 2006 >> myerr 2>&1
$ cal 05 2006 >> myerr 2>&1
```

Содержимое файла **myerr**:

```
      Мая 2006
Вс Пн Вт Ср Чт Пт Сб
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31

cal: illegal month value: use 1-12
    Октября 2006
Вс Пн Вт Ср Чт Пт Сб
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

Краткая сводная таблица символов переадресации.

Операция	Формат ее записи	Действие операции
>	команда > файл команда **1**> файл	Переключение стандартного вывода команды в указанный файл.
>>	команда >> файл команда **1** >> файл	Перенаправление стандартного вывода команды в *хвост* указанного файла.
<	команда < файл	Указанный файл используется в качестве стандартного ввода команды
|	команда1 | команда2	Выполняется первая команда и ее стандартный вывод подается на стандартный вход второй.
2>	команда **2**> файл	Переключение вывода стандартной диагностики в указанный файл.
2>>	команда **2**>> файл	Добавление стандартной диагностики в *хвост* указанного файла.
&> >**&**	команда **&**> файл команда >**&** файл команда > файл **2**>**&1** команда **1**> файл **2**>**&1**	Переадресация и **stdout**, и **stderr** в один и тот же поток. Тот же самый результат. Тот же самый результат.
	команда >> файл **2**>**&1**	Аналогично, но с добавлением в *хвост* указанного файла.
tee -i	команда| **tee -i** файл	Одновременная запись стандартного вывода команды и на экран и в файл, игнорируя сигналы прерывания
tee -ia	команда| **tee -a** файл	Аналогично, но с добавлением записи в *хвост* файла.

Без команды **tee** не возможен стандартный вывод в файл с одновременной записью его содержимого на экран. Например, конвейер

```
cal 02 2006 | tee feb2006
```

выведет календарь на февраль месяц и на экран и в файл **feb2006**. А конвейер

```
spell feb2006 | tee qq
```

обеспечит вывод ошибочных *с точки зрения машины* слов из файла **feb2006** не только на экран, но и в файл **qq**. Подробнее о команде **spell** можно узнать, например, из [5]. В литературе и в справках, выдаваемых командой **man** , иногда можно встретить заключение букв некоторых опций в квадратные скобки. Например, **tee -i[a] qqq**. Сами скобки и их информационная нагрузка в данном случае предназначены исключительно для человека. Они означают, что опция, обозначаемая буквой **a**, не необходима по синтаксису команды. Явное употребление таких скобок при написании команды сразу приведет к синтаксической ошибке.

1.3.7 Немного о команде ls.

Команда **ls** высвечивает имена файлов и подкаталогов текущего каталога:

```
$ ls
calendar.2004  myf95  mypoem2.inp  mypoem3.inp  mypoem.inp
```

Более подробную справку о содержимом текущего каталога можно получить, вызывая команду **ls** с опцией **-l**:

```
$ ls -l
итого 32
-rw-r--r--    1 aw       users        1961 Окт  2 18:19 calendar.2004
drwxr-xr-x    2 aw       users        4096 Сен 29 17:56 myf77
-rw-r--r--    1 aw       users          83 Окт  5 18:20 mypoem2.inp
-rw-r--r--    1 aw       users         157 Окт  5 18:28 mypoem3.inp
-rw-r--r--    1 aw       users         248 Окт  5 18:36 mypoem.inp
```

Из нее, в частности, видно, что **myf95** – поддиректория текущего каталога (по наличию литеры **d** в колонке идентификации типа файла), а все остальные имена – это имена обычных файлов. Узнать, что **myf95** – имя подкаталога можно и по цвету имени, дав команду **ls** без опций. Подробно о справке, получаемой с опцией **-l**, узнаем в курсе "Операционные системы". Для сохранения в файле результата работы команды **ls -l** можно переадресовать стандартный вывод **ls** на вывод в файл:

```
$ ls -l > listmyfirst
```

Вывести содержимое полученного файла можно вьюером **cat** (выгоднее **less**):

```
bash-2.05b$ cat listmyfirst         высветка содержимого файла
итого 20
-rw-r--r--    1 aw       users        1961 Окт  2 18:19 calendar.2004
-rw-r--r--    1 aw       users           0 Окт  6 17:35 listmyfirst
drwxr-xr-x    2 aw       users        4096 Сен 29 17:56 myf77
-rw-r--r--    1 aw       users          83 Окт  5 18:20 mypoem2.inp
-rw-r--r--    1 aw       users         157 Окт  5 18:28 mypoem3.inp
-rw-r--r--    1 aw       users         248 Окт  5 18:36 mypoem.inp
```

Можно добавить в хвост файла listmyfirst имя текущего каталога, например, так

```
$ pwd >> listmyfirst
$ cat listmyfirst
итого 20
-rw-r--r--    1 aw       users        1961 Окт  2 18:19 calendar.2004
-rw-r--r--    1 aw       users           0 Окт  6 17:35 listmyfirst
drwxr-xr-x    2 aw       users        4096 Сен 29 17:56 myf77
-rw-r--r--    1 aw       users          83 Окт  5 18:20 mypoem2.inp
-rw-r--r--    1 aw       users         157 Окт  5 18:28 mypoem3.inp
-rw-r--r--    1 aw       users         248 Окт  5 18:36 mypoem.inp
/home/aw/lecture/myfirst
```

Имена всех файлов можно упорядочить по времени последней модификации файла, так что самые новые окажутся первыми, включив опцию **-t**:

```
$ ls -t
listmyfirst  mypoem.inp  mypoem3.inp  mypoem2.inp  calendar.2004  myf77
$
```

Вообще говоря, команда **ls** имеет массу опций. Как написано в справочнике [5] команда **ls** одновременно является и одной из самых простых, и одной из самых сложных. Оперативно узнать о её возможностях (и о возможностях других команд) можно, например, посредством команд **man имя_команды** и/или **info имя_команды**.

1.3.8 Глобальные символы.

Наличие оболочки позволяет указать целую группу файлов посредством так называемого **шаблона** имени файла. Вместо термина **шаблон** часто используются его синонимы : **глобальный символ**, **трафарет**, **wildcard**. Шаблоны удобны.

Например, хотим высветить список всех файлов с расширением **inp**. Указать в аргументе команды **ls**, что в имени файла до точки нас устроит любое имя, можно посредством шаблона, символа *, смысловая нагрузка которого – произвольная (в том числе и пустая) комбинация символов. Включим его в имя файла-аргумента команды **ls**:

```
$ ls *.inp
mypoem2.inp  mypoem3.inp  mypoem.inp
$
```

Оболочка подаст на вход команде **ls** каждое из имен текущей директории, заканчивающееся на **.inp**.

К **глобальным** символам относятся, как раз, все те символы, которые в пункте **1.2** не рекомендовалось использовать в именах файлов (в качестве обычных символов) именно потому, что это **символы–шаблоны** глобального назначения (**глобальные**).

Шаблон **?** обозначает любой символ в той позиции в имени файла, где находится. Поэтому, например, задав аргумент команды **ls** в виде **calendar.200?** получим

```
b$ ls calendar.200?
calendar.2004
```

Если бы в каталог содержал много кадендарей в пределах первого десятка лет, то были бы выведены имена их всех.

1.4 Чуть-чуть еще о некоторых командах.

1. Команда удаления – **rm**.
2. Команда копирования – **cp**.
*3. Команда перемещения –***mv**.
4. Команда подсчета строк, слов и символов **wc**
5. Кое-что о текстовых редакторах.
6. О чем узнали из третьего параграфа (конспективно).

Обычно задается вопрос:“ Зачем знать команды копирования, перемещения и удаления файлов, если Midnight Commander, не требуя от человека знания правил записи упомянутых команд, выполняет нужное?”.

Ответ: Диалог с ЭВМ ведет не только человек, сидящий за терминалом. Например, копирование или уничтожение файла может потребоваться и программам, написанным на языке оболочки (так называемым скриптам). Странно, если для безусловного удаления файла вызывается Midnight Commander, требующий подключения к работе программы человека для нажатия клавиш клавиатуры, которое вызовет команду удаления, хотя гораздо проще, быстрее и надежнее сразу написать в скрипте эту команду без вызова посредников.

1.4.1 Команда rm.

Команда **rm** (от **remove**) служит для удаления файлов и каталогов. Наиболее безопасный вариант ее использования – включение опции **-i**.

```
$ rm -i mypoem?.inp
rm: удалить обычный файл ‘mypoem2.inp’? y
rm: удалить обычный файл ‘mypoem3.inp’? y
$ ls
calendar.2004  listmyfirst  myf95  mypoem.inp
$
```

Здесь у команды **rm** помимо задания опции страховки **-i** в имени аргумента (файла) снова использован символ-шаблон **?** (знак вопроса). Последующая команда **ls** подтверждает отсутствие удаленных файлов в текущей директории **myfirst**.

Особенно опасен вариант

```
$ rm -r CATALOG    удалит весь каталог CATALOG со всеми
$                  вложенными в него подкаталогами и файлами.
```

Более безопасно

```
$ rm -ir CATALOG    удаление с подтверждением
$                   (наряду с -r включена опция -i)
```

Более мягкий способ удаления каталогов – команда **rmdir**, которая удалит каталог лишь при полном отсутствии в нем файлов.

1.4.2 Команда cp.

Команда **cp** предназначена для копирования файлов. Например, в результате выполнения команды

```
$ cp mypoem.inp mypoemnew.inp
$ ls
calendar.2004  listmyfirst  myf95  mypoem.inp  mypoemnew.inp
```

в текущем каталоге **myfirst** под именем **mypoemnew.inp** появится копия файла **mypoem.inp**. Копирование файла под новым именем удобно в качестве средства создания файла для набора новой или модификации старой программы.

1.4.3 Команда mv.

Команда **mv** предназначена для переноса файла из одного каталога в другой, либо для переименования файла в пределах одного каталога. Например, в результате выполнения команды

```
$ cp mypoem.inp mypoemnew.inp
$ mv mypoemnew.inp mypoem.new
$ ls
calendar.2004  listmyfirst  myf95  mypoem.inp  mypoem.new
$
```

длинное имя **mypoemnew.inp** заменится на более короткое **mypoem.new**.

1.4.4 Команда wc.

Команда **wc** относится к группе команд, нацеленных на обработку текста. При написании скриптов часто приходится анализировать текстовые строки, например, выделять составные части имен файлов, и т.д. Команда **wc** подсчитывает число строк, слов и символов в одном или нескольких файлах. Используем в качестве аргумента для нее файл с именем **feb2006**, который содержит календарь, уже полученный выше.

```
    Содержимое файла feb2006     ! Пример обработки файла командой wc
---------------------------------!-----------------------------------
    Февраля 2006                 !      $ wc feb2006
Вс Пн Вт Ср Чт Пт Сб             !        8  37 133 feb2006
          1  2  3  4             !-----------------------------------
 5  6  7  8  9 10 11             !  Здесь 8 - количество строк;
12 13 14 15 16 17 18             !        37- количество слов;
19 20 21 22 23 24 25             !       133- количество символов.
26 27 28                         !
                                 !
```

При подсчете количества символов надо учитывать, что каждая строка завершается символом перевода строки, который трактуется **wc** как отдельный символ, хотя и невидимый человеку. Кроме того, после слова (**сб**) в строке находится еще один пробел, так что $\mathbf{133}=\mathbf{17}+\mathbf{22}+\mathbf{21}+\mathbf{21}+\mathbf{21}+\mathbf{21}+\mathbf{9}+\mathbf{1}$.

1.4.5 Немного о текстовых редакторах.

Имеется много редакторов: **ed, vi, joe, gedit, mcedit, emacs**. Команды **man** и **info** позволяют быстро находить справки по многим командам, в частности, и по любому из указанных редакторов. **emacs** – наиболее мощный из них. Краткий обзор назначений клавиш, используемых в нем, и технологии редактирования можно найти, например, в [29].

Редактор **mcedit** – это встроенный редактор команды **mc**, привычной для людей старшего поколения. Наиболее удобный режим работы с **mcedit** через среду команды **mc**, вызванной из командной строки консоли или терминала. Расположим подсвечивающий курсор **mc** на строке с именем редактируемого файла. Тогда нажатие функциональной клавиши **F4** приведет к вызову редактора и высветке в его окне текста из файла. После внесения и сохранения исправлений переключение с окна редактора на окно консоли (или обратно) достигается нажатием клавиш **Ctrl** и **o**. Теперь из командной строки можно, например, вызвать нужный транслятор.

Если нужного файла нет (хотим создать новый), то достаточно, нажав клавишу **Shift**, и, удерживая ее, нажать дополнительно клавишу **F4**. Появится пустое окно, в котором можно набирать текст. Для сохранения набранного с выходом из редактора дважды нажимаем клавишу **Esc**, что приводит к запросу о явном подтверждении нашего намерения. Предоставляется три варианта ответа: **Cancel quit**, **Да** и **Нет**. Первый (**Отменить выход**), действующий по умолчанию, подсвечен. При нажатии на клавишу **enter** (или левую клавишу мыши) снова возвращаемся в редактируемый текст (иначе говоря, решили еще что-то подредактировать). Если решаем, что набранное необходимо запомнить в файле, то посредством клавиши табуляции или (опять же левой клавиши мыши) выбираем в качестве нужного ответа второй и после нажатия на **enter** получаем возможность набрать желаемое имя нового файла. Очередное нажатие на **enter** инициирует создание файла с нужным именем и выход из редактора. Если же обнаружили, что все только что набранное ошибочно (то есть мы внося изменение находились в невменяемом состоянии и изменять файл на самом деле было не нужно), то выбираем третий ответ.

После входа в редактор в нижней строке высвечивается смысловая нагрузка функциональных клавиш. В частности, за клавишей **F9** закреплен выход в главное меню **mc**. При наборе текста полезно достаточно часто нажимать клавишу **F2** – сохранение отредактированного текста в текущей директории.

Для копирования или перемещения части текста в пределах редактируемого файла необходимо эту часть выделить, указав ее границы (выделенную часть называют блоком). Курсор подводят к начальному символу предполагаемого блока и нажимают клавишу **F3**, затем курсор переводят на последний символ блока и помечают опять же клавишей **F3**. Выделенный текст автоматически подсвечивается. После курсор переводится на ту позицию файла, начиная с которой требуется расположить выделенный текст. Нажатие клавиши **F5** приведет к копированию, клавиши **F6** – перемещению, клавиши **F8** – удалению блока.

О выделении прямоугольных блоков и о копировании блоков из одного файла в другой см. пункты **1.8.15–1.8.18** из параграфа **1.8 Ответы на часто задаваемые вопросы (FAQ)**

1.4.6 О чем узнали из третьего и четвертого параграфов (кратко).

1. Стандартные устройства ввода/вывода: **stdin, stdout,stderr**.
2. Программный канал – средство связи между потоками.
3. Символ $>$ – переназначение стандартного вывода в файл .
4. Символ $<$ – переназначения стандартного ввода из файла .
5. Слово $>>$ – добавление стандартного вывода в *хвост* файла.
6. Слово $\mathbf{2} > \&\mathbf{1}$ – переадресация и **stdout**, и **stderr** в один и тот же поток.
7. Операция $|$ перенаправления стандартного вывода одной команды на стандартный ввод другой.
8. Конвейер – соединение нескольких команд на базе операции $|$.
9. Команды (вьюеры) **more**, **less**, **cat**.
10. Командой **cat** в сочетании со словами $>$ и $<$ можно смоделировать простейший редактор.
11. Команда **ls** – список содержимого текущей директории.
12. Команда **spell** – поиск ошибок в словах.
13. Команда **tee** – одновременный вывод содержимого стандартного ввода и на экран и в требуемый файл.
14. Смысл заключения некоторых опций при описании синтаксиса команд в квадратные скобки.
15. Глобальные символы ***** и **?**.
16. Команда **rm** – уничтожение файлов и каталогов.
17. Команда **cp** – копирование файлов и каталогов.
18. Команда **mv** – пересылка файлов и каталогов.
19. Команда **wc** – подсчет числа строк, слов и символов в указанных файлах.
20. Имена некоторых из имеющихся в системе редакторов.
21. Об элементарной работе с редактором **mcedit** из среды **mc**.

1.5 Простая линейная программа на ФОРТРАНе.

1. Текст программы на очень старом ФОРТРАНе.
2. Текст программы на ФОРТРАНе–77.
3. Текст программы на ФОРТРАНе–95.
4. О правиле умолчания ФОРТРАНа.
5. Выполняемые и невыполняемые операторы.
6. Оператор присваивания.
7. Немного об операторе вывода WRITE.
8. Немного об операторе FORMAT.
9. Программная единица ФОРТРАНа и оператор ее завершения.
10. О чем узнали из пятого параграфа (кратко)

Познакомимся с элементарным синтаксисом записи программ на языках программирования ФОРТРАН–77, ФОРТРАН–95, СИ и С++ на примере программы, которая

используя четыре переменные вещественного типа, вычисляет для аргумента x=10 величины y, z, t по формулам: y=1+x; z=y-1; t=z/x.

Задача не требует нарушения порядка выполнения приведенных формул. Такие программы называют **линейными**. Линейные программы реализуются алгоритмической структурой – **следование**. Задача, безусловно, проста. В курсовых и дипломных работах формулы, программируемые для вычислений, несомненно будут гораздо сложнее. Тем не менее, на данной задаче можно приобрести не только некоторый навык по синтаксису и технологии пропуска программ, но и обнаружить некоторые неожиданности там, где никаких вопросов изначально не возникало.

1.5.1 Текст программы на очень старом ФОРТРАНе.

```
c  Буква 'c' или 'C' в первой колонке - признак комментария.
C  Комментарии в программах пишутся для человека. Поэтому
C  они должны не перефразировать операторы программы, а
c  выявлять их смысловое проблемное толкование.
c  Здесь комментарии нацелены на объяснение правил их записи
c  и никакого отношения к проблемной сути примера не имеют.
      x=10
      y=1+x
      z=y-1
      t=z/x
      write(*,*) x, y, z, t
      end
```

Текст программы дан в **фиксированном формате** (иной возможности на старых версиях ФОРТРАНа не было). Это означает, что любая строка состоит из нескольких неизменных по размеру и местоположению полей, в которые можно помещать лишь соответствующие им элементы текста программы, то есть запись последней на экране в некоторой мере фиксирована. Именно, колонки с 1-й по 5-ю отводятся

для размещения меток (в данной программе меток нет); операторы размещаются в колонках с 7-й по 72-ю (причем по одному оператору в строке). Колонка 6 называется **колонкой продолжения**, ибо присутствие в ней символа отличного от пробела или нуля означает, что содержимое строки должно трактоваться как продолжение оператора из предыдущей.

1.5.2 Текст программы на ФОРТРАНе–77.

```
c   1) Признаком оператора комментария в ФОРТРАНе-77 служат не только
C      символы 'c' и 'C' в первой колонке, но и символ '*'.

*   2) Допускается наличие в исходном тексте программы пустых строк,
*      что повышает ее удобочитаемость.

      !  3) В  качестве признака начала комментария после 6-ой колонки
      !     можно  использовать символ '!' (восклицательный знак).
c.....................................................
c234567890123456...<--= нумерация первых 16 колонок экрана
c.....................................................
      x=10             ! Восклицательный знак, помещенный
      y=1+x            ! в любом месте строки после оператора
      z=y-1            ! ФОРТРАН-программы, означает, что после
      t=z/x                 ! '!' и до окончания строки следует
      write(*,*) x, y, z, t  ! комментарий ( по сути '!' - это
      end                    ! аналог // в СИ).
```

1.5.3 Текст программы на ФОРТРАНе–95.

Программа, написанная на ФОРТРАНе-77, без каких-либо переделок должна пройти и на ФОРТРАНе-95. Однако, программистский инструментарий ФОРТРАНа-95 гораздо богаче. Допустима запись программы в **свободном формате**, когда ограничения, определяющие **фиксированный формат**, отменены (в частности, ФОРТРАН-операторы могут располагаться с любой позиции строки). Например,

```
  x=10;  y=1+x;                      ! ФОРТРАН-90 и ФОРТРАН-95 позволяют
  z=y-1; t=z/x; write(*,*) x, y,&    ! использовать   с в о б о д н ы й
                           z, t      ! формат записи текста программы
  end
```

Ключом формата записи программы служит расширение файла:

Формат	Расширение файла
Фиксированный	.f .for .F .FOR
Свободный	.f90 .f95 .F90 .F95

1.5.4 О правиле умолчания ФОРТРАНа.

Правило умолчания ФОРТРАНа (если его не отменять и не модифицировать):

“Любое имя, начинающееся с букв i, j, k, l, m, n, толкуется транслятором, как имя (переменной или функции), нацеленное на работу с данными целого типа (integer), а все остальные имена (начинающиеся с любой другой буквы), – соответствуют данным вещественного типа (real).

Так, в приведенных выше программах все имена (**x**, **y**, **z**, **t**) – имена **переменных** вещественного типа, переменных целого типа нет, хотя значения целого типа встречаются – это **константы** единица и десятка.

Чуть–чуть не по теме пункта, но важно.

Важно осознать, что в ФОРТРАНе (и в СИ) внутренее машинное представление данных целого и вещественного типов принципиально различно. Поэтому арифметические операции $+ \quad - \quad * \quad /$ (сложение, вычитание, умножение, деление), хотя и обозначаются для типов **integer** и **real** одинаковыми значками, аппаратно выполняются по-разному. Так, например, операция деления при целочисленных операндах понимается как деление нацело, то есть **5/3=1**, а **5.0/3.0=1.6666667**. Особенно забавно выглядит в ФОРТРАНе результат операции возведения в степень, обозначаемой двумя **звездочками**, следующими сразу друг за другом: **10**(-3)=0**.

Выгодно или нет пользоваться правилом умолчания?

Сегодня его прежнее достоинство – меньший размер программы из-за отсутствия описаний переменных – весьма сомнительно. Самое неприятное последствие его применения – возможность **не заметить опечатку в имени переменной**, хотя многие трансляторы автоматически предупреждают об именах переменных, которым к моменту использования не присвоено никакого значения. В СИ аналогичного правила умолчания НЕТ.

ФОРТРАН позволяет явно указывать тип переменной (например, **real x, y, z**). Однако при действии правила умолчания такое описание не гарантирует нас от опечаток в именах переменных типа **real** или **integer**. Поэтому при работе со старыми ФОРТРАН-программами помнить о правиле умолчания просто необходимо. Явное описание переменных в ФОРТРАНе **при действии правила умолчания** оправдано, если востребованы типы отличные от **integer** и **real**, например, **logical** или **complex**.

Наряду с явным описанием ФОРТРАН предоставляет возможность и неявного описания через оператор **implicit** (неявного в том смысле, что не нужно указывать имя переменной полностью), который позволяет изменить правило умолчания для нужного диапазона переменных и даже *полностью отключить действие правила*. Так, оператор **implicit real*8 (a-h,o-z)** сопоставит любой переменной вещественного типа восьмибайтовую ячейку, не отключая четырехбайтового представления переменных типа **integer**; а оператор **implicit none** – отменит действие правила умолчания полностью, т.е. в программе придется описывать все нужные имена, как в СИ или в ПАСКАЛе.

Наше кредо: ВСЕГДА отключаем правило умолчания.

Текст программы решающей нашу задачу в этом случае примет вид:

```
program myfirst   ! Оператор заголовка главной программы.
implicit none     ! О т к л ю ч е н и е  правила умолчания.
real*8 x, y, z, t ! Явное объявление используемых имен и их типа
x=0.1d0
y=1+x
z=y-1
t=z/x
write(*,*) x, y, z, t
end
```

По сравнению с предыдущими вариантами в последнем не только отключено правило умолчания, но и появился оператор заголовка программы **program**. В принципе он не обязателен, но дисциплинирует: заголовок психологически повышает ответственность. Имя заголовка должно всегда начинаться с буквы (как и имя переменной). Заметим, что в операторе явного описания указан тип **real*8**. Это означает, что под каждую из указанных далее переменных отводится восьмибайтовая ячейка. Если бы хотели отвести четырехбайтовую, то могли бы указать в качестве типа **real*4** или даже просто **real**. ФОРТРАН-95 наряду с указанной формой объявления типа имеет и альтернативный вариант, именно **real(4)** или **real(8)**. Так что после упомянутых модификаций та же программа на ФОРТРАНе-95 может иметь вид:

```
program myfirst
implicit none
real(8) x, y, z, t
  x=10;  y=1+x;  z=y-1;   t=z/x;
  write(*,*) x, y, z, t
end
```

1.5.5 Выполняемые и невыполняемые операторы.

Первые три разобранных оператора: **program**, **implicit**, **real** – это операторы описания. Они необходимы компилятору для адресации и выбора типа машинных команд. Так что результат перевода, уже построенный на их учете, ни содержать, ни, тем более, выполнять машинных аналогов ФОРТРАН-команд **program**, **implicit**, **real** не будет. Такие операторы языка ФОРТРАН называют **невыполняемыми** (в смысле оттранслированной программой). Все операторы описания – **невыполняемые**. Наряду с ними программа содержит и **выполняемые** операторы – те действия, которые мы явно назначили ей выполнить. В данной задаче к ним относятся операции **сложения**, **вычитания**, **деления**, **присваивания**, а также оператор печати.

Первые две имеют общепринятое в математике обозначение $+$ и **-**; **деление** обозначается значком $/$ (иногда его называют "прямой слэш"). Операция умножения (в данной программе ее нет) обозначается символом ***** (**звездочка**).

Наиболее употребительной операцией является **операция присваивания**, которая (и в ФОРТРАНе, и в СИ) обозначается значком $=$. Последний в математике используется для обозначения операции отношения **равно**. Именно поэтому последняя операция в ФОРТРАНе-77 обозначается **.eq.** , а в СИ и C++ через $==$ (ФОРТРАН-95 допускает оба последних обозначения).

1.5.6 Оператор присваивания.

В **ФОРТРАН-программе** из пункта **1.5.4** – четыре оператора присваивания. Первый – пересылка числа десять в переменную с именем **x** (на русском языке читается: **x** присвоить 10). До выполнения пересылки значение переменной **x** не определено.

В **СИ-программах** операция присваивания помимо выполнения пересылки еще приобретает и значение равное присваиваемому. Так что в СИ можно одним предложением выполнить несколько операций присваивания:

```
float x,y,z,t;
x=y=z=t=10;        // в ФОРТРАНе так нельзя !!!
```

Оператор присваивания выполняется справа налево.

1.5.7 Немного об операторе вывода WRITE.

Фортран имеет два оператора вывода:

print – на экран (стандартное устройство вывода);

write – либо на экран, либо в файл.

Будем использовать **write**, чтобы привыкнуть к наиболее общей форме. В программе в круглых скобках после **write** видим две звездочки, разделенные запятыми.

Первая звездочка – это псевдоним устройства вывода на экран.

Первый параметр оператора **write** всегда указывает **куда** выводятся данные. Для вывода не на экран вместо первой звездочки можно указать, например, константу целого типа, которая *"понимается"* компилятором, как номер некоего логического устройства вывода, которое может направить данные и на экран, и в файл какой-то директории, и даже в область оперативной памяти. Выводом на экран обычно ведает устройство с номером **6**, вводом с экрана — устройство с номером **5**. Остальные номера соответствуют файлам на диске. Вообще для вывода в файл во многих случаях можно обойтись и *****, если при запуске программы переназначить ей стандартный вывод.

Второй параметр (тоже звездочка) – это условное обозначение правила приведения выводимых данных к нужному нам виду, которое означает, что выбираем самую непритязательную форму вывода (*по типу данного*). Нас не интересует: с какой именно колонки строки печатается результат, безразлична, до некоторой степени, и форма вывода (лишь бы было понятно). После закрывающейся круглой скобки указываем имена переменных, содержимое которых хотим вывести:

write(*,*) x, y, z, t

Здесь **x, y, z, t** – это **список вывода**.

Именно типы печатаемых данных и определяют стандарт печати, на который нацеливает вторая звездочка. В нашем случае у всех печатаемых переменных тип один: **real**. Значения данных этого типа в текстах ФОРТРАН-программ могут записываться в двух формах (с фиксированной запятой и с плавающей). Посмотрим на стандарт, выбранный компилятором **g77**:

```
 10.  11.  10.  1.
```

Вполне приемлемая для нас пока форма с фиксированной запятой. Если в программе аргументу **x** присваивается значение $\mathbf{10^{-7}}$, то по **второй звездочке** выводится

```
1.00000001E-07  1.00000012  1.1920929E-07  1.1920929
```

Видим, что малые числа (порядка $\mathbf{10^{-7}}$) отпечатались с порядком (**E-форма**, форма с плавающей запятой), а числа порядка единицы в обычном виде (**F-форма**, форма с фиксированной запятой). Результат для $\mathbf{x = 10^{-8}}$ таков:

```
9.99999994E-09  1.  0.  0.
```

Заметим, что при $\mathbf{x = 10^{-7}}$ и $\mathbf{x = 10^{-8}}$ значения переменных **z** и **t**, вычисленные ЭВМ, оказываются сомнительными. (*Почему?*). Осмысление результатов более удобно проводить на основе сводных табличек в пункте **1.7.7 - Тестирование программ** из параграфа **1.7 – В первый раз в дисплейный класс**.

1.5.8 Немного об операторе FORMAT.

Как уже знаем, **вторая звездочка** в операторе **write(*,*)** задает некое стандартное правило вывода данных. Вообще, ФОРТРАН для задания формы вывода предоставляет средство – оператор **format**, который позволяет однозначно указать в какие позиции строки и в каком виде желаем вывести каждое данное из списка вывода.

```
      program myfirst     ! Пример использования оператора format
      implicit none       ! для вывода данных.
      real x, y, z, t     !
      x=1e-7
      y=1+x
      z=y-1
      t=z/x
      write(*,1000) x, y, z, t
 1000 format(1x,e15.7,3x,f10.7,3x,e15.7,3x,f10.7)
      end
```

Здесь оператор вывода вместо второй звездочки содержит целое число без знака. Это число (его придумывает пишущий программу) служит меткой строки с оператором **format**, из которого оператор **write** черпает информацию о нужных преобразованиях выводимых данных.

Содержимое оператора **format** посредством некоторого условного языка (**спецификации формата**) задает форму (шаблон, форматную строку) представления выводимых данных. Правило преобразования конкретного выводимого данного указывается **дескриптором преобразования**.

Спецификация формата состоит из заключенного в круглые скобки списка **дескрипторов**. В данной программе в операторе **format** встречаются **дескрипторы**: **1x**, **3x**, **e10.3**, **e15.7**, **f10.7**.

Посмотрим на результат работы программы:

```
0.100E+02   11.0000000     0.1000000E+02    1.0000000
```

Видим четыре числа в строке, как и должно быть по списку вывода. Первое и третье число выведены в форме с плавающей запятой (**E-форме**, есть порядок числа). Дескриптор вида **e** выводит данные типа **real** в форме с плавающей запятой, а дескриптор вида **f** – в форме с фиксированной запятой (**F-форме**).

Буквы **f** и **e** задают правило преобразования.

Первое число после буквы – количество позиций для размещения данного.

Второе число – количество цифр после десятичной точки в записи данного.

Дескриптор **1x** означает один пробел в форматной строке; **3x** – три пробела.

Дескриптор, задаваемый литерами **I5** можно использовать для вывода не более чем пятизначного положительного или четырехзначного отрицательного целого.

Количество позиций, отводимых дескриптором, должно хватать для размещения данного (иначе в соответствующих позициях строки отпечатается символ *****).

Когда спецификация формата невелика, то зачастую удобно размещать ее, заключенную в апострофы, вместо второй звездочки. оператора **write**. Например,

```
write(*,'(1x,e10.3,3x,f10.7,3x,e15.7,3x,f10.7)') x, y, z, t
```

В этом случае ни метка, ни отдельная строка для оператора **format** не нужны. Если же спецификация формата занимает несколько строк, которые загромождают логическую структуру программы многочисленными подробностями вывода, то, использование оператора **format** с меткой, видимо, предпочтительнее.

Оператор **format** невыполняемый. Поэтому может располагаться достаточно произвольно по отношению к использующему его оператору вывода или ввода (например, до или после него, в начале программной единицы или перед завершающим программную единицу оператором **end**.

1.5.9 Программная единица ФОРТРАНа и оператор ее завершения.

Оператор **end** – оператор синтаксичсеского завершения любой единицы компиляции ФОРТРАНа (в том смысле, что операторы, расположенные после **end** уже не будут принадлежать единице компиляции, завершенной **end**.

Программные единица (единица компиляции) – это то, что воспринимается транслятором, как единое целое, и может быть оттранслировано независимо. В простейшем случае, как сейчас, – это одна главная программа. В ФОРТРАНе помимо главной программы **program** есть несколько разновидностей единиц компиляции:

- процедуры (подпрограммы **subroutine** и функции **function**), которые могут вызываться главной программой **program** и друг другом;
- подпрограмма **block data**, которая вообще никогда не вызывается при выполнении программы, а отрабатывает на этапе компиляции;
- модули **module** (начиная с ФОРТРАНа-90), которые так же не вызываются, но зато предоставляют возможность экспорта в программную единицу, использующую их, необходимых глобальных имен типов, переменных, процедур.

1.5.10 Один пример использования module в ФОРТРАНе

Удобно, когда изменения исходного текста программы при переходе на ячейки иной разрядности минимальны. ФОРТРАН-компиляторы позволяют осуществить переход посредством указания соответствующей опции. Однако её имя (название) у компиляторов разных фирм может быть неодинаково. Поэтому полезна ФОРТРАН-возможность разрешения ситуации.

1. В современном ФОРТРАНе оператор описания переменной вещественного типа допускает формы **real(4)**, **real(8)** и **real(mp)**, где **mp** — имя предварительно описанной именованной константы. Так что нашу первую программу можно оформить так

```
program test_mp
implicit none
integer, parameter :: mp=4  ! возможно 8 и 16
real(mp) x, y, z, t
x=0.1_mp
y=1+x
z=y-1
t=z/x
write(*,*) x, y, z, t
end
```

   и для перехода на переменные иной допустимой разрядности достаточно заменить значение константы mp.

   Однако, как правило, исходный текст программы состоит не из одной единицы компиляции, а нескольких (или даже многих). Поэтому выгодно, чтобы каждой была доступна именованная константой **mp**, которая описана отдельно. Такую возможность предоставляет единица компиляции вида **module**. Например,

```
module my_prec              ! файл my_prec.f
implicit none               !
integer, parameter :: mp=4 !
end module my_prec

program test_mp              ! файл main.f
use my_prec
implicit none
real(mp) x, y, z, t
x=0.1_mp
y=1+x
z=y-1
t=z/x
write(*,*) x, y, z, t
end
```

Теперь получить исполнимый код можно командой

```
gfortran my_prec.f main.f -o main
```

Подключение модуля **my_prec** обеспечило доступ только к константе **mp**, но не к оператору **implicit none**. Действие последнего распространяется только в пределах модуля **my_prec**. Поэтому **implicit none** необходимо указать и в главной программе. Если же этого не сделать, то за последствия отвечаем мы. Если при его отсутствии в оператор **real(mp)** включены и **x**, и **y**, и **z**, и **t**, то при **mp=10** получим ожидаемый результат

```
x=  1.00000000000000000000E-0001
y=   1.100000000000000000
z=  1.00000000000000000002E-0001
t=  1.000000000000000000002
```

Однако, если в случае отсутствия **implicit none**, в описании переменных не указать **x**, то получим:

```
x=  0.1000000
y=   1.100000001490116119
z=  0.100000014901161194
t=  1.000000000000000000000
```

2. В старом ФОРТРАНе тоже есть средство передачи значений переменных и именованных констант в другие единицы компиляции, минуя список аргументов последних. Это — так называемые **common**-блоки; их коснёмся в третьем семестре). Однако они не являются единицами компиляции, как **program**, **subroutine**, **function** или **module**, и не могут быть откомпилированы сами по себе. Подключение **common**-блоков к той или иной программной единице осуществимо либо явной записью его текста, либо посредством использования инструкции **include**, указывающей имя файла с **common**-блоком. Последнее, конечно, более практично по сравнению с явной записью, но далеко не так безопасно и удобно как подключение **module**.

3. Выгода от возможности компиляции **module** — получение соответствующего объектного файла. В отличие от ФОРТРАН-оператора **use** инструкция **include** лишь импортирует подключаемый текст в намеченную область программной единицы, Эту единицу придётся компилировать как единое целое, в то время как при использовании **module** именно его перекомпиляции можно избежать, что в случае больших проектов существенно экономит время.

4. **Рекомендация:**

   При необходимости используем единицы компиляции типа module, а не common-блоки.

   О common-блоках вспоминаем, лишь копаясь в древних ФОРТРАН-программах.

1.5.11 О чем узнали из пятого параграфа (кратко)?

1. **Фиксированный** и **свободный форматы** записи ФОРТРАН-программ.
2. **Признаки комментария** в ФОРТРАН- и СИ- программах.
3. **Правило умолчания** ФОРТРАНа. **Всегда отключаем !!!**
4. **implicit** – оператор неявного описания.
5. Операторы **real**, **real(4)**, **real(8)**, или **real*4**, или **real*8** и **real(mp)**.
6. Оператор **присваивания**.
7. Операторы вывода ФОРТРАНа **print** и **write**.
8. **Переменная** – имя ячейки оперативной памяти, содержимое которой программа по инструкции программиста может изменить.
9. **Константа** – имя ячейки оперативной памяти, содержимое которой программа даже по инструкции программиста на этапе исполнения изменить не может – любая попытка изменения трактуется компилятором как ошибка.
10. **ЭВМ иногда не способна правильно выполнить расчет даже по очень простой явной формуле.**
11. **format** – оператор с правилами записи данных в строку или чтения из нее.
12. **f**, **F**, **e**, **E**, **d**, **D** – дескрипторы данных вещественных типов (задают правила преобразования данных либо из внутреннего машинного представления в требуемое десятичное, либо из заданного десятичного во внутреннее машинное).
13. Первое число после дескрипторов **f**, **F**, **e**, **E**, **d**, **D** – количество позиций, отводимое под запись данного на внешнем носителе, а второе число, записанное через точку после первого – количество цифр данного после десятичной точки.
14. **i** – дескриптор данных целого типа; число после дескриптора **i** – количество позиций для записи данного целого типа.
15. **x** – дескриптор пробельного символа; число перед дескриптором **x** означает количество пробелов, включаемых в форматную строку. **t** – дескриптор позиции строки; число после дескриптора **t** означает номер позиции строки, после которой следует начать выбор данного (или запись его) при работе.
16. Дескрипторы **x** и **t** выгодно дополняют друг друга: **x** – выгоден для указания расположения данного относительно соседнего, а **t** – для аналогичного указания относительно первой позиции строки.
17. **программная единица** – единица компиляции. В ФОРТРАНе есть пять видов единиц компиляции: **program**, **subroutine**, **function**, **block data**, **module**.
18. **end** - оператор завершения любой программной единицы ФОРТРАНа.

1.6 Простая линейная программа на СИ и С++.

1. Текст программы на СИ.
2. Препроцессор СИ.
3. Главная программная единица СИ-программ.
4. Уяснение значения термина "функция" в СИ и ФОРТРАне.
5. Операторные скобки языка СИ.
6. Объявление числовых переменных и констант в СИ и ФОРТРАНе.
7. Элементарный форматный вывод в СИ.
8. Текст программы на СИ++.
9. О двух стилях записи программ на СИ++.
10. Почти ничего о перегрузке операций.
11. Элементарный ввод числовых данных в СИ и ФОРТРАНе.
12. О чем узнали из шестого параграфа (кратко)

1.6.1 Текст программы на СИ.

Рассмотрим запись алгоритма из предыдущего пункта на языке СИ:

```
#include <stdio.h>  // директива препроцессора              ! Файл main.c

int main()          // все, что после //
{                   //  и до конца строки - комментарий
 float x, y, z, t;
 x=1e-8;  y=1+x;  z=y-1; t=z/x;
 printf("%e %e %e %e\n", x, y, z, t); return 0;
}
```

1.6.2 Препроцессор СИ.

Препроцессор (текстовый процессор) упрощает запись СИ-программ. Для управления его работой используются **директивы** (указания, инструкции). Препроцессор доводит текст исходной программы до вида, который можно подать на вход компилятору. **Почему нельзя обойтись без препроцессора?**

В СИ нет встроенных операторов ввода/вывода (в ФОРТРАНе и С++ есть). СИ-программы реализуют ввод/вывод через вызов соответствующих функций (например, **printf** для вывода, **scanf** для ввода). Их имена, типы, атрибуты параметров определяются в специальном *"заголовочном"* файле **stdio.h**. Первое слово в имени **stdio.h** – сокращение от **standard input/output**, **h** – первая буква слова **header**. Без подключения **stdio.h** к исходной программе компиляция не пройдет. Директива **#include <stdio.h>** и выполняет подключение. Значок диеза – признак директивы препроцессора. Имеется две формы указания маршрута поиска файла **stdio.h**: угловые скобки (**<stdio.h>**)и двойные кавычки (**"stdio.h"**). При неполном имени файла в угловых скобках поиск ведется только в стандартных каталогах, маршруты к которым известны компилятору. Если же неполное имя дано в двойных кавычках, то в первую очередь ищется файл с указанным именем в текущем каталоге, и только в случае неудачи поиск продолжается по маршруту угловых скобок.

1.6.3 Главная программная единица СИ-программ.

Вторая строка: СИ–программа всегда содержит **функцию** с зарезервированным именем **main**. Именно оно в языках СИ и С++ определяет заголовок главной программы. Наличие круглых скобок после **main** указывает, что **main** – имя функции. В СИ пустота в круглых скобках означает, что о параметрах функции сказать ничего нельзя; в СИ++: что параметров у функции нет. Аналог последнего в СИ – обязательное наличие в круглых скобках служебного слова **void**, которое с той же целью может использоваться и в СИ++. Так что объявления **int main()** и **int main(void)** в СИ++ эквивалентны.

Описатель типа **int** перед словом **main** означает, что главная программа должна вернуть целое значение вызвавшей ее программе (**оболочке** или **скрипту**). Это значение не обязано быть результатом, который печатает в **main** программист. Оно нужно для организации реакции **скрипта** (сценария) или **оболочки** на причину завершения работы **main**. Безаварийному завершению обычно сопоставляют число нуль (посредством оператора **return 0;**), в противном случае – число отличное от нуля. Посмотреть его можно командой **echo $?**, где **$?** – имя переменной окружения, в которую поступает код завершения текущей задачи. Например, запуск команд **cal 12 2006; echo $?** приведут к высветке после календаря числа **0**, а команд **cal 13 2006; echo $?** – числа **1** (после сообщения о неверном номере месяца).

1.6.4 Уяснение термина “функция” в СИ и ФОРТРАНе.

СИ-**функция** – это именованная программная единица, которой можно передавать параметры и которая может через свое имя возвращать некоторое выработанное значение или не возвращать. Так, функция **sin(x)** возвращает через имя **sin** значение синуса, функция **printf** не только печатает требуемое программистом, но и возвращает через имя **printf** количество символов, из которых состоит печатаемое. В СИ можно объявить функцию с описателем типа **void**, которая через свое имя ничего не возвратит вызывающей среде. В ФОРТРАНе такую функцию называют **подпрограмма** (оператор описания – **subroutine**), а функцию, возвращающую через свое имя некий результат – **функцией** (оператор описания – **function**).

Если результатов несколько, то **функция** может один из них возвратить через свое имя, а остальные через свои параметры. Так **СИ-функция** форматированного ввода **scanf** (в нашей программе не вызывается) через свое имя передает вызвавшей среде значение типа **int** – количество введенных параметров, а значения самих параметров возвращает через список ввода (см. далее пункт **1.6.9**).

1.6.5 Операторные скобки СИ.

Третья строка: Фигурные скобки в СИ всегда обрамляют тело любой функции (в частности, и главной программы). Они – аналог служебных слов **begin** и **end** языка ПАСКАЛЬ. При желании посредством директив препроцессора #**define begin {** и #**define end }** можно было бы указанные слова назначить синонимами фигурных скобок. #**define**, как и #**include**, еще одна из директив препроцессора.

1.6.6 Объявление переменных и констант числовых типов

Четвертая строка: содержит объявление имен переменных, которые используем в тексте программы. Термин "переменная", как и в ФОРТРАНе, означает, что посредством **выполняемых** операторов программы можем изменять содержимое ячеек памяти, названных **x, y, z, t**. Если же нужно запретить изменение содержимого какой–то ячейки на все время выполнения программы (для подстраховки от необдуманного его изменения выполняемыми операторами), то соответствующее этой ячейке имя следует объявить не переменной, а константой. Например, так

```
#include <stdio.h>                        // аналогичная ФОРТРАН-программа:
int main()                                //      program testconst
{ double y=10.3;                          //      real*8 x, y /10.3d0/
  const double x=10.3;                    //      parameter (x=10.3d0)
  printf("  x=%25.17le  y=%25.17le\n",    //      write(*,'(''x='',d25.17,
            x,y);                         //     > ''y='',d25.17)') x, y
  return 0;                               //      stop 0
}                                         //      end
```

В этих примерах **x** – имя константы, а **y** – имя переменной. Слова **double** и **real*8** означают, что имена **x** и **y** нацелены на работу с восьмибайтовыми значениями вещественного типа. Константе **10.3** здесь сопоставлено имя **x**. Именовать константу выгодно, если она часто встречается в тексте программы. **Всегда**

1) явно объявляем тип имени именованных констант;

2) ФОРТРАН-константы типа REAL*8 (в случае древних программ) задаем с порядком в форме D, но, начиная с ФОРТРАНа-95, разрядность константы типа real выгоднее указывать посредством _mp, где mp — константа целого типа, описанная в модуле, который подсоединяется к нужной единице компиляции оператором use.

Термин **"объявление"** означает выделение места в памяти под хранение значения указанного типа. Завершается объявление списка переменных символом **;** (точка с запятой). Этот символ в СИ и СИ++ синтаксически завершает не только оператор объявления переменных, но и вообще любой оператор, за исключением

1) директивы препроцессора; 2) заголовка функции;

3) закрывающей операторной скобки **}**.

1.6.7 Элементарный форматный вывод в СИ.

Форматированный вывод на экран в СИ обеспечивает функция **printf**. Первый параметр этой функции – строка, управляющая выводом (аналог содержимого оператора **format** ФОРТРАНа. Она заключается в двойные кавычки. Её содержимое: спецификаторы формата, указывающие, как следует печатать элементы списка вывода, если таковой имеется, а также поясняющий текст, если последний нужен. Элементы списка вывода разделяются запятыми.

Язык спецификации форматов в СИ отличен от языка оператора **format** ФОРТРАНа, хотя и похож немного. Так, команда правила преобразования данного в СИ

должна всегда начинаться со знака % (в ФОРТРАНе этого нет). Перевод вещественного значения в форму с фиксированной запятой без порядка задается литерой **f**, в форму с порядком – литерой **e**, перевод целого – литерой **i**. Как видно, литеры, задающие правила преобразования, в этих случаях у ФОРТРАНа и СИ одинаковы. Количество позиций для записи числового данного и количество цифр, приходящееся на дробную часть числа и в ФОРТРАНе, и в СИ кодируется одинаково, именно:

1) сначала пишется количество позиций отводимое на все данное;

2) затем ставится точка;

3) после нее пишется количество цифр нужное в дробной части.

Однако в ФОРТРАНе это сочетание пишется после буквы дескриптора, а в СИ между значком % и дескриптором.

И в ФОРТРАНе, и в СИ больше дюжины правил преобразования данных, а, значит, и задающих их литер (дескрипторов преобразования). Обозначения соответствующих друг другу ФОРТРАН- и СИ-дескрипторов для символьных и строковых данных разные (в СИ используются дескрипторы: **%c** – для вывода одного символа, **%s** – для вывода строки, а в ФОРТРАНе оба случая обслуживает дескриптор **a**).

```
#include <stdio.h>  // Пример элементарного использования функции printf.
int main(void)      // Обратите внимание: Верен ли печатаемый результат?
{const c0=100.34;   // ================== Как объяснить печатаемое программой?
 float  x=1.2345678901234567890;
 double y=1.2345678901234567890, z;
 printf(" c0=%d     c0=%e",c0,c0);
 z=c0*1000;
 printf("  z=%23.17le\n",z);
 printf("x=%f   x=%e   x=%E\n",x,x,x);
 printf("y=%f",y);
 printf("   y=%e",y);
 printf("   y=%E\n",y);
 printf("--\n");
 printf("y=%25.17f\n",y);
 printf("y=%25.17e\n",y);
 printf("y=%25.17E\n",y);
 printf("---------------------------\n\a");
 printf("x=%25.17f\n",x);  printf("x=%25.17e\n",x);  printf("x=%25.17E\n",x);
 return 0;
 }
```

В СИ есть специальные символьные константы, которые *понимаются* компилятором особым образом. Все они начинаются с символа **обратный слеш**, за которым сразу (без пробела) располагается нужная литера. В частности,

```
1.  \n -- при выводе  на принтер и на STDOUT служит признаком перевода
          строки. В ASCII-таблице эта константа имеет обозначение  LF
          (от Line Feed -- перевод строки; дословно Feed -- питание;
          подача материала; снабжение сырьем, в смысле -- "строкой";
          шестнадцатеричный код 0A).
2.  \a -- при выводе на STDOUT вызывает звуковой сигнал. В ASCII-таблице
          ее обозначение BEL (bell -- звонок, зуммер; шестнадцатеричный
          код 07).
```

Результат работы этой программы:

```
 c0=100     c0=4.852872e-270  z=1.00000000000000000e+05
x=1.234568   x=1.234568e+00   x=1.234568E+00
y=1.234568   y=1.234568e+00   y=1.234568E+00
--
y=      1.23456789012345669
y=  1.2345678901234566 9e+00
y=  1.2345678901234566 9E+00
---------------------------
x=      1.23456788063049316
x=  1.2345678806304931 6e+00
x=  1.2345678806304931 6E+00
```

1.6.8 Текст программы на C++.

```
#include <stream.h>
main()
{
 float x, y, z, t;  // Вообще, в задачах расчетного характера безопаснее
 x=10;              // использовать тип double. Здесь же использован тип
 y=1+x;             // float с тем, чтобы при замене x=1e-7 увидеть
 z=y-1;             // эффект потери точности z на типе float.
 t=z/x;             // Для double этот же эффект проявится при x=1e-16.
 cout << x <<" "<< y <<" "<< z <<" "<< t << "\n"; // в СИ++ "\n" и endl
}                                                 //     эквивалентны.
```

Первая строка: содержит директиву препроцессора. Имя в операторных скобках отлично от соответствующего имени СИ–программы. В C++ можно использовать **stdio.h**, но тогда, как и в СИ–программе, придётся вызывать функцию **printf**. Однако в C++ есть и другое средство вывода – оператор вывода **cout**. Имя **cout** – это имя стандартного потока вывода (действующей программной модели устройства вывода). Эта модель сконструирована в СИ++ через наборы иерархий классов и описание ее помещено в файл **stream.h**. Именно через классы C++ позволяет реализовывать идеи объектно–ориентированного программирования.

1.6.9 О двух стилях записи программ на C++.

В языке STANDARD C++ имя, которое пишется в угловых скобках отлично от **stream.h**, например, **iostream**, или **fstream**, и т.д. Последние – не имена файлов (указывать расширение **h** не надо), а стандартные идентификаторы по которым компилятор находит файл. При подключении заголовка стиля STANDARD C++ содержимое заголовка оказывается в **пространстве имен std**. Для подсоединения последнего к области видимости программы используется оператор

using namespace std;

В [16] рекомендуется перед работой установить: поддерживает или нет компилятор СИ++, который намереваемся использовать, запись исходного текста в стиле STANDARD C++ путём пропуска двух альтернативных вариантов программы:

```
   STANDARD C++                  BORLAND C++

#include <iostream>                //    #include <iostream.h>
using namespace std;               //
int main()                         //    int main()
{                                  //    {
 cout << "ЗАРАБОТАЛО!"; return 0;  //     cout << "ЗАРАБОТАЛО!"; return 0;
}                                  //    }
```

1.6.10 Почти ничего о перегрузке операций на C++.

Консольный ввод и вывод в C++ реализуются операторами

cin >> имя_переменной; // ввод с экрана
cout << имя_переменной; // вывод на экран

Символы << в СИ означают операцию сдвига влево, а символы >> – операцию сдвига вправо. В C++ это их назначение сохраняется, но в контексте операторов ввода-вывода происходит изменение их смысловой нагрузки.

```
#include <iostream>    // текст программы в Standat C++.
using namespace std;   // подключение пространства имен std.
main()
{
 float x, y, z, t;
 x=1e-8;   y=1+x;   z=y-1;   t=z/x;
 cout << x <<" "<< y <<" "<< z <<" "<< t << endl;
}
```

Это изменение — пример **перегрузки операций**, т.е. обозначение одним и тем же значком разных алгоритмов, что позволяет не придумывать для каждого из них нового имени, осуществляя выбор по типу обрабатываемого данного. Например, значок + удобен для обозначения операции сложения и целых, и вещественных, и комплексных, и векторов, и матриц, хотя алгоритмы, естественно, разные. Если язык программирования предоставляет такую возможность, то сильно упрощается запись программы.

1.6.11 Элементарный ввод числовых данных в СИ и ФОРТРАНе.

Следующая **СИ-программа** демонстрирует элементарное использование функции форматного ввода **scanf**, о которой уже упоминалось в пункте **1.6.4**.

```
#include <stdio.h>
int main()
{  float x, y, z;   int n, m;
   m=printf(" введи вещественные x, y, z\n");
   n=scanf("%e  %e  %e",&x, &y, &z);
   printf(" n=%i     x=%e     y=%e    z=%e\n", n,x, y, z);
   printf(" m= %d\n",m);    return 0;
}
```

Программа выведет на экран:

```
 введи вещественные x, y, z    <- Это вывела программа после запуска.
2.3  4.5 6.7                   <- Это результат набора чисел на экране
 n=3     x=2.300000e+00     y=4.500000e+00    z=6.700000e+00 <- Это вывод
 m= 28                                                      <- программы.
```

1. Число, записанное при первом обращении к функции **printf** в переменную **m**, равно **28** (символ перевода строки – один, хоть и обозначается двумя значками).

2. Список ввода функции **scanf** отличается от соответствующего списка вывода функции **printf** наличием перед именами переменных знака **&** (амперсанд), обозначающего в СИ операцию взятия адреса переменной.

   Имена переменных **x, y, z** в контексте функции **printf** отождествляются со значениями переменных, хотя у любой переменной имеется ещё и адрес.

   printf нацелена на вывод значений согласно указанному формату.

   Перевод на русский язык действия функции **scanf**:

   "Сканирование строки, введенной с экрана, и запись из неё числовых значений, переведенных во внутреннее машинное представление согласно спецификациям формата, **по адресам** *переменных* **x**, **y** *и* **z** *соответственно.*

   Сканирование строки начинается только после нажатия клавиши **enter**, когда завершен прием символов во внутренний буфер (до ее нажатия текст, предназначенный для ввода можно исправлять).

Ввод данных на ФОРТРАНе записывается без какого-либо намёка на адреса:

```
program testread
implicit none
real x, y, z;
write(*,*) ' введи вещественные x, y, z'
read (*,*) x, y, z
write(*,*) ' x=',x,'   y=',y,'   z=',z
stop 5
end
```

Правда, здесь не находится количество вводимых аргументов, не подсчитывается число букв в выводимой фразе и не используется форматный ввод. Однако, сейчас просто знакомимся с организацией простейшего ввода чисел на ФОРТРАНе. Результат пропуска данной программы после компиляции на **gfortran**:

```
  введи вещественные x, y, z
2.3  4.5 6.7
  x=   2.300000         y=   4.500000         z=   6.700000
STOP 5
```

ФОРТРАН-оператор **stop 5** – аналог СИ-оператора **return 5**: возвращает оболочке код завершения работы программы. При необходимости он доступен в скрипте, например, посредством команды **echo $?** (см. пункт **1.6.3**).

Обращение к оператору ввода не отличается от вызова оператора вывода за исключением замены имени **write** на имя **read**. Смысловая нагрузка остальных синтаксических компонент с точностью до направления потока ввода/вывода неизменна:

Первый параметр в скобках после слова READ – условное имя (программный псевдоним) устройства ввода: * – ввод из командной строки экрана.

Второй параметр в скобках указывает правила перевода набранных данных во внутреннее машинное представление. Вторая * означает, что программа сама определит нужные правила, исходя из количества и типа переменных, принимающих данные, независимо от места расположения последних в командной строке. Конечно, при желании вместо * можно явно указать и формат ввода.

Список ввода оператора **read** выглядит в ФОРТРАНе точно так же, как и список вывода оператора **write**: через запятую имена переменных, которые принимают вводимые данные (никаких амперсандов).

Вообще и в СИ, и в ФОРТРАНе вводимые данные, если их не слишком много, всегда полезно сразу после ввода вывести на печать с тем, чтобы убедиться в правильности их восприятия программой. Часто спрашивают:

Зачем печатать вводимые данные, если они уже видны при наборе?

1. Анализ результатов расчета в первую очередь требует знания исходных параметров. Если их нет в файле с результатом, то анализ просто невозможен.

2. В тексте программы возможна досадная опечатка, из-за которой очередное значение запишется не в ту переменную. Пусть вместо **read(*,*) a, b** случайно набрали **read(*,*) a, a**. Без **write(*,*) ' a=',a,' b=',b** на экране виден правильный с точки зрения вводящего набор чисел, хотя при работе программы второе число, которое должно попасть в переменную **b**, окажется в переменной **a** , затерев значение последней, введенное ранее. Таким образом, наличие контрольной печати может значительно сэкономить время поиска подобных опечаток в программе.

3. Наконец, при вводе можно не заметить ошибку в наборе данного. Например, вместо **0.15+5** в спешке ввели **0.15** (не указали порядок числа). В таком случае наличие контрольной печати опять же экономит время, которое без нее будет истрачено, возможно, на поиск в программе несуществующей ошибки.

Для ввода данных на языке C++, проще всего использовать поток **cin**, о котором упоминалось в параграфе **1.6.10**:

```
#include <iostream>    // текст программы в Standat C++.
using namespace std;   // подключение пространства имен std.
int main()
{
 float x, y, z, t;
 cout <<"Введи вещественные x, y, z:"<<endl;
 cin >>x>>y>>z;
 cout <<" x="<<x <<" y="<< y <<" z="<< z << endl; return 0;
}
```

Чтение данных Фортран-программой из файла. Рассмотрим программу которая требует ввода нескольких данных типа **real** и одного данного типа **integer** под управлением оператора **format**.

```
      program tstread
      implicit none
      real*8 a, b, p, q, eps
      integer  n
      read  (*,100)  a,  b, eps, p, q
      read  (*,101)  n
      write (*,1000) a,  b, eps, p, q, n
  100 format(d10.3)
  101 format(i10)
 1000 format(1x,'  a=',d25.17/
     >          '   b=',d25.17/
     >          ' eps=',d25.17/
     >          '   p=',d25.17/'   q=',d25.17/'   n=',i5)
      end
```

Работа с ней неприятна из-за необходимости следить за размещением каждого данного в пределах первых десяти колонок, не говоря уже о повторе набора всех данных в случае ввода опечатки. Данные выгодно поместить в файл (например, с именем **tstrdr.inp**) на диске и при запуске программы использовать операцию перенаправления стандартного ввода, то есть **$ ./a.out < tstrdr.inp**. В этом случае текст программы не меняется, а ошибки или опечатки набора данных в файле гарантированно исправляются его редактированием.

Выгодно ли управление чтением данных из файла по списку ввода

read(*,*) a,b,eps,p,q,n

Вроде бы все данные можно разместить в одной-двух строках, экономя место и не закрепляя числа за определёнными позициями строки. В этом случае, однако, пользователю при подготовке файла с данными нужно помнить о порядке ввода. Если к программе обратиться через полгода, то уже придется вспоминать и о смысловой (проблемной) нагрузке имён, используемых в программе.

Оператор **format** – ценен тем, что позволяет, вводя данное из нужных позиций строки, вообще игнорировать при чтении содержимое оставшейся ее части, в которой можно поместить достаточно емкий и ценный комментарий. Например,

```
  5.300+0<--=...a....(нижний предел )
  6.900+0<--=...b....(верхний предел)
 1.000-13<--=..eps...(относительная погрешность)
    2.100<--=...p....(параметр при аргументе)
    4.   <--=...q....(свободный коэффициент)
       32<--=...n....(число промежутков дробления >= 1)
```

Операция перенаправления стандартного вывода позволяет и результат работы программы перенаправить в файл текущей директории.

```
$./a.out < tstrdr.inp > tstrdr.res
```

1.6.12 О чем узнали из шестого параграфа (кратко).

1. **Препроцессор** – средство приведения СИ-программ к виду пригодному для подачи на вход к компилятору.
2. #**include** – директива препроцессора (подключение нужного файла)
3. Функциональное различие между угловыми скобками и двойными кавычками, обрамляющими имя файла, подключаемого посредством директивы #**include**.
4. #**define** – директива препроцессора (определение новых обозначений).
5. **Фигурные скобки** – операторные скобки языка СИ.
6. **main** – имя функции, главной программной единицы СИ.
7. По умолчанию функция СИ возвращает через свое имя значение типа **int**.
8. В C++ тип функции необходимо указывать всегда.
9. Функция **printf** – средство форматирования выводимых данных в СИ.
10. **const** – оператор описания константы в СИ. объявлении
11. **parameter** – оператор описания константы в ФОРТРАНе.
12. Почему иногда выгодно помимо **переменных** использовать и **константы**?
13. При объявлении именованной константы **всегда** указываем ее тип.
14. **float** – СИ-оператор описания типа переменных, значения которых предполагается хранить с одинарной точностью (ФОРТРАН-аналог – **real**).
15. **double** – СИ-оператор описания типа переменных, значения которых предполагается хранить с удвоенной точностью (ФОРТРАН-аналог — **real(8)**).
16. **Два стиля записи программ на СИ++.**
17. **Перегрузка операций** – обозначение одним именем разных алгоритмов.
18. **scanf** – СИ-функция форматированного ввода данных с экрана.
19. Переменные в списке ввода функции **scanf** задаются своими адресами.
20. **&** – операция определения адреса переменной в СИ.
21. **&** – признак продолжения ФОРТРАН-оператора в следующей строке при свободном формате записи программы.
22. **read** – оператор ввода в ФОРТРАНе.
23. **cin** – поток ввода в C++ ($\gg$).
24. **cout** – поток вывода в C++ ($\ll$).
25. Полезна **контрольная** печать вводимых данных сразу после ввода.

1.7 В первый раз – в дисплейный класс.

1. Формальная часть.
2. Элементарная работа с каталогами в UNIX.
3. Элементарная работа с редактором **mcedit** *в среде* **mc**.
4. Отладка простейшей программы на ФОРТРАНе.
5. Отладка первой программы на ФОРТРАНе.
6. Отладка первой программы на СИ и C++.
7. Тестирование программ.
8. Исходный, объектный и загрузочный файлы.
9. Сводная таблица команд вызова трансляторов.
10. О чем узнали из седьмого параграфа (кратко)?.
11. Домашнее задание N 1.

Перед уходом из ДК не забыть грамотно выйти из своей ЭВМ !!!

1. Установить указатель мыши на иконку **Open Suse** и (после нажатия левой клавиши мыши) выбрать **Выйти из сеанса** (зелёный человечек) и опять нажать левую клавишу мыши.

2. В окне, появившемся на экране, снова выбрать **Выйти из сеанса**. Снова нажать левую клавишу мыши и дождаться появления на экране **login**-окошка.

1.7.1 Формальная часть.

Пока наши **login**-имена не задействованы все пользуемся домашней директорией **student**. Это, вообще говоря, не очень удобно, так как любой может зайти. Поэтому, заканчивая работу, лучше все свои наработки сбрасывать на флешку.

Как только **login**-имя будет задействовано, то в поле ввода **login**-имени пишем его, а в поле ввода **пароля** пишем пароль. Если всё набрано верно, то оказываемся в своей **домашней** директории. Для повторения услышанного на лекциях создаём (посредством команды **cd**) каталоги **for**, **f95**, **myc** и **mycpp**:

```
$> cd for    ! для ФОРТРАН-программ в фиксировнном формате
$> cd f95    ! для ФОРТРАН-программ в свободном формате
$> cd myc    ! для СИ-программ
$> cd mycpp  ! для программ на C++
```

Вы будете получать домашние задания. В каждом задании несколько задач. Каждую обязательно делать на двух языках ФОРТРАНе и СИ. Выбор ФОРТРАН-формата (фиксированный или свободный) остаётся за Вами. Условимся об организации размещения решений на внешней памяти (имеется ввиду диск, а не «флешка»; хотя все наработки полезно копировать и на «флешку»). Для первого домашнего задания на ФОРТРАНе создаём, например, внутри каталога **f95** каталог **home1**. Для первой задачи первого домашнего задания создаём внутри каталога ∼/**f95**/**home1** каталог **task1** и только внутри **task1** создаём файлы, нацеленные на решение задачи.

Возможна и иная иерархия размещения решений. Создать в домашней директории набор каталогов **home1**, **home2**, и т.д. В каждом из них создать каталоги **for**,

f95, **myc** и **mycpp** и уже в каждом из них создавать каталоги **task1**, **task2** и т.д. Выбор за Вами.

1. Нажимаем мышиный указатель левой клавиши мыши на иконке **Open Suse**.
2. Выбираем в возникающем меню строку **Система** и далее в новом меню **Konsole** и получаем на экране окошко для набора **Linux**-команд.
3. Тренируемся, пробуя команды из таблички **1.7.2** пункта

Элементарная работа с каталогами UNIX.

4. Переходим к пункту **1.7.3**

Элементарная работа с редактором mcedit в среде mc,

осваивая навыки по работе с **mc** (Midnight Commander).

Итак, делаем текущей директорию **for** и вызываем Midnight Commander:

```
cd for
mc    // вызов   (Midnight Commander) далее всё по пункту 1.7.3
```

После того как освоили вход и выход, осваиваем вызов окна терминала и переходим к выполнению пунктов **1.7.k, k=2(1)11**. Приобретаем навыки работы со знакомыми **unix**-командами по созданию и уничтожению каталогов. Создаем в домашней директории иерархию каталогов с соответствующими исходными текстами решения первой задачи на разных языках программирования:

/first/myf95/first.f95 и **/first/myfff/first.f**

/first/myc/first.c, **/first/mycpp/first.cpp**.

Набор исходных текстов осуществляем в среде **mc**, используя встроенный редактор **mcedit**. Нумерация колонок в **mcedit** начинается с нуля, что учитываем при наборе ФОРТРАН-текста в фиксированном формате (колонка продолжения в среде **mcedit** – пятая, а не шестая).

Уяснение ситуации. Для тех, кто привык работать с другими редакторами (**vim**, **gedit**, **emacs**), не возбраняется их использовать.

Каждую из программ отлаживаем, тестируем, осмысливаем результаты, сравниваем их между собой. Письменно формулируем обнаруживаемые несуразности, если таковые находятся. Задачи домашнего задания делаем письменно в отдельной тетради, записывая текст каждой из программ в двух вариантах:

ФОРТРАН-95, СИ.

1.7.2 Элементарная работа с каталогами в UNIX.

Цель действия	Содержимое командной строки	Примечания
Что в текущем каталоге?	**ls**	
Каков путь к текущему?	**pwd**	
Какой мой **login** ?	**whoami**	Пока наш логин всё-таки **student**
Создать каталог	**mkdir first**	для первой задачи
Войдем в него	**cd first**	
Что в нем есть?	**ls**	Да, ничего!
Создать в **first** каталог	**mkdir for**	фикс. ФОРТРАН-формат
Создать в **first** каталог	**mkdir f95**	своб. ФОРТРАН-формат
Создать в **first** каталог	**mkdir myc**	на C
Создать в **first** каталог	**mkdir mycpp**	на C++
Что теперь в **first**?	**ls**	четыре директории **for**, **f95**, **myc** и **mycpp**
Выйти в домашний каталог	**cd**	работа с **cd**
Увидеть его оглавление	**ls**	там есть и **astro191**
Войдем в него	**cd astro191**	просто тренировка
Что в нем?	**ls**	Тот же список, что уже видели
Входим снова в личную директорию	**cd m14yyy**	подготовка
Входим в директорию **for**	**cd for**	к ФОРТРАН-программированию

1.7.3 Элементарная работа с редактором mcedit в среде mc.

1. На экране должно быть окно **Konsole**.

2. В командной строке пишем **mc** (вызываем Midnight Commander).

3. Убеждаемся, что он нацелен на директорию **for** (если нет, то переводя подсвечивающий курсор посредством клавиш управления ↑ и ↓, и нажимая клавишу **enter**, добиваемся нужного результата; того же эффекта можно добиться и мышью).

4. Сейчас подкаталог **for** пуст. Нажимаем клавиши **Shift+F4**: на экране появляется окно редактора. Тренируемся по набору ФОРТРАН-текста в фиксированном формате. В нём колонка **6** – **колонка продолжения**; колонки с первой по пятую включительно – поле меток; остальные (до 72-й включительно) – поле операторов программы.

5. Выводим курсор на **7**-ю позицию **1**-й строки, пишем **end** и нажимаем на **enter**.

6. **Все! Простейшая ФОРТРАН-программа написана.** Правда, у нее явно не указано имя (нет оператора заголовка **program**), она ничего не делает, да и сама, пока, находится не в файле текущей директории, а в оперативной памяти редактора. Все же это – полноценная программа (из одного единственного невыполняемого оператора **end**), которую можно откомпилировать, получить и **объектный**, и **загрузочный** файлы, и инициировать выполнение последнего.

7. Запишем текст нашей программы в файл текущей директории, получая так называемый **исходный файл**. Запись осуществим *с выходом из редактора*. Цель – освоить этот путь, поскорее добраться до этапа трансляции и технологии пропуска задачи. Ничего, что пока программа ничего не делает. Потом ее нарастим. Кстати, и отлаживать её, постепенно наращивая, гораздо проще нежели искать ошибки в *"тысячестроковом монстре"*.

8. Итак, два раза нажимаем клавишу **Esc**. Выводится запрос о подтверждении нашего желания выйти из редактора с предоставлением трех вариантов ответа (см. параграф **1.4.5**).

9. Выбираем ответ **Да** (то есть: хотим записать и выйти из редактора).

10. Выводится окно с предложением набрать в нем имя файла. Придумываем имя, например, **main.f**. Расширение **.for** (или **FOR**, или **f**, или **F**) означает, что запись ФОРТРАН-программы будет вестись в режиме фиксированного формата. Набираем в окне придуманное имя и нажимаем на **enter**.

11. Оказываемся в окне среды **mc**. Убеждаемся, что в его активной половине (у которой вверху подсвечено имя каталога) теперь появилось имя **main.f**.

12. Нажмем клавиши **Ctrl+o** – возврат из **mc** к терминалу с командной строкой.

1.7.4 Отладка простейшей программы на ФОРТРАНе.

1. Пишем в командной строке команду вызова компилятора **gfortran**, которой в качестве аргумента подаем имя файла **main.f** с **исходным** ФОРТРАН-текстом и, используя опцию **-o**, указываем после неё желательное нам имя **загрузочного файла** с кодом программы на машинном языке.

   ```
   $ gfortran main.f -o main
   ```

2. Компоновку загрузочного файла всегда выполняем при включённой опции **-o**, явно указывая его имя. Удобно, чтобы оно отличалось от от имени исходного файла **главной программы** просто отсутствием расширения, так как в этом случае команда вызова программы будет похожа на вызов любой другой **unix**-команды, напоминая мнемоникой своё назначение.

3. Если при наборе программы в операторе **end** не допущено ошибок, то трансляция завершится успешно и в текущей директории появится **загрузочный файл** с именем **main**. Убедимся в этом, дав в командной строке команду **ls** или, вернувшись в окно **mc**, посредством **Ctrl+o**.

4. Инициируем выполнение **main**, набрав в командной строке

   ```
   $ ./main
   ```

   и нажав клавишу **enter**. В этой команде первая точка слева – псевдоним текущего каталога. Косая черта отделяет его имя от имени **загрузочного файла**.

5. Включим в программу один оператор печати, чтобы она хоть что-нибудь печатала. Для этого вернемся в среду **mc** посредством **Ctrl+o**, поместим подсвечивающий курсор на строку с именем **исходного модуля** и нажмем клавишу **F4**. Теперь на экране – окно редактора с исходной программой. Поместим курсор на начальную позицию строки с **end** и нажмем на **enter**, получая пустую строку над старым текстом. Наберем в ней оператор печати, например,

   ```
   write(*,*) x, y, z, t
   ```

6. Сохраним измененный файл на диске, нажав клавишу **F2** и вернёмся в окно **Konsole** посредством **Ctrl+o**. Клавишей ↑ получим в командной строке команду **gfortran main.f -o main** и нажмём на **enter**.

7. При отсутствии ошибок посредством ↑ получим команду вызова загрузочного файла **./main** и, нажав на **enter** увидим на экране результат в виде четырех странных чисел. Причина: программа не присваивала никаких значений переменным **x, y, z, t**. Поэтому в формате типа **real** отпечатались те данные, которые остались в ячейках **x, y, z, t** от работы программ предыдущего пользователя или операционной системы.

8. При обнаружении ошибки **компилятор** напечатает строку в которой *“по его мнению”* содержится ошибка, поместив под ошибочным (*опять-таки “по его мнению”*) символом номер ошибки в строке.

1.7.5 Отладка первой программы на ФОРТРАНе

Для удобства работы ниже приведены тексты ФОРТРАН-программ из подразделов **1.5.1**, **1.5.3**. Используя эти тексты, закрепим технику набора, вызова компилятора, исправления ошибок и запуска программы.

```
x=10                              ! пока фиксированный формат.
y=1+x                             ! записи ФОРТРАН-программы
z=y-1                             ! (файл main.f из директории
t=z/x                             !                         for)
write(*,*) x, y, z, t
end
```

Дополним текст, набранный в предыдущем пункте недостающими операторами и повторим указанные в нём действия, исправляя по ходу дела ошибки, если таковые найдутся. После окончательного пропуска программы убедимся в правильности результата её работы.

Теперь эту же программу запишем в свободном формате, размещая некоторые из выполняемых операторов до седьмой колонки. Сначала, однако, сделаем текущим каталог **f95** — всегда полезно новую версию программы разрабатывать в отдельном каталоге. Для этого выйдем из каталога **for** в родительский **first**, дав команду **cd ..**, после чего войдём в каталог **f95** посредством команды **cd f95**. Далее нажав клавиши **Shift+F4** получим пустое окно встроенного редактора среды **mc** и наберём в нём исходный текст программы:

```
                                  ! В свободном формате ФОРТРАН-записи
x=10;  y=1+x;                     ! значок амперсанда (&), завершающий
z=y-1; t=z/x; write(*,*) x, y,&   ! строку, означает, что на следующей
                         z, t     ! строке помещается продолжение
  end                             ! оператора, начатого на предыдущей.
```

Помним, для свободного формата расширение имени должно быть либо **f90**, либо **f95**. Сохраним набранное в файле, нажав клавишу **F2** и указав имя **main.f95**.

Вызовем компилятор **gfortran**:

```
$ gfortran main.f95 -o main
```

Если нет ошибок, убедимся в появлении загрузочного файла **main**. Инициируем его выполнение. Сравниваем результаты обоих пропусков.

1.7.6 Отладка первой программы на языках СИ и С++.

Для удобства здесь повторно приведены ее **С-** и **С++-** тексты из пунктов **1.6.1** и **1.6.10**. Вся работа по их набору и редактированию аналогична работе, проделанной при наборе ФОРТРАН-программ (отличие лишь в расширениях имен исходных файлов: **.c** для СИ и **.c++** или **.cpp** для СИ++).

```
#include <stdio.h>  // директива препроцессора                ! Файл main.c

int main()          // все, что после //
{                   //  и до конца строки - комментарий
 float x, y, z, t;
 x=1e-8;  y=1+x;  z=y-1; t=z/x;
 printf("%e %e %e %e\n", x, y, z, t); return 0;
}

#include <iostream>    // текст программы в Standat C++.
using namespace std;   // подключение пространства имен std.
main()
{
 float x, y, z, t;
 x=1e-8;   y=1+x;   z=y-1;   t=z/x;
 cout << x <<" "<< y <<" "<< z <<" "<< t << endl;
}
```

Вызов компилятора СИ : **gcc main.c -lm -o main** .

Вызов компилятора СИ++ : **c++ main.cpp -lm -o main** или
g++ main.cpp -lm -o main .

Опция **-lm** нацелена на подключение математической библиотеки. При отсутствии ошибок трансляции в соответствующей текущей директории получим загрузочный файл **main**. Запуск его выполняем командой **./main**.

1.7.7 Тестирование программ.

Рассмотрим результаты пропуска рассмотренных программ для аргумента **x=10**:

x	**y**	**z**	**t**	**Компилятор**
10.00000	11.00000	10.00000	1.000000	**gfortran**
1.000000e+01	1.100000e+01	1.000000e+01	1.000000e+00	**gcc**
10	11	10	1	**c++**

Видно, что несмотря на незначительные различия в форме вывода все результаты верны в пределах семи значащих цифр мантиссы.

Изменим в программах значение аргумента на $\mathbf{x = 10^{-7}}$ и $\mathbf{x = 10^{-8}}$, последовательно транслируя и пропуская программы вновь. Результаты должны совпасть с данными таблицы, приведенной ниже.

x	**y**	**z**	**t**	**Компилятор**
0.100E-06	1.0000001	0.1192093E-06	1.1920929	**gfortran**
0.100E-07	1.0000000	0.0000000E+00	0.0000000	
1.000000e-07	1.000000e+00	1.192093e-07	1.192093e+00	**gcc**
1.000000e-08	1.000000e+00	0.000000e+00	0.000000e+00	
1e-07	1	1.19209e-07	1.19209	**c++**
1e-08	1	0	0	

Налицо явно неожиданный, на первый взгляд, результат: значение **t**, которое в точности должно быть равно единице, достаточно сильно от последней отличается. При $\mathbf{x = 10^{-7}}$ у значения **t** верна лишь одна старшая цифра мантиссы – все остальные **неверны**. При $\mathbf{x = 10^{-8}}$ величина **t** вообще равна нулю. Так что, несмотря на очень простые формулы и малое количество арифметических операций, при одних значениях аргумента одна и та же программа получает верный ответ, а при других – неверный. Наша цель – на этом простом примере постараться понять причины, по которым аналитически точные и математически верные формулы могут оказаться плохо приспоспособленными для ведения по ним расчетов на ЭВМ.

1.7.8 Исходный, объектный и исполнимый файлы.

Файл с исходным текстом любой **единицы компиляции** (см. пункт**1.5.9**) называют **исходным файлом**. Компилятор, в конечном итоге, осуществляет перевод совокупности всех **исходных файлов**, входящих в программный проект, на машинный язык, получая один **исполнимый** (или **загрузочный**) **файл**.

Компилятор предоставляет программисту возможность получить **исполнимый файл** двумя способами:

1) непосредственно в виде файла с исполнимым кодом. Например,

```
$ gfortran main.f -o main   # main - имя загрузочного файла
```

2) предварительно получив файлы-посредники в виде **объектных файлов**, из которых затем надо скомпоновать **загрузочный файл**. Например,

```
$ gfortran -c main.for     # опция -c (от слова compilation) - приказ
                           # компилятору создать только  объектный модуль
                           # имя которого отличается от имени исходного
                           # лишь расширением ( .for заменяется на .o)

$ gfortran main.o -o main  # компоновка загрузочного файла main
                           # из объектного файла main.o
```

В зависимости от сложности проекта программы возможно объективное обоснование применения каждого из способов.

Объектный файл - это результат перевода исходного текста единицы компиляции в машиннозависимый код в командах процессора с условной адресацией. Для получения **объектного файла** требуется больше времени, чем на превращение условной адресации в абсолютную. Поэтому, если в программный проект входит несколько файлов, то выгодно каждую **единицу компиляции** помещать в отдельный **исходный файл** с целью получения из него соответствующего **объектного**, так как при модификации одного исходного файла придётся перекомпилировать только его.

Вообще говоря, сборку **загрузочного файла** из **объектных модулей** обеспечивает специальная системная программа, называемая **редактором связей** (или **компоновщиком**, или **линкёром**). Вызов её обычно можно инициировать и вызовом компилятора, если последнему в качестве аргумента подать объектный файл, т.е. соответствующий файл с расширением **.o** (см. пример из пункта 2).

На первый взляд кажется, что при большом количестве файлов, входящих в проект, можно случайно перекомпилировать не тот исходный файл, который хотели, и что реальное время, затраченное на уяснение этого факта, гораздо больше времени перекомпиляции всех исходных файлов. Подобная аргументация основана **на незнании** о существовании утилиты-координатора **make**, использование которой исключает такую ситуацию.

Конечно, при пропуске простых программ, в которых нет обращения к подпрограммам пользователя, выделение этапа получения объектного модуля кажется излишним. Однако, при выполнении курсовых и дипломных работ, а также в курсе **Операционные системы** потребуется умение грамотной работы с **объектными файлами**. Поэтому навык их независимой генерации и использования, как и составления элементарных **make**-файлов, полезно получить на простых программах.

Замечания:

1. Подчеркнём ещё раз: **выгодно, чтобы проект какой-то одной конкретной задачи находился в директории, предназначенной именно для него**. В частности, выгодно, чтобы имя директории мнемонически напоминало о теме проекта. Выгода — в возможности создания для всего проекта в целом наглядных и коротких **make**-файлов (о них будем говорить позже).

2. Имя **объектного** файла, создаваемого по умолчанию, при включении только опции **-c** будет отличаться от имени исходного файла лишь расширением **.o**. Изменение имени объектного файла достигается одновременным включением наряду с опцией **-c** и опции **-o** с указанием желаемого имени.

   gfortran -c main.f -o main_A.o

3. Конечно, набирать вручную в командной строке имена нескольких объектных файлов для создания **загрузочного** не слишком удобно. Однако, нужные команды можно записать в **исполнимый файл** (*скрипт, сценарий*), а потом просто вызывать его для исполнения.

4. **Исполнимый файл** может быть не только **загрузочным**, но и **скриптом**.

   Загрузочный файл генерируется компилятором и представляет собой программу на машинном языке в виде двоичного кода.

   Скрипт (или **сценарий**) – это обычный текстовый файл с текстом на языке, *понимаемом* оболочкой, которая интерпретирует его в соответствующие машинные команды. Поэтому, если такому текстовому файлу приписать статус исполнимого, то его можно по имени вызывать на исполнение, как любую другую обычную **unix**-команду.

5. При работе с программой, которая состоит из десятков исходных файлов время ее трансляции резко замедляется. Поэтому перекомпиляция всей программы неприемлема. Требуется перекомпиляция лишь изменяемых исходных файлов. Из-за усталости можно забыть откомпилировать измененный файл. Как уже упоминалось, существует утилита **make**, которая реализует автоматическое слежение за ситуацией и гарантирует невозможность подобных проколов, компилируя из исходных файлов лишь те, время модификации которых более позднее, чем время генерации соответствующих объектных файлов. Программист должен написать соответствующий **make**-файл. О том, как он пишется и используется, узнаем из курса **Операционные системы**.

6. Часто вместо **исходный файл**, **объектный файл**, **загрузочный файл** говорят соответственно **исходный модуль**, **объектный модуль**, **загрузочный модуль**. В данном случае последние термины используются в качестве синонимов первых. Однако, важно помнить, что термин **модуль** в современном ФОРТРАНе может обозначать и особую единицу компиляции, которой не было в ФОРТРАНе-77 и которая обеспечивает возможность импорта из неё описанных в ней имён типов, переменных, процедур и интерфейсов.

1.7.9 Примеры команд вызова компиляторов.

Язык	Команда вызова компилятора	Получаемый файл
ФОРТРАН-95	gfortran prog.f -o main gfortran prog.f95 -o progfree gfortran -c prog.f gfortran -c prog.f -o prog1.o gfortran prog.o -o prog	main progfree prog.o prog1.o prog
C	gcc mytest.c -lm -o mytest	mytest
C++	g++ equation.cpp -lm -o equation	equation

1.7.10 О чем узнали из седьмого параграфа (кратко).

1. О правилах поведения в дисплейном классе.
2. Как набирать программы в редакторе **mcedit** в среде **mc**?
3. О технологии пропуска программ на ФОРТРАНе и СИ.
4. Отладка программы – поиск в ней **синтаксических** ошибок.
5. В свободном формате записи ФОРТРАН-программы значок амперсанда **&**, завершающий строку, означает, что на следующей строке помещается продолжение оператора, начатого на предыдущей.
6. Тестирование программы – поиск и устранение причин ее неправильной работы.
7. **НЕ ВСЕГДА компьютер способен получить верный результат даже по математически верной и формально правильно запрограммированной формуле.**
8. Чем и когда опасно вычитание почти равных значений типа **real**?
9. Исходный, объектный и загрузочный файлы.
10. По умолчанию **unix**-имя загрузочного файла – **a.out**
11. Как получить загрузочный файл с требуемым именем.
12. При генерации загрузочного файла выбираем для него **мнемонически значимое имя**; например, основное (**БЕЗ РАСШИРЕНИЯ**) имя исходного файла главной программы.
13. Как получить объектный файл с требуемым именем.
14. Термин **модуль** в старом ФОРТРАНе – просто синоним термина **файл с исходным, объектным или загрузочным кодом**.
15. Термин **модуль** в современном ФОРТРАНе обозначает особую единицу компиляции, нацеленную на импорт из неё имен типов, переменных, процедур и интерфейсов в ту программную единицу, к которой **модуль** подсоединён оператором **use**.
16. Команда вызова загрузочного файла из текущего каталога: **./имя_файла**.

1.7.11 Первое домашнее задание.

1. **Письменно ответить на вопросы, обосновать и зафиксировать свои выводы:**
   (a) "Какая операция и почему оказывается самой неудобной для ЭВМ в первой программе?"
   (b) "Какой конкретно ситуации надо стремиться избегать в своих формулах, если их предполагается использовать для ведения расчетов на ЭВМ?"
   (c) "При каком значении аргумента $\mathbf{x} = \mathbf{10^{-7}}$ или $\mathbf{x} = \mathbf{10^{-8}}$ в данной задаче легче обнаружить абсурдность (неверность) результата и почему?"
   (d) Пропуская программу при разных $\mathbf{x}$, вручную построить примерный график зависимости количества верных цифр в $\mathbf{t}$ от величины $\lg \mathbf{x}$.
   (e) Адаптировать текст программы к работе в режиме удвоенной точности и выяснить, пропуская их, при каких значениях аргумента наступают аналогичные эффекты?
   (f) Для режима удвоенной точности работы первой программы вручную построить график аналогичный графику из задачи **5**.
2. **На языках программирования ФОРТРАН-95, СИ и C++ написать линейную программу, которая вводит значения двух переменных целого типа и обменивает эти переменные своим содержимым, используя одну дополнительную рабочую переменную.**
3. **Задача похожа на предыдущую, НО необходимо обойтись без дополнительной рабочей переменной, моделируя процесс обмена придуманными расчетными формулами.**
4. **Задача аналогична предыдущей. Единственное отличие в условии – это требование – использовать только две четырехбайтовые переменные типа real в случае ФОРТРАНа или типа float в случае СИ и C++.**
5. **Письменно объяснить результаты работы программы последней задачи, вводя в качестве начальных значений переменных типа real числа:**
   a) **1.5 и 0.125**;
   b) **1.5 и 0.125e-7**;
   c) **1.5 и 0.125e-6**;
   d) **1.5 и 9.5367432e-7**;
   e) **1.5 и 9.6e-7**;
   f) **1.5 и 1.907348e-6**;
   g) **1.5 и 1.91e-6**;
   h) **1.5 и 0.1e-7**.

1.8 Ответы на часто задаваемые вопросы (FAQ).

Информацию по многим разделам программирования и математического обеспечения ЭВМ можно найти по адресу:

`http://gamma.math.spbu.ru/user/rus/cluster/docs.shtml`

1. Как подмонтировать **flash***–память?*
2. О переформатировании текстовых файлов (уяснение ситуации).
3. Некоторые пояснения к программе из 1.8.2 (чуть-чуть о вводе).
4. Переформатирование. Утилиты **dos2unix** *и* **unix2dos***.*
5. Перекодировка. Утилиты **iconv** *и* **konwert***.*
6. Понятие о скриптах.
7. Понятие о файле **.bash_profile***.*
8. Сценарий перевода из **cp866** *в* **utf8***.*
9. Модификация сценария **cp866utf8***.*
10. Сценарии **codeunix1**, **codeunix2** *и* **codeunix3***.*
11. Настройка кириллицы в mc.
12. Как выделить блок текста в файле? (редактор **mcedit***)?*
13. Как копировать выделенный блок?
14. Как скопировать блок из одного файла в другой?
15. Как выделять прямоугольные блоки?
16. Немного о настройке терминала.
17. Чуть-чуть о упаковке и распаковке архивов.
18. О чем узнали из восьмого параграфа (кратко)?

1.8.1 Как подмонтировать flash–память?

1. Вставить **flash**–память в соответствующий разъем.

2. Если не появилось окно со списком содержимого **flash**–памяти, то **левой** клавишей мыши нажать на иконку **flash**–памяти, получая его, и, при необходимости, выбирая далее в нём нужную рабочую поддиректорию **flash**–памяти.

3. Получить окно **домашнего** каталога, и посредством **левой** клавиши мыши выбрать из него нужную текущую поддиректорию.

4. Для копирования нужного файла (или поддиректории): **левой** клавишей мыши (и не отпуская её), подцепив нужное в исходной текущей директории, перетащить его в окно требуемой.

5. **Завершение работы с flash–памятью:**

   (a) Закрыть все выбранные подокна **flash**–памяти.

   (b) Подвести к зеленой лампочке на иконке **flash**–памяти указатель мыши и нажать ее **правую** клавишу.

   (c) Выбрать опцию **размонтировать**.

1.8.2 О переформатировании текстовых файлов (уяснение ситуации).

Иногда наработки, сделанные в средах **Windows** или **DOS**, необходимо перенести в среду **UNIX** или наоборот. Текстовый **UNIX-файл** содержит среди имеющихся в нем литер лишь один служебный символ – **символ начала новой строки**. Он называется **LF** (Line Feed) и имеет шестнадцатеричный код **0A**.

В этом можно убедиться, посмотрев на содержимое текстового **UNIX-файла** в шестнадцатеричном виде. Для этого в среде **mc** при наличии светового курсора на интересующем имени файла после нажатия клавиши **F3** надо дополнительно нажать клавишу **F4**.

В операционных системах **DOS** и **WINDOWS** для разделения строк текстового файла используется пара служебных символов **возврат каретки** (шестнадцатеричный код **0D**; общепринятое в литературе обозначение – **CR**, carrage return), и признак начала новой строки – **LF**, упоминавшийся чуть выше. Поэтому чтение в среде **UNIX** (например, редактором **mcedit**) содержимого скопированного **DOS**-файла, завершит каждую строку странным символом ∧ на черном фоне.

Если текст записан латиницей, то его можно компилировать и вызывать загрузочный файл. Однако при редактировании признак разделения новых текстовых строк, добавляемых в программу, будет естественно **UNIX**-овый. Поэтому в системах **WINDOWS** или **DOS** строки, отредактированные в **UNIX**, могут восприняться не как новые, а как продолжения старых, что неудобно. Кроме того, может оказаться, что данные, подготовленные в **UNIX** без приведения к **DOS**-формату, не будут читаться в **DOS**. Например, рассмотрим программу в файле **testf1.for** и данные к ней в файле **testf1.inp**.

```
        testf1.for                  :      testf1.inp
....................................:..........................
       program testf1               : 0.1234567890 333
       open (5,file='f1.inp')       :
       read (5,*) a, i              :
       write(*,*) 'a=',a,'  i=',i   :
       end                          :
....................................:..........................
     Результат ее пропуска в UNIX   : 0.123456791  i= 333
....................................:..........................
                            в DOS   : Invalid list-directed input
(при условии, что файл  testf1.inp  : file=f1.inp, unit=5 record=1,
 не приведен DOS-формату            : position=17 in testf1
```

Именно в 17-ой позиции файла данных находится признак новой **UNIX**-строки.

Вывод:

При переносе исходных текстов или данных между операционными системами UNIX и WINDOWS используем утилиты преобразования:

dos2unix	из dos(windows)-формата к unix-формату;
unix2dos	из unix-формата к dos- и windows-формату.

Помним, что переформатирование файла – **не перекодировка!!!**

1.8.3 Некоторые пояснения к программе из 1.8.2 (чуть-чуть о вводе).

При исследовательской работе исходные данные часто помещают в текстовый файл. Одно число, конечно, можно ввести и с экрана. Однако, это требует от нас знания смысловой проблемной нагрузки данного и его характерный рабочий диапазон. Даже при одном вводимом параметре трудно без заглядывания в текст программы и просмотра старых результатов вспомнить забытое.

Так что даже одиночное данное выгодно изначально помещать в файл на диске с указанием его проблемной нагрузки (на разговорном языке). Выведенное на экран содержимое такого файла позволит без труда вспомнить и изменить необходимые величины. Программа из пункта **1.8.2** – простейший пример организации ввода данных из файла исключительно через операторы ФОРТРАНа, без указания операций перенаправления оболочки (сравни с **1.6.11**).

ФОРТРАН-оператор **open (5,file='f1.inp')** сопоставляет реально существующему на диске файлу **f1.inp** устройство ввода под номером **5**. В качестве номера используется любое неотрицательное целое или имя целой переменной с аналогичным содержимым. Выбирает номер устройства программист. По умолчанию под устройство с номером **5** ФОРТРАН-компиляторы часто отводят экран. Поэтому, если в программе устройство ввода посредством оператора **open** переопределено, то и ввод будет происходить не с экрана, а из указанного файла. Оператор **open** – оператор открытия файла. Естественно перед запуском программы файл **f1.inp** с уже подготовленными в нем данными должен быть на диске в наличии.

Оператор **read (5,*) a, i** – оператор чтения данных в переменные **a** и **i** под управлением списка ввода с устройства под номером **5**. Если не хотим явно указывать место расположения данного во вводимой строке правила его преобразования в стандарт **IEEE**, то именно режим ввода, обозначаемый псевдонимом *, которая указана вторым параметром оператора **read**, оказывается наиболее бесхлопотным.

Рекомендация:

Данные для ввода всегда готовим в файле на диске.

1.8.4 Переформатирование. Утилиты dos2unix и unix2dos

Команда

dos2unix f1.for

преобразует файл **f1.for** к **unix**-формату, замещая исходное содержимое файла результатом. При этом имя файла может включать глобальные символы, например:

```
{\bf\$ dos2unix *.f}
dos2unix: converting file fff.f to UNIX format ...
dos2unix: converting file first00.f to UNIX format ...
dos2unix: converting file tstwrite.f to UNIX format ...
```

преобразует к **unix**-формату в текущей директории все файлы с расширением **.f**. При повторном обращении **dos2unix *.f** результат не портится. Типичная реакция утилиты – сообщение о приведении файла к **unix**-формату.

Для размещения результата переформатирования в новом файле используем вызов утилиты с опцией **-n**, указывая пару имён исходного и результирующего файлов:

dos2unix -n f2.f g2.f (в случае опции -n глобальные символы недопустимы)

Утилита **unix2dos** преобразует **unix**-формат файла к его **windows**-формату:

```
$ unix2dos f1.f
unix2dos: converting file f1.f to DOS format ...
```

Об остальных опциях утилит узнаём посредством **man dos2unix** и **man unix2dos**.

1.8.5 Перекодировка. Утилиты iconv и konwert

После переформатирования файла из **DOS** в **UNIX** редактировать текст можно лишь в том случае, когда он записан символами из **первой половины** кодовой **ASCII**-таблицы (первые 128 символов, которые, в частности, включают латиницу, цифры, знаки арифметических операций и знаки препинания). В последние десятилетия 20-го века для размещения алфавита отличного от латиницы (например, кириллицы) использовалась **вторая половина**. Появилось множество так называемых кодовых страниц (**code page**) – таблиц, сопоставляющих каждому **восьмибитному** числу некоторый символ. Упомянем **cp866**, **cp1251**, **koi8r**, используемых в **MS-DOS**, **WINDOWS** и **LINUX** соответственно. Поэтому наряду с переформатированием структуры файла при переходе с одной операционной системы на другую в случае наличия в тексте алфавита отличного от латиницы возникает и задача **перекодировки**. В разных реализациях **unix**-систем могут использоваться разные перекодировщики. Широко известна утилита **iconv**, содержащая несколько сотен кодовых таблиц и обеспечивающая перекодировку текста с любой из них. Например, пусть файл **chudos.txt** содержит в **DOS**-формате текст:

```
Муха-муха цокотуха,
Позолоченное брюхо,
Муха по полю пошла -
Муха денежку нашла.
```

После копирования **DOS**-файла **chudos.txt** в среду **UNIX** без переформатирования и чтения его в среде **mc** редактором **mcedit** посредством нажатия клавиши **F4**, увидим на экране примерно следующее:

```
1. ЦЕ-ЦЕ ФБЦЕ,^
2. Г ЮНЕ.^
3. ЦЕ  Н Х -^
4. ЦЕ Ц Х.^
```

Проведем переформатирование посредством **dos2unix**:

```
$ dos2unix CHUDOS.TXT
dos2unix: converting file CHUDOS.TXT to UNIX format ...
```

Теперь обзор **CHUDOS.TXT** по **F4** дает:

```
1. ЦЕ-ЦЕ ФБЦЕ,
2. Г ЮНЕ.
3. ЦЕ  Н Х -
4. ЦЕ Ц Х.
```

Налицо замена служебных символов соответствующим **UNIX**-аналогом (исчезли литеры ⋀ на черном фоне). Однако, в отличие от текста программы из пункта **1.8.2**, где использовалась латиница, здесь – полная абракадабра: не та кодировка – не **koi8r**.

Перекодируем посредством утилиты **iconv**:

```
$iconv -f=cp866 -t=koi8 CHUDOS.TXT -o CHUNIX.txt      Вместо знака =
                                                      возможен пробел, т.е:
$ iconv -f cp866 -t koi8 CHUDOS.TXT -o CHUNIX.txt
```

Здесь **iconv** – имя утилиты-перекодировщика;
-f – опция, указывающая на имя исходной кодировки;
-t – опция, указывающая на имя конечной кодировки;
CHUDOS.TXT – аргумент команды **iconv** (имя перекодируемого файла);
-o – опция, задающая имя файла с результатом перекодирования.
Теперь в файле **CHUNIX.txt** тот текст, который можно редактировать:

```
1. Муха-муха цокотуха,
2. Позолоченное брюхо.
3. Муха по полю пошла -
4. Муха денежку нашла.
```

Список кодировок, содержащихся в **iconv** можно получить дав команду

```
$ iconv -l > myiconv
```

В результате в текущей директории появится файл **myiconv**, в котором указаны названия всех кодовых таблиц, с которыми справляется **iconv**. В дисплейных классах факультета наряду с **iconv** имеется перекодировщик **konwert**. Пример:

```
konwert koi8r-cp1251 CHUNIX.txt -o CHUWIN.txt
```

Замечаниея:

1. Наряду с восьмибитными кодовыми таблицами был создан единый стандарт универсальной шестнадцатибитовой кодировки **Unicode** (Юникод), который включает в себя множество алфавитов. Сейчас широко распространена кодировка **UTF-8** (*Unicode Transformation Format* – формат преобразования Юникода), которая реализует возможности Юникода в сочетании с **восьмибитным** кодированием текста.

   Рекомендация.

   Работаем в UTF-8 !!!

2. В любом случае после перекодировки (или до нее) НЕ ЗАБЫВАТЬ про необходимость и переформатирования:

1.8.6 Понятие о скриптах.

При переносе файлов из сред **DOS** или **WINDOWS** в среду **UNIX** поочередный набор команд **dos2unix** и **iconv** (или **konwert**) и утомителен, и чреват досадными опечатками (особенно, когда файлов много). **UNIX** позволяет записать эту пару команд (или любые другие нужные команды) в отдельный файл, который затем вызывать на исполнение, как новую команду, так что все входящие в него команды выполнятся автоматически. Текст такого командного файла пишется **на языке оболочки**. Поэтому наряду с языками программирования ФОРТРАН и СИ полезно иметь навыки написания и **командных файлов**, которые часто называют **скриптами** или **сценариями оболочки**.

Ясно, что **сценарий** должен иметь статус **исполнимого файла**. Так файл с исходным текстом программы на ФОРТРАНе **не является исполнимым**. Для получения **исполнимого** аналога программы необходимо исходную программу откомпилировать с получением соответствующего **загрузочного файла** (исполнимого ЭВМ; см., например, пункт **1.7.8**).

В случае **командного файла** ситуация несколько иная. Его содержимое – текст, который исполняется интерпретатором оболочки. Поэтому после создания соответствующего текстового файла ему необходимо придать статус **исполнимого**, например, дать команду **chmod a+x имя_файла**.

Помещать скрипты удобно в какую-то одну специально отведенную директорию. Обычно имя такой директории называют **~/bin** – по аналогии с каталогом **/bin**, который содержит основные команды **UNIX**. При вызове скрипта, необходимо обеспечить к нему доступ из любой своей поддиректории (если, конечно, **скрипт** не находится в текущей), указав путь к нему (в противном случае операционная система сообщит, что **файл со скриптом не найден**). В то же время путь ко скрипту может оказаться и длинным, так что набирать его вручную (пусть даже и один раз) довольно утомительно, не говоря уже о том, что этот путь надо еще и помнить.

Проблема решается путем помещения маршрута к директории со своими скриптами в особый файл с именем **.bash_profile**, точнее записи строки маршрута в соответствующую переменную окружения, которая определяется в **.bash_profile**.

1.8.7 Понятие о файле .bash_profile.

.bash_profile – файл команд оболочки, который обычно размещается в домашнем каталоге пользователя. При каждом входе в систему, до начала интерактивного диалога с **UNIX**, выполняются команды, содержащиеся в **.bash_profile**. Эти команды служат для настройки режима работы терминала, установки значений так называемых **переменных окружения пользователя** (в частности, и переменной **PATH**, из которой операционная система, как раз и узнает поддиректорию, предназначенную пользователем, для хранения его оригинальных команд) и т.д. Узнать о содержимом переменной окружения **PATH** можно по команде **echo $PATH**. Например,

```
$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/aw/binmy
```

```
# .bash_profile                  ( Пример содержимого файла .bash_profile )
# Get the aliases and functions ( Значок #  - признак начала комментария )
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
# User specific environment and startup programs
PATH=$PATH:$HOME/binmy
BASH_ENV=$HOME/.bashrc
export USERNAME BASH_ENV PATH
```

Работу с языком оболочки освоите в курсе **Операционные системы**. Здесь же обратите внимание лишь на строку, обеспечивающую расширение стандартного содержимого переменной окружения **PATH** маршрутом к подкаталогу **binmy** из домашней директории. Таким образом, получен доступ из любых подкаталогов вашего личного **login**'а к любому скрипту, помещенному вами в директорию **binmy**.

1.8.8 Сценарий перевода из cp866 в utf8

Подготовим каким-нибудь редактором **скрипт**

```
#!/bin/bash
for j in * # заголовок цикла по всем именам файлов текущей директории
  do                            # открывающая рамка тела цикла
    dos2unix $j                 # $j - получение значения переменной j
    iconv -f cp866 -t utf8 $j -o iconv.tmp
    mv iconv.tmp $j
  done                          # закрывающая рамка тела цикла
```

и поместим его в директорию $\sim$/**bin**/ под именем **cp866utf8**, при условии, конечно, что в **.bash_profile** доопределена переменная **PATH**. Теперь файлы, подготовленные в **DOS**, после их копирования в некоторую текущую поддиректорию среды **UNIX**, можно преобразовать к правильному рабочему виду, дав команду **cp866utf8**. При этом **ВСЕ** файлы в текущей директории будут приведены к **unix**-формату и перекодированы из **cp866** в **utf8**.

Некоторые пояснения к скрипту:

1. Хотя значок # - признак комментария до конца строки, тем не менее, сочетание #**!** означает, что команда **bash** из каталога **bin** (дочернего по отношению к **корневому**) должна выполниться, получив в качестве аргумента текст, расположенный в данном файле. Другими словами, первая строка информирует оболочку, что обработка скрипта поручена интерпретатору **bash**. Поэтому в первой строке после команды **bash** ни в коем случае не помещаем какой-либо комментирующий текст, так как он будет принят за её аргумент.

2. Вместо опции **-o** при вызове **iconv** можно было бы использовать операцию перенаправления:

   ```
   iconv -f koi8r -t utf8 $j > iconv.tmp
   ```

1.8.9 Модификация сценария cp866utf8

Обычно после работы в дисплейном классе в текущей директории помимо текстовых исходных файлов могут появиться загрузочные и объектные. Ясно, что вряд ли имеет смысл осуществлять и их перенос на домашний **windows**-компьютер. Удобно модифицировать скрипт **cp866utf8**, приведенный в предыдущем пункте, так, чтобы он обрабатывал файлы лишь с заданным расширением. При этом возникает желание подать скрипту в качестве аргумента само расширение, например, так

```
$ cp866utf8a .cpp
```

Полезно знать:

Обозначение $1 интерпретируется оболочкой как первое слово, расположенное в командной строке после вызова скрипта.

Так что скрипт

```
#!/bin/bash
#                                        Скрипт   cp866utf8a
for j in *$1
  do
    dos2unix $j
    iconv -f=cp866 -t=utf8 $j > iconv.tmp
    mv iconv.tmp $j
  done
```

обеспечит в текущей директории преобразование формата и перекодировку всех файлов только с расширением **.cpp**. Если же захотим провести аналогичную обработку для файлов с расширением **.qqq** (при наличии таковых в текущей директории), то будет достаточен вызов **cp866utf8a .qqq**.

1.8.10 Сценарии codeunix1, codeunix2 и codeunix3

Сопоставим первому аргументу скрипта имя исходной кодировки, второму – имя требуемой, а третьему - расширение имени файла. Тогда скрипт

```
#!/bin/bash
#                                        Скрипт   codeunix1
for j in *$3
  do
    dos2unix $j
    iconv -f $1 -t $2 $j -o iconv.tmp
    mv iconv.tmp $j
  done
```

посредством вызова с тремя аргументами

codeunix1 cp1251 utf8 .for

позволит с лёгкостью приводить к **unix**-формату группу текстовых файлов требуемого расширения, используя при этом всю мощь утилиты **iconv**.

Скрипт **codeunix1** способен обработать группу файлов с каким-то одним расширением. Возможна ситуации, когда в текущей директории присутствует несколько групп файлов (причём у каждой своё расширение), и тогда скрипт **codeunix1** придётся вызывать несколько раз, подавая в качестве третьего аргумента соответствующее расширение.

Полезно знать:

Обозначение $* интерпретируется оболочкой как все аргументы, подаваемые на вход к скрипту, начиная с $1.

Поэтому скрипт **codeunix1** легко преобразовать в скрипт **codeunix2**, который будет за один вызов решать задачу обработки для нескольких групп файлов:

```
#!/bin/bash
#                                                    codeunix2
for j in $*
  do
    dos2unix $j
    iconv -f $1 -t $2 $j -o iconv.tmp
    mv iconv.tmp $j
  done
```

При его вызове (в отличие от вызова **codeunix1**) в качестве аргументов, начиная с третьего, пишем не расширения файлов, а глобальные (шаблонные, трафаретные) имена нужных групп файлов:

codeunix2 cp1251 utf8 *.cpp *.txt

Недостаток скрипта **codeunix2** в том, что согласно трактовке **$***, интерпретатор **bash** в операторе **for j in $*** помещает в переменную **j** в качестве имени файла и каждое из значений первых двух параметров скрипта, которые служат именами кодировок, а не файлов. Опуская детали, с которыми познакомитесь в курсе **Операционные системы**, приведём возможный пример скрипта, свободного от указанного недостатка:

```
#!/bin/bash
#                                                    codeunix3
a1=$1
a2=$2
shift 2
for j in $*
  do
    unix2dos $j
    iconv -f ${a1} -t ${a2} $j -o iconv.tmp
    mv iconv.tmp $j
  done
```

1.8.11 Настройка кириллицы в mc.

1. Вызвать **mc**.
2. Войти в главное меню, нажав клавишу **F9**.
3. Войти в подменю **Настройки**.
4. Выбрать пункт **Биты символов**.
5. Поставить поочередно поддержку кириллицы в полях **'Полный восьмибитный вывод'** и **'Полный восьмибитный ввод'** (путем нажатия клавиши **пробел**).
6. Сохранить настройки.

1.8.12 Как выделить блок текста в файле (редактор mcedit)?

Первый способ

1. Поместить курсор на начальный символ выделяемого блока.
2. Нажать клавишу **F3** (фиксируем начало блока).
3. Поместить курсор на последний символ выделяемого блока.
4. Нажать клавишу **F3** (фиксируем окончание блока).

Второй способ

1. Поместить курсор на начальный символ выделяемого блока.
2. Нажав и НЕ отпуская клавишу **Shift**, использовать клавиши управления курсором вплоть до окончательного выделения блока.
3. Отпустить клавишу **Shift**.

1.8.13 Как копировать выделенный блок?

1. Выделить копируемый блок текста (см. предыдущий пункт).
2. Поместить текстовый курсор в позицию, с которой предполагается вставка.
3. Нажать клавишу **F5** для копирования блока (или **F6** для перемещения).
4. Не забыть сохранить внесенное изменение, нажав клавишу **F2**.

1.8.14 Как скопировать блок из одного файла в другой?

Предполагается, что на экране два окна с содержимым обоих файлов, которое получено посредством **mcedit**.

1. Выделить нужный блок текста.
2. Нажать клавишу **F9** (выход в меню **mcedit**).
3. Поместить световой курсор на опцию **Файл**.
4. Нажать клавишу **Enter**, раскрывая меню опции **Файл**.
5. Поместить световой курсор последнего меню на опцию **Копировать в файл**.
6. Нажать клавишу **Enter**.
7. Выбрать кнопку <**дальше**> и нажать **Enter**.
8. Переключиться на терминал с файлом–приемщиком.
9. Поместить курсор в позицию, начиная с которой хотим вставить копию блока.
10. Нажать клавишу **F9** (выход в меню **mcedit**, редактирующего второй файл).
11. Поместить световой курсор на опцию **Вставить в файл** и нажать клавишу **Enter**.
12. Нажать клавишу **F2** для сохранения изменения.

Есть и более практичный способ копирования выделенного блока текста из одного файла в другой. Способ использует так называемые **горячие клавиши**, нажатие которых инициирует немедленную реакцию компьютера без высветки промежуточных запросов и утомительного бегания указателем мыши по необходимым виртуальным кнопкам. Узнать о сопоставлении **горячей клавише** (или их комбинации) того или иного действия можно высветив нужное подменю после нажатия **F9**. Например, из меню **Файл** видно, что копирование **выделенного блока** в буферный файл достигается нажатием клавиш **Cntrl** и **f** (точнее достигается вызов поля имени буферного файла, в котором намереваемся сохранить выделенный блок). Важно только перед упомянутым нажатием **НЕ ЗАБЫТЬ** выделить блок. Если нет желания изменять предлагаемое по умолчанию имя **cooledit.clip** – имя буферного файла, то нажатие клавиши **ENTER** обеспечит запись в него выделенного блока. После переключения на терминал с файлом-приемником вызов поля имени буферного файла достигается совместным нажатием клавиш **Shift** и функциональной **F5**, о чем подменю опции **Файл** своеобразно сообщает, требуя от пользователя нажать клавишу **F15**, которая на клавиатуре часто отсутствует. Далее – установка курсора на нужное знакоместо экрана и копирование из буфера посредством **Enter** с последующим сохранением измененного файла (**F2**).

1.8.15 Как выделять прямоугольные блоки?

Редактор **mcedit** позволяет выделять блоки не только построчно, но и поколоночно. Так, если в процессе заготовки какой-то таблицы один из ее столбцов по вине наборщика оказался не в своей графе (например, второй столбец оказался третьим), то посредством инструментария **mcedit** легко поменять столбцы местами, не портя взаиморасположения остальных. Пусть, например, в некотором файле ручным или программным путем получилась таблица:

Аргумент	Первая разность	Функция	Вторая разность	Третья разность
0	1	0	6	6
1	7	1	12	6
2	19	8	18	6

в которой третий столбец необходимо поместить на место второго, а второй на место третьего.

1. Устанавливаем курсор на знакоместо любого угла выделяемого прямоугольника.
2. Нажимаем клавиши **Shift** и, не отпуская ее, функциональную **F3**.
3. Отпускаем нажатые клавиши (начало блока зафиксировано).
4. Клавишами управления управления курсором **leftarrow** (или *rightarrow*) смещаем курсор на нужное число позиций влево (или вправо).
5. Клавишами управления управления курсором **uparrow** (или *downarrow*) смещаем курсор на нужное число позиций вверх (или вниз).
6. Нажимаем функциональную клавишу **F3**, окончательно фиксируя блок.
7. Помещаем курсор в левый верхний угол второго столбца.
8. Нажимаем функциональную клавишу **F6**.

В результате получаем желаемую таблицу, которую спасаем на диске, нажав **F2**.

Аргумент	Функция	Первая разность	Вторая разность	Третья разность
0	0	1	6	6
1	1	7	12	6
2	8	19	18	6

1.8.16 Немного о настройке терминала.

Иногда текст, высвечиваемый тем или иным редактором, несмотря на явную понятность, оказывается неудобным для работы (например, курсор слишком далеко от промпта командной строки, разные расстояния между буквами, временное искажение текста при пробегании по нему курсора и др.). Причина в сбое настройки терминала, в частности в использовании невыгодных фонтов. Для подбора приемлемого режима следует после высветки окна терминала инициировать (например, указателем мыши) вызов опции **Settings** после чего из предложенного меню выбрать строчку **Fonts**, а затем поварьировать установку различных фонтов с целью добиться наиболее удобочитаемого результата. Обычно вполне достаточно установить фонт под названием **Normal**.

1.8.17 Чуть-чуть про упаковку и распаковку архивов

Перед копированием данных на *флешку* их можно упаковать (сжать), что существенно экономит память на *флешке* и время копирования. В **UNIX**-системах широко используется утилита **tar**, которая позволяет создавать архивы, в частности, и из упакованных утилитами сжатия файлов. Далее приводится пара простых примеров.

1. Создание тарбола (архива с его упаковкой) утилитой **tar**:

   tar -cvzf mycat.tar.gz .

   В результате все файлы текущей директории **mycat** будут сжаты и заархивированы. Сжатый архив **mycat.tar.gz** будет в текущей директории. Точка в конце через пробел после gz - **обязательна**, т.к. обозначает имя текущей директории.

2. Распаковка в директории, хранящей тарбол:

   tar -xvzf mycat.tar.gz

   Здесь **mycat.tar.gz** - имя упакованного архива.

3. Опции означают следующее:

   - **-c** — первая буква слова **create** (создать) сообщает утилите **tar**, что архив нужно создать, а вкупе с опцией **-z**, что полученный архив следует упаковать утилитой **gzip**.
   - **-v** — в сцепке **-xvzf** означает, что содержимое входного файла должно быть разбито на файлы, из которых входной файл был составлен.
   - **-z** — файл с расширением **.tar.gz** перед разархивированием должен быть распакован утилитой **gunzip**.
   - **-f** — имя, записанное после команды **tar** есть имя файла с которым **tar** должен работать. Эта опция не всегда обязательна, но её рекомендуется использовать.
   - **-x** — извлечение данных из архива.

1.8.18 О чем узнали из восьмого параграфа (кратко)?

1. Как подмонтировать **flash**-память?
2. **dos2unix** и **unix2dos** – утилиты переформатирования.
3. Как переформатировать **DOS**- и **WINDOWS**-файлы в **UNIX**-формат?
4. О перекодировщиках **iconv** и **konvert**.
5. Как на ФОРТРАНе проще всего осуществить ввод данных из файла?
6. О назначении файла (**.bash_profile**)
7. **Скрипт** – (**сценарий оболочки**). Зачем он нужен?
8. **$1** – обозначение в скрипте, которое **bash** интерперетирует как первый аргумент скрипта, указанный при вызове (соответственно **$2** и **$3** – второй и третий аргументы).
9. **$*** – обозначение в скрипте, которое **bash** интерперетирует как **все** аргументы скрипта (начиная с **$1**), указанные при его вызове.
10. Примеры использования элементарных скриптов.
11. О настройке кириллицы в **mcedit**.
12. Как выделить строковый блок в тексте, редактируемом **mcedit**?
13. Как выделить прямоугольный блок в тексте, редактируемом **mcedit**?
14. Как копировать блок в пределах одного файла?
15. Как копировать блок из одного файла в другой?
16. Умение пользоваться утилитой **tar** очень полезно.
17. **Массу полезной информации можно получить выйдя на сайт класса параллельных вычислений: http://gamma.math.spbu.ru/user/rus/cluster/docs.shtml**
    (a) В разделе **Документация** на русском языке выложены книги и по **C++**, и по **ФОРТРАНу-95**, и по параллельным вычислениям, и по операционной системе **LINUX**. Многое, что дается на занятиях, представляет краткие выжимки или из упомянутых книг, или из других печатных изданий.
    (b) В режиме командной строки можно воспользоваться справочной системой вызываемой командой **info** (в частности, по GNU-ФОРТРАНу, транслятор которого вызывается командой **gfortran**). Естественно, придется затратить некоторое время на ее освоение, но привыкнув к правилам **info** можно быстро находить справки по интересующим темам, если последние включены в ее систему.

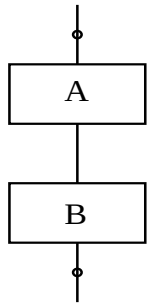
2 Базовые алгоритмические структуры.

2.1 Правильная программа.

1. имеет один вход и один выход;
2. состоит из конечного количества шагов;
3. не содержит недостижимых фрагментов.

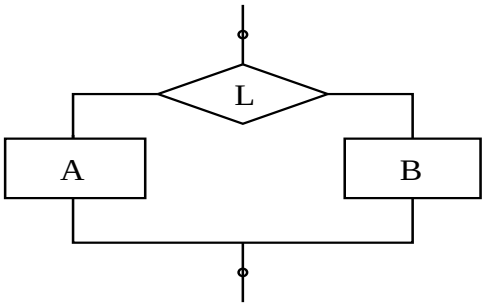
Ее всегда можно сконструировать из трех базовых алгоритмических структур.

Следование



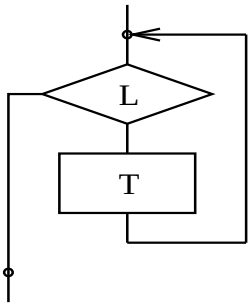
Последовательное выполнение шагов **A** и **B**

Развилка



При истинности условия **L** один раз выполнится лишь шаг **B**, иначе – только шаг **A**.

Повтор



Пока логическое условие **L** истинно, повторение тела цикла **T**

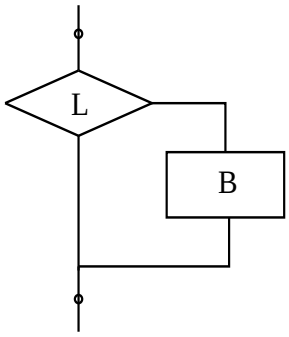
Повтор с последней блок-схемы часто называют **повтор с предусловием**, так как расчет логического выражения **L** всегда происходит **перед** выполнением тела цикла **T**. При этом, если начальное значение **L** окажется **ложью** (**.false.** в ФОРТРАНе; **false** в C++; **0** в СИ и C++), то тело цикла **T** вообще не выполнится ни разу.

При построении программ указанные алгоритмические структуры можно комбинировать по схеме **следования** и (или) по схеме **вложения** (погружения) одной структуры в другую. Например, каждый из шагов **A**, **B** или **T** может состоять из целой серии развилок, следований, повторов. Число вложений, естественно, конечно.

Из соображений удобства, к базовым алгоритмическим структурам отнесены еще три (их просто построить из трех, приведенных выше): **полуразвилка**, **выбор** (вариант, переключатель) и **повтор с постусловием**.

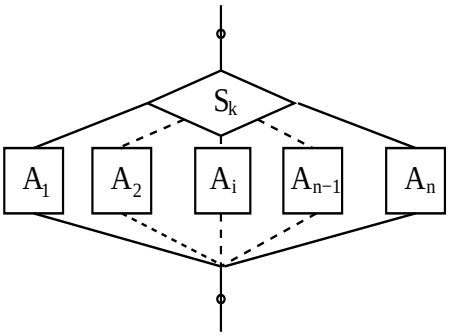
Замечание: Под термином **правильная программа** здесь понимается программа, получаемая при традиционной (то есть последовательной) модели программирования, когда команды программы всегда выполняются последовательно друг за другом, обрабатывая в конкретный момент времени один элемент данных. Идеи и методы параллельного программирования изложены, например, в книге [12]. Данное пособие их не касается.

Полуразвилка



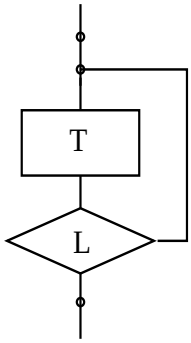
Одна из двух альтернатив развилки – пустое действие

Выбор



Развилка обобщается на число альтернатив большее двух. Альтернатива выбирается по значению, соответствующему ее номеру

Повтор с постусловием



Повторяй тело цикла до тех пор, пока НЕ выполнится условие **L**

В **повторе с постусловием** тело цикла **T** всегда выполняется хотя бы один раз, поскольку условие **L**, управляющее продолжением или прекращением повтора, проверяется после выполнения тела цикла.

2.2 Следование

Следование реализуется размещением очередного оператора в новой строке (если предыдущий оператор не нацелен на обход очередного) или завершением оператора символом ;, который

- в СИ служит синтаксическим завершением каждого оператора (исключение составной оператор, заключённый в фигурные скобки);
- а в ФОРТРАНе служит просто разделителем.

2.3 Замечания

1. **Полуразвилка**, **развилка** и **повтор** — это управляющие алгоритмические структуры, которые, в частности, содержат запись **булева выражения** (логического выражения).
2. Запись значений **истина** и **ложь** в ФОРТРАНе и СИ:

Название значения	**ФОРТРАН**	**C**	**C++**
Истина	**.true.**	не нуль	не нуль или **true**
Ложь	**.false.**	**0**	**0** или **false**

3. **Служебное слово для обозначения булева типа**

Название типа	ФОРТРАН	C	C++
Логический (или булев) тип	**logical**	Отсутствует	**bool**

4. Запись булева выражения часто содержит **операции отношения**:

Название операции	Типичное обозначение	ФОРТРАН	СИ и СИ++
Строго меньше	$<$	**.lt.** или $<$	$<$
Меньше или равно	$\leq$	**.le.** или $<=$	$<=$
Равно	$=$	**.eq.** или $==$	$==$
Не равно	$\neq$	**.ne.** или $/=$	**!=**
Больше или равно	$\geq$	**.ge.** или $>=$	$>=$
Строго больше	$>$	**.gt.** или $>$	$>$

и **операции математической логики**:

Название операции	Типичное обозначение	ФОРТРАН	C или C+
Отрицание	$\neg$	**.not.**	**!**
Конъюнкция (логическое умножение)	$\wedge$	**.and.**	**&&**
Дизъюнкция (логическое сложение)	$\vee$	**.or.**	||
Исключающее или		**.xor.**	
Эквивалентность (равнозначность)	$\equiv$	**.eqv.**	
Неэквивалентность (неравнозначность)		**.neqv.**	

5. В C любая из логических операций или операций отношения в случае истинности ее результата дает **1**, а в случае ложности дает **0**. Тем не менее **!777** дает **0** (то есть **ложь**).

2.4 Полуразвилка.

1. Синтаксис неполного условного оператора.
2. Пример использования полуразвилки на ФОРТРАНе-77.
3. Пример использования полуразвилки на ФОРТРАНе-95.
4. Пример использования полуразвилки на СИ.
5. Пример использования полуразвилки на СИ++.

2.4.1 Синтаксис неполного условного оператора.

```
  ФОРТРАН        !      СИ         !  Здесь
                 !                 !
 if (L) then     !  if (L)         !    L  - булево выражение;
   A1; A2        !    { A1; A2;    !    Ai - операторы языка
   A3; ...       !      A3; ...;   !         (шаги алгоритма)
   An            !      An;        !
 endif           !    }            !
```

2.4.2 Пример использования полуразвилки на ФОРТРАНе-77

Запись алгоритма поиска решения квадратного уравнения.

```
      program testif
      implicit none
      real*8 a, b, c, d, sd, a2, x1, x2
      a=1;  b=-3.1;  c=2.2
      write(*,*) ' a=',a,' b=',b,' c=',c
      d=b*b-4*a*c
      if (d < 0d0) then
                  write(*,*) ' Вещественных',
     >            ' корней нет  '
      endif
      if (d.ge.0d0) then
        sd=dsqrt(d)  ; a2=2*a
        x1=(-b-sd)/a2; x2=(-b+sd)/ a2
        write(*,*) ' x1=', x1,' x2=',x2
      endif
      write(*,*) ' Задача завершена.'
      end
```

Результат пропуска имеет на экране вид:

```
a=  1. b= -3.0999999 c=  2.20000005
x1=  1.10000017 x2=  1.99999974
Задача завершена.
```

Замечания по синтаксису:

1. GNU-компилятор с ФОРТРАНа при фиксированном формате записи допускает в одной строке несколько операторов, разделённых символом **;** (точка с запятой). Однако, размещение их в строке после **then** или **else** требует **;** и после этих служебных слов.

2. Вместо **if (L) then; A; endif** (если **A** – **единственное действие!!!**) иногда удобнее условный оператор **if (L) A** (без закрывающей рамки **endif**), в котором действие **A** размещается в той же строке, что и логическое выражение **(L)**.

3. В старых версиях языка ФОРТРАН оператора **if (L) then ... endif** не было вовсе . Для выполнения нескольких действий в случае истинности **L** приходилось посредством оператора **if (.not.L)** и оператора передачи управления **goto** моделировать работу конструкции **if (L) then** . Например, так

```
      if (.not.L) goto 10 ! т.е, если L - ЛОЖЬ, то переход на метку 10
        A1
        A2
        ...
        AN
   10 continue            ! оператор приема передачи управления
```

   Такие конструкции часто можно видеть в старых ФОРТРАН-программах.

4. Оператор **if (L) then** имеет **рамочную структуру**: **if** – открывающая рамка, а **endif** – закрывающая. **Рамочная структура** повышает надежность программирования по сравнению с СИ. В СИ отсутствие рамочной структуры при необходимости выполнить несколько действий (если **L** истинно), вынуждает объединять их в одно составное (**составной оператор**), заключая между операторными скобками **{** и **}** .

   Если в СИ забыть поставить операторные скобки, то при условии истинности **L** выполниться лишь первый оператор **A1**, а выполнение остальных от значения **L** зависеть не будет. **A2, A3, A4** будут выполняться всегда.

   В ФОРТРАНе отсутствие закрывающей рамки **endif** приведет к выдаче сообщения о синтаксической ошибке еще на этапе трансляции программы и, тем самым, сэкономит массу времени, которое невнимательный СИ-программист может потерять при выяснении причины неверности результата, вызванного отсутствием операторных скобок.

Замечания по счетной части:

Кажется, что результат приемлем: вместо **-1.1** и **2.0** получено **x1= 1.10000017** и **x2= 1.99999974**. Однако в программе переменные вещественного типа описаны **восьмибайтовыми** (**real*8**), а это означает, что в расчете используются данные, записываемые (в десятичной системе счисления) 15-16 десятичными цифрами мантиссы. Результат же, полученный программой, верен лишь в пределах семизначной. **Налицо первая неувязка**.

Величины исходных коэффициентов уравнения **b, c** также подозрительно неточны: **a= 1. b= -3.0999999 c= 2.20000005**. Если относительно **a** и **b** еще можно предположить, что в первом случае **нули**, а во втором **девятки** продолжаются вплоть до семнадцатой цифры, то пятерка в девятой цифре мантиссы **c** должна насторожить. Модифицируем программу, потребовав от нее вывода переменных **a, b, c, x1, x2** с шестнадцатьюзначной мантиссой (ниже приводится ее текст и результаты пропуска):

```
      program testifdiv16
      implicit none
      real*8 a, b, c, d, sd, a2, x1, x2
      a=1;  b=-3.1;  c=dble(2.2);             write(*,1000) a,b,c
      d=b*b-4*a*c
      if (d < 0d0) then;  write(*,*) ' Вещественных',
     >                               ' корней нет  '
      endif
      if (d.ge.0d0) then
        sd=dsqrt(d)  ; a2=2*a; x1=(-b-sd)/a2; x2=(-b+sd)/ a2
        write(*,1001) x1,x2
      endif
      write(*,*) ' Задача завершена.'
 1000 format(1x,' a=',d23.16,' b=',d23.16,' c=',d23.16)
 1001 format(1x,' x1=',d23.16,' x2=',d23.16)
      end
```

Результаты пропуска:

```
a= 0.1000000000000000E+01 b=-0.3099999904632568E+01 c= 0.2200000047683716E+01
x1= 0.1100000169542151E+01 x2= 0.1999999735090418E+01
Задача завершена.
```

1. Значение **a** в пределах всей разрядной сетки действительно точно равно единице — относительно **a** наше предположение оправдалось.
2. Однако в отношении **b** наша надежда НЕ ОПРАВДАЛАСЬ!
3. Значения **b** и **c** наводят на мысль, что несмотря на описание переменных с удвоенной точностью, их численные значения всё-таки задаются с одинарной.
4. **Причина – в нашей небрежности!** В тексте программы использована форма записи констант с фиксированной запятой без порядка. Это – форма записи констант с одинарной точностью. Транслятор свято выполнил наше требование: значение константы с одинарной точностью преобразовал по форме к типу **real(8)**, сохранив погрешность округления одинарной, и записал в переменные, предназначенные для хранения значений с удвоенной.
5. **Правило ФОРТРАНА-77: Единственный способ записи константы с удвоенной точностью – это ее запись в форме D** (вспоминаем параграф **1.6.6**).

Приведем правильный текст программы и результаты ее работы.

```
      program testifdiv16
      implicit none
      real*8 a, b, c, d, sd, a2, x1, x2
      a=1d0;  b=-3.1d0;  c=2.2d0;
      write(*,1000) a,b,c
      d=b*b-4*a*c
      if (d < 0d0) then
                  write(*,*) ' Вещественных',
     >            ' корней нет  '
      endif
      if (d.ge.0d0) then
        sd=dsqrt(d)  ; a2=2d0*a
        x1=(-b-sd)/a2; x2=(-b+sd)/ a2
        write(*,1001) x1,x2
      endif
      write(*,*) ' Задача завершена.'
 1000 format(1x,' a=',d23.16,' b=',d23.16,' c=',d23.16)
 1001 format(1x,' x1=',d23.16,' x2=',d23.16)
      end
```

Результаты пропуска:

```
 a= 0.1000000000000000E+01 b=-0.3100000000000000E+01 c=0.2200000000000000E+01
 x1= 0.1100000000000000E+01 x2= 0.2000000000000000E+01
 Задача завершена.
```

2.4.3 Пример использования полуразвилки на ФОРТРАНе-95

```
      program testifdiv16
      use my_prec
      real(mp) a, b, c, d, sd, a2, x1, x2
      a=1.0_mp;  b=-3.1_mp;  c=2.2_mp
      write(*,*) ' # a=',a
      write(*,*) ' # b=',b
      write(*,*) ' # c=',c
      d=b*b-4*a*c
      if (d < 0.0_mp) then
                  write(*,*) ' Вещественных',  &
                  ' корней нет  '
      endif
      if (d .ge. 0.0_mp) then
        sd=sqrt(d)  ; a2=2.0_mp*a
        x1=(-b-sd)/a2; x2=(-b+sd)/ a2
        write(*,*) ' # x1=',x1,'  x2=',x2
      endif
      write(*,*) ' Задача завершена.'
      end
```

Замечания

1. В тексте программы демонстрируются возможности ФОРТРАНа-95: использование модуля **my_prec**, из которого (в случае его подключения) любая единица компиляции наследует переустановку правила умолчания и имя целочисленной константы **mp**. Последнее в круглых скобках можно указать в операторе описания переменных вещественного типа (**real(mp)**), что принципиально упрощает перевод программы на ячейки иной допустимой разрядности.

2. Форма записи констант вещественного типа, использующая окончание **_mp** более универсальна, так как изменение **mp** в модуле автоматически переведёт константы в желаемую разрядность.

3. В современном ФОРТРАНе во многих случаях нет нужды при вызове встроенной функции указывать её специфическое имя (например, **dsqrt**), так как задействован механизм перегрузки функций. Гораздо проще указать **родовое** имя **sqrt**, которое в случае изменения типа аргумена выберет соответствующее специфическое имя автоматически.

4. Текст программы дан в свободном формате, хотя на первый взгляд кажется, что в фиксированном. Заметить это можно по значку **&**, который служит признаком продолжения текущего оператора в следующей строке.

2.4.4 Пример использования полуразвилки на СИ.

```
#include <stdio.h>
int main()
{ double a, b, c, d, sd, a2, x1, x2;
  a=1; b=5.0; c=6.0;
  printf(" a=%23.16e b=%23.16e c=%23.16e\n",a,b,c);
  d=b*b-4*a*c; if (d<0)  printf(" Вещественных корней нет\n");
  if (d>=0) { sd=sqrt(d);
              a2=2*a; x1=(-b-sd)/a2; x2=(-b+sd)/ a2;
              printf(" x1=%e\n",x1); printf(" x2=%e\n",x2);
            }
 printf("Задача завершена.\n"); return 0;
}
```

Компиляция, вызов и результат работы СИ-программы:

```
$ gcc ifdiv2.c -lm -o ifdiv
$ ./ifdiv
 a= 1.0000000000000000e+00 b= 5.0000000000000000e+00 c= 6.0000000000000000e+00
 x1=-3.000000e+00
 x2=-2.000000e+00
   Задача завершена.
```

Если при вызове **gcc** опустить опцию **-lm** (подключения математической библиотеки), то на этапе линковки обнаружится, что в программе используется ссылка (обращение) к неизвестной функции **sqrt**.

Напоминание:

1. Печать результата (x1,x2) по формату (**%e**) имеет по умолчанию семизначную мантиссу, несмотря на то, что, опять-таки по умолчанию, вещественные константы имеют тип **double**.

2. При необходимости получить на печати мантиссу шестнадцатизначную явно указываем между **%** и **e** ширину поля, отводимого для записи всего значения и количество цифр мантиссы **%23.16e** (в качестве примера см. контрольную печать **a, b, c**).

3. При вводе данного вещественного типа удвоенной точности функцией **scanf** соответствующее правило формата **%e, %f, %g** обязательно должено снабжаться префиксом **l**, то есть, например, так

   ```
   scanf("%le %le %le", &a, &b, &c);
   ```

   Если его не использовать, то данное введется с одинарной точностью.

2.4.5 Пример использования полуразвилки на СИ++.

Все базовые алгоритмические структуры на языке СИ++ имеют тот же синтаксис, что и на языке СИ. Тем не менее приведем исходные текст на СИ++ особо, используя, характерные для него операторы ввода-вывода **cin** и **cout** (потоки ввода-вывода).

```
#include <iostream>
using namespace std;
int main()
{
double a, b, c, d, sd, a2, x1, x2;
 a=1; b=5l; c=6l;
 cout<<" a="<<a<<" b="<<b<<" c="<<c<<endl;
 d=b*b-4*a*c;
 if (d<0)
  { cout << " Вещественных корней нет" << endl;}
 if (d>0)
    {   sd=sqrt(d);
        a2=2*a;
        x1=(-b-sd)/a2;
        x2=(-b+sd)/ a2;
        cout<<" x1="<<x1;
        cout<<" x2="<<x2<<endl;
    }
     cout<<"Задача завершена."<< endl;
 return 0;
}
```

Замечание: Ответ на вопрос о том, как осуществлять форматированный ввод-вывод посредством **cin** и **cout**, узнаем в седьмой главе.

2.5 Развилка.

1. Синтаксис полного условного оператора.
2. Примеры использования развилки на ФОРТРАНе и СИ.
3. Условная операции СИ **? :**.
4. Арифметический условный оператор ФОРТРАНа.
5. Ветвящаяся развилка (ФОРТРАН: синтаксис; пример).

2.5.1 Синтаксис полного условного оператора.

```
   ФОРТРАН     !      СИ         !  Здесь
 if (L)  then  !  if (L)        !    L  - булево выражение;
   A1; A2      !    { A1; A2;   !    Ai - операторы языка
   A3; An      !      A3; An; } !         (шаги алгоритма)
         else  !   else          ! Совокупность действий, входящих
   B1; B2      !    { B1; B2;   ! в одну альтернативу ФОРТРАНа
   Bm          !      Bm;       ! часто называют (БОК)
 endif         !    }           ! БЛОК ОПЕРАТОРОВ И КОНСТРУКЦИЙ
```

2.5.2 Примеры использования развилки на ФОРТРАНе и СИ.

Решение предыдущей задачи с использованием полного условного оператора:

```
      program testifdiv16                  ! Решение задачи из параграфа 2.4
      implicit none                        ! посредством полного условного
      real*8 a, b, c, d, sd, a2, x1, x2    !                      оператора.
      a=1d0;  b=-3.1d0;  c=2.2d0;          ! ФОРТРАН-77.
      write(*,1000) a,b,c                  !---------------------------
      d=b*b-4*a*c
      if (d < 0d0) then
                        write(*,*) ' Вещественных',
     >                             ' корней нет  '
                  else
                        sd=dsqrt(d)  ; a2=2d0*a
                        x1=(-b-sd)/a2; x2=(-b+sd)/ a2
                        write(*,1001) x1,x2
      endif
      write(*,*) ' Задача завершена.'
 1000 format(1x,' a=',d23.16,' b=',d23.16,' c=',d23.16)
 1001 format(1x,' x1=',d23.16,' x2=',d23.16)
      end
```

Результаты ее работы:

```
 a= 0.1000000000000000E+01 b=-0.3100000000000000E+01 c=0.2200000000000000E+01
 x1= 0.1100000000000000E+01 x2= 0.2000000000000000E+01
 Задача завершена.
```

```
#include <stdio.h>                          // Программа на СИ.
int main()                                  //_________________
{
 double a, b, c, d, sd, a2, x1, x2;
 a=1; b=5l; c=6l;  printf(" a=%23.16e b=%23.16e c=%23.16e\n",a,b,c);
 d=b*b-4*a*c;
 if (d<0) printf(" Вещественных корней нет\n");
   else { sd=sqrt(d); a2=2*a;
          x1=(-b-sd)/a2; x2=(-b+sd)/ a2;
          printf(" x1=%23.16e\n",x1); printf(" x2=%23.16e\n",x2);
         }
 printf("Задача завершена.\n");  return 0;
}
```

Замечания:

1. При необходимости сопоставить альтернативам несколько действий заключаем соответствующие операторы в операторные скобки. Если же это требование не выполнить для альтернативы **else** (в данном случае), то результат работы при **b=0** окажется странным: программа сообщит об отсутствии корней и **... напечатает значения обоих** (конечно неверные).

2. Особенно важны операторные скобки при работе со вложенными **if**. Без операторных скобок СИ-транслятор каждое ключевое слово **else** будет связывать с наиболее близким **if**, для которого нет **else**. Например,(см. также [15] стр. 69):

```
#include <stdio.h>            //       #include <stdio.h>
int main()                    //       int main()
{ int a=2, b=7, c=3;          //       { int a=2, b=7, c=3;
  if (a>b)                    //         if (a>b)
    { if (b<c) c=b; }         //           if (b<c) c=b;
  else c=a;                   //         else c=a;
   printf("c=%d\n",c);        //         printf("c=%d\n",c);
}                             //        }
```

Левая программа напечатает **c=2.0**, а правая – **c=3.0**.

2.5.3 Условная (тернарная) операция СИ ? :

Порой используется для получения более элегантной записи. Иногда называется **тернарной** (ternary – три, тройка, т.е. состоящей из трех составных частей).

(Операнд1)	?	Операнд2	:	Операнд3
(либо целого,либо вещественного типа, либо типа указатель)		вычисляется, если операнд1 не равен нулю		вычисляется если операнд1 равен нулю

```
Пример.                              // переменной max присваивается
            max=(a<=b) ? b : a       // максимальное из значений a и b.
```

2.5.4 Арифметический условный оператор ФОРТРАНа

Часто использовался в древних ФОРТРАН-программах. В отличие от логического условного оператора после служебного слова **if** пишется НЕ **логическое выражение**, а **арифметическое**. Последнее вычисляется и, в зависимости от того меньше, равно или больше его значение нуля, управление передается на указываемую метку.

```
   if (арифметическое)  10 , 20, 30  ! <--= Это номера меток.
        выражение                    !      Их придумываем сами.
10 continue         ! Если значение арифметического выражения < 0,
           A10      ! то управление передается на метку 10;
           goto 77  ! обход оставшихся альтернатив;
20 continue         ! Если значение AB = 0, то на метку 20;
           A20      !
           goto 77  ! обход оставшихся альтернатив;
30 continue         ! Если AB > 0, то на метку 30.
           A30
77 continue
```

Рекомендации:

1. **Лучше никогда не использовать.** Метки предоставляют возможность **'пометить'** не то, **'послать'** не туда, куда надо, **'принять'** не там, где хотелось, нарушить требование **один вход–один выход**.

2. Если решились, то лучше с целочисленным арифметическим выражением, т.к. оно вычисляется абсолютно точно (не заражено погрешностью округления).

3. Если же рискуем выбирать ветвь алгоритма, опираясь на выражение типа **real**, то вся ответственность за последствия — на нас. Так, если на ЭВМ вычислить при разных значениях аргумента **x** выражение $\sqrt{\mathbf{x}} - \mathbf{e}^{\mathbf{0.5} * \ln(\mathbf{x})}$, которое формально можно (**но разумно ли?**) использовать при проверке на точное равенство нулю в условных операторах ФОРТРАНа и СИ. Результат для любого из приведенных ниже значений аргумента аналитически точно должен равняться нулю, а ЭВМ получает следующее:

x	**real*4** или **float**	**x**	**real*8** или **double**
2.099990	0.000E+00	2.099990	-0.222E-15
0.019100	-0.149E-07	1.990000	0.278E-16
2.100000	0.119E-06	2.009000	0.000E+00

   Видно, что из-за ненулевой погрешности округления ЭВМ не может посредством селектора типа **real** во всех случаях объективно корректно выбрать нужную ветвь обработки.

Вывод: При сравнении на равенство значений вещественного типа

1. **используем только логический условный оператор;**

2. **явно указываем погрешность выполнения равенства, например, так: if (abs(y1-y2).lt.1e-5) write(*,*) 'y1=y2'**

2.5.5 Ветвящаяся развилка (ФОРТРАН: синтаксис, пример).

Удобна при записи условного оператора, когда после служебного слова **else** (в качестве тела этой альтернативы) используется новая развилка.

```
if (L1) then                  !  Как и раньше, (Li) - логические
            A1                !                       выражения;
  elseif (L2) then            !  Ai - тело i-й альтернативы.
                 A2           !  Если Li - истинно, то выполняем Ai
        elseif (L3) then      !  и управление передается на оператор,
                         A3   !  следующий за закрывающей рамкой endif.
              elseif ...      !  Если Li - ложно, то идем на оператор,
                     ...      !  следующий за ближайшим elseif.
                   else       ! Допустимо и отсутствие завершающего
                        A0    ! else (тогда все завершается одним
endif                         ! endif).
```

Рассмотрим программу анализа и решения линейного уравнения **ax**=**b**, где **a** и **b** – вводимые коэффициенты, а **x** - единственное решение, если оно существует.

```
program testlin1
implicit none
real*8 a, b, x
write(*,*) ' input a и b'   ! подсказка пользователю;
read (*,*) a, b
write(*,*) ' a=',a,' b=',b  ! контрольная печать введенного;

if (a .ne. 0d0) then
                    x=b/a
                    write(*,*) ' ед. решение x=', x
                elseif (b.eq.0d0) then
                    write(*,*) 'решений бесконечно много'
                                 else
                     write(*,*) 'решений нет'
endif
end
```

Пропуск программы и ее результаты

```
a=  3.5 b=  7.
ед. решение x=  2.

a=  0. b=  7.
решений нет
a=  0. b=  0.
решений бесконечно много
```

Убедиться, что программа действительно получает приведенные результаты
Модернизировать программу так, чтобы результат печатался с 15-значной мантиссой
Привести тексты соответствующих программ на СИ и СИ++. Провести их тестирование.

2.6 О чем узнали из первых пяти параграфов?

1. При разработке и написании программ используем только базовые алгоритмические структуры: **следование, полуразвилка, развилка, выбор, повтор с предусловием, повтор с постусловием**

2. В ФОРТРАНе и СИ **следование** реализуется последовательным расположением операторов друг за другом и не нарушается очередность их выполнения.

3. **Полуразвилка** реализуется **неполным условным оператором**.

4. В ФОРТРАНе есть два варианта неполного условного оператора:
рамочной структуры (удобной, когда выполняемое по условию действие состоит из нескольких операторов и неуклюжа, когда из одного) и **нерамочной структуры**, которая удобна в последнем случае (когда рамочная неуклюжа).

5. Операторы языка СИ не обладают рамочной структурой. Поэтому, если при истинности условия неполного условного оператора надо выполнить несколько действий, то их совокупность заключается в операторные (фигурные) скобки.

6. Условное выражение, следующее после служебного слова **if** необходимо заключать в круглые скобки (и в СИ, и в ФОРТРАНе).

7. **Рамочный неполный условный оператор** ФОРТРАНа содержит служебные слова **if** – открывающая рамка,**then** – предваряет последовательность выполняемых по условию действий; **endif** – закрывающая рамка.

8. Неполный условный оператор СИ содержит только одно служебное слово **if**.

9. Наличие погрешностей округления данных **вещественного типа** не всегда позволяет объективно установить очередность их расположения на вещественной оси (равенство или неравенство могут оказаться наведенными процессом округления). Поэтому проверку на равенство данных типа **real, real*8, float, double, long double** берем на себя, используя прием

```
if (abs(y1-y2).lt.eps) write(*,*) 'Y1 РАВНО Y2 пределах ', eps
```

   где значение **eps** задаем, исходя из условия задачи.

10. Числовые константы вещественного типа в общепринятой форме записи (без порядка) по умолчанию в СИ имеют тип **double**, а в ФОРТРАНе – **real*4** (если не изменена настройка опций компилятора)

11. Развилка в ФОРТРАНе и в СИ реализуется **полным условным оператором**, альтернативная часть которого начинается со служебного слова **else**.

12. В СИ **else** при нескольких вложенных полных условных операторов и отсутствии операторных скобок всегда относится к ближайшему **if**, для которого не найдено **else**.

13. **ФОРТРАН** позволяет закрывать все открытые условия вложенных друг в друга полных условных операторов – **ветвящуюся развилку** одним **endif**.

14. **ФОРТРАН** наряду с **логическим условным оператором** (условное выражение имеет булев тип) предоставляет и **арифметический условный оператор**, в котором условное выражение формально записывается арифметическим выражением любого допустимого в ФОРТРАНе числового типа.

15. **Избегаем в своих программах пользоваться арифметическим условным оператором**, поскольку он требует явной постановки меток, а, значит, и возможность лишней опечатки, не говоря уже о вычислительных нюансах при работе с почти равными значениями типа **real**. С некоторой натяжкой при целочисленном селекторе арифметический условный оператор можно считать примитивной моделью **оператора выбора (варианта)**, о котором узнаем в следующем параграфе.

16. В **СИ** наряду с **условным оператором** имеется **условная операция** (иногда называемая **тернарной**), состоящая из трех операндов. Первый от второго отделяется знаком вопроса **?**, а второй от третьего двоеточием **:**. При необходимости сравнения в первом операнде на точное равенство данных вещественных типов используем прием из пункта **7** (см. вывод **2** из параграфа **2.5.4**).

2.7 Второе домашнее задание

1. Придумать и записать целочисленные алгебраические выражения, которые моделируют операции математической логики (отрицание, конъюнкцию, дизъюнкцию, импликацию, эквивалентность, неравнозначность), получая целочисленную **единицу** в качестве **true** и **нуль** в качестве **false**. Например, операция отрицания моделируется формулой **f(a)=1-a**, а операция конъюнкции формулой **f(a,b)=a*b**.

2. Даны два предиката **A: x>0** и **B: y>0**. Заштриховать ту и только ту часть координатной плоскости, на которой каждая из рассмотренных в параграфе **2.3** двуместных операций математической логики над этими предикатами принимает значение **истина**.

3. Написать программу, которая вводит значения двух переменных **k** и **m** целого типа и помещает в переменную **k** наибольшее из двух введенных значений, а в переменную **m** – наименьшее.

4. Задача **3**, но без использования какой-либо разновидности условного оператора.

5. Задача аналогична предыдущей, **но переменные вещественного типа**.

6. Убедиться, что на первый взгляд программа работает верно.

7. Привести примеры таких вводимых значений переменных вещественного типа, при которых один из результатов работы предыдущей программы окажется абсолютно неверным.

8. Письменно дать ясное объяснение сути обнаруженного эффекта и его очевидное подтверждение, используя подходящие для этой цели иные числовые значения.

9. Написать программу расчета учетверенного произведения наибольшего корня квадратного уравнения $x^2 + bx + 0.25 = 0$ на коэффициент **b**.

   (a) Убедиться в правильности ее работы при некоторых пробных значениях коэффициента **b**.

   (b) Аналитически выяснить, чему должен равняться результат при $\mathbf{b} \to \infty$

   (c) Проверить правильность работы программы при $\mathbf{b} = \mathbf{10^8}, \mathbf{10^9}, \mathbf{10^{10}}$.

   (d) Письменно сформулировать свой вывод по этому поводу.

   (e) Добавить в программу численно устойчивый способ расчёта искомого произведения и продемонстрировать её работу на тех же данных.

   (f) Обеспечить простоту перевода программы на любую из допустимых разновидностей типа **real** и для каждой установить диапазон опасных значений **b**.

2.8 Оператор варианта (выбора)

— обобщение условного оператора на число альтернатив большее двух.

1. *Запись оператора варианта на ФОРТРАНе-90, -95*
2. *Пример использования селектора символьного типа*
3. *Пример использования селектора целого типа*
4. *Моделирование оператора варианта вычислимым GOTO*
5. *Запись оператора варианта в СИ (операторы switch и break)*
6. *Пример использования селектора типа unsigned char на СИ++*
7. *Пример использования селектора типа int на СИ++*
8. *О чем узнали из восьмого параграфа?*
9. *Третье домашнее задание*

2.8.1 Запись оператора варианта на ФОРТРАНе-90, -95

Ниже **select**, **case**, **default**, **endselect** – служебные слова ФОРТРАНа.
A1, A2, ..., An – тела альтернатив (блоки операторов и конструкций).

```
select case ( селектор ) ! Возможные типы селектора: integer, character*1
  case(list1);  A1       !                                        и logical.
  case(list2);  A2       ! listi - список констант; их тип должен
  case(list3);  A3       ! соответствовать типу селектора и может
    ...      ;           ! содержать: 1) значение; 2) список значений,
  case(listn);  An       ! разделенных запятыми;   3) диапазон значений,
  case deafult; A0       ! т.е. конструкцию вида  LEFT : RIGHT
endselect                ! (левая и  правая границы диапазона).
```

Наличие **;** после очередного элемента **CASE** позволяет размещать в его строке соответствующие операторы. Работа конструкции **select case**:

1. вычисляется значение селектора (иногда его называют тест-выражение);
2. при совпадении значения селектора с одной из констант списка **list1** выполняется **A1** и управление автоматически передается на оператор, следующий за **endselect**, что выгодно отличает ФОРТРАН-оператор **select case** от соответствующего СИ-оператора **switch**;
3. при несовпадении значения селектора с любой из констант списка **list1** поиск продолжается в списке **list2** и т.д;
4. если значение селектора нет ни в одном из списков, то выполняется альтернатива **case default**, а при её отсутствии — оператор, следующий за **endselect**.
5. При наличии в списке констант диапазона значений **left : right**
   (a) левая граница должна быть меньше правой **left** $<$ **right**;
   (b) отсутствие левой границы означает наличие в списке всех констант не больших правой границы;
   (c) отсутствие правой границы — наличие всех констант не меньших левой.

2.8.2 Пример использования селектора символьного типа

```
program testselect                      ! Результаты работы testselect:
implicit none                           !.......
character*1 s; character*50 t                  ! input spectral class
write(*, * ) ' input spectral class'           ! F
read (*,*) s;  write(*,*) ' spectral class=',s !   spectral class=F
select case( s )                               ! T=  7 500 K
  case('O') ;  t='T= 25 000 - 30 000 K'        !.....................
  case('B') ;  t='T= 15 000 - 25 000 K'        !  input spectral class
  case('A') ;  t='T= 11 000 K'                 ! v
  case('F') ;  t='T=  7 500 K'                 !   spectral class=v
  case('G') ;  t='T=  6 000 K'                 ! I don't know
  case('K') ;  t='T=  5 000 K'                 !      spectral class=v
  case('M') ;  t='T= 2 000 - 3 000K'
  case default;  t='I don''t know spectral class='//s
endselect
write(*,*) t
end
```

Здесь использован тип **character** для работы с данными символьного типа. Описание **character*1 s** отводит один байт памяти для хранения одного символа в переменной **s**. Переменная **t** может хранить до **50** символов.

2.8.3 Пример использования селектора целого типа

```
program testselect
implicit none
integer npos,&   ! число положительных в диапазоне [-10:10]
        nzer,&   ! --"-- нулевых --"--"--"--"--"--"--"--"--
        nneg,&   ! --"-- отрицательных --"--"--"--"--"--"--
        nout,&   ! количество чисел вне диапазона [-10:10]
        n, i, ia
npos=0; nzer=0; nneg=0; nout=0         ! Программа выясняет: сколько
write(*, * ) ' введи количество чисел' ! среди  n  введенных целых
read (*,*) n; write(*,*) ' n=',n       ! чисел, находящихся
do i=1, n                              !
  read(*,*) ia
  select case(ia)
    case(:-11,11:) ;  nout=nout+1      ! вне отрезка [-10,10],
    case(-10 : -1) ;  nneg=nneg+1      ! в пределах отрезка [-10,-1],
    case(0)        ;  nzer=nzer+1      ! равных нулю,
    case(  1 : 10) ;  npos=npos+1      ! в пределах отрезка [1,10]
  endselect
enddo
write(*,*)' nout=',nout,' nneg=',nneg,' nzer=',nzer,' npos=',npos
end
```

В этой программе встретился **оператор цикла с параметром i**:

```
do i=1,n          ! Это - заголовок оператора цикла. Значение i
  тело цикла      ! меняется автоматически от 1 до n с шагом 1, так что
enddo             ! тело цикла выполняется n раз.  Имя параметра, его
                  ! начальное и конечное значения выбирает программист.

                   // Похожая (но более универсальная) конструкция СИ
for (i=0;i<3;i++) // - оператор for. i++ - операция увеличения значения
  {  Тело         // i на 1 (аналог оператора i=i+1). Если тело цикла
     цикла;       // состоит более чем из одного действия, то его надо
  }               // заключать в операторные скобки.
```

Результат работы программы из пункта 2.8.3:

```
  введи количество чисел
5
  n= 5
  -3
  66
  2
  9
  1
    nout= 1 nneg= 1 nzer= 0 npos= 3
```

2.8.4 Моделирование оператора варианта вычислимым GOTO

```
      program testselect
      implicit none
      integer npos, nzer, nneg, nout, n, i, ia,
     >        key   ! ключ-селектор вычислимого GOTO
      npos=0; nzer=0; nneg=0; nout=0
      write(*, * ) ' введи количество чисел' ; read (*,*) n
      write(*,*) ' n=',n
      do i=1, n
        read(*,*) ia; if ((ia.le.11) .or. (ia.ge.11)) key=2
                      if ((ia.ge.-10).and.(ia.le.10)) then
           key=ia
           if (key.ne.0) key=key/iabs(key)
        endif
        goto (10,20,30,40), key+2
   10   continue ; nneg=nneg+1 ; goto 77
   20   continue ; nzer=nzer+1 ; goto 77
   30   continue ; npos=npos+1 ; goto 77
   40   continue ; nout=nout+1
   77   continue
      enddo
      write(*,*)' nout=',nout,' nneg=',nneg,' nzer=',nzer,' npos=',npos
      end
```

2.8.5 Запись оператора варианта в СИ (операторы switch и break)

```
switch ( switch-выражение ) //  Тип селектора: один из целых (char, int,
            селектор        //  unsigned int, long int, long unsigned)
{
  case list1 : A1; [ break; ] // listi - список констант или константных
  case list2 : A2; [ break; ] //         выражений
  case list3 : A3; [ break; ]
     ...     : ...    ...
  case listn : An; [ break; ]
  [ default  : A0;]
}
```

Как и в ФОРТРАНе, при совпадении значения селектора с одной из констант списка **listi** выполняется шаг **Ai**. Однако, если после **Ai** не указан оператор **break**, предназначенный для передачи управления на оператор, следующий за **switch**, то будут выполняться операторы следующих за **listi** ветвей: A_{i+1}, A_{i+2} и т.д., пока не встретится **break** или **A0**. Если среди всех **listi** не найден равный значению селектора, то при наличии **default** выполняется **A0**. Если же **default** нет, то выполняется оператор, написанный после операторной скобки, закрывающей тело **switch**.

Оператор break – обеспечивает прекращение выполнения самого внутреннего из объемлющих его операторов **switch**, **do**, **for**, **while**. При выполнении оператора **break** управление передается оператору следующему за прерванным.

2.8.6 Пример использования селектора типа char на СИ++

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{ char f;   char t[50];
  cout<< " input spectral class"<<endl;        // подсказка пользователю;
  cin >>f; cout<<" spectral class="<<f<<endl;  // ввод и контрольная печать;
  switch (f)
    { case'O': strcpy(t,"T= 25 000 - 30 000 K"); break;
      case'B': strcpy(t,"T= 15 000 - 25 000 K"); break;
      case'A': strcpy(t,"T= 11 000 K"         ); break;
      case'F': strcpy(t,"T=  7 500 K"         ); break;
      case'G': strcpy(t,"T=  6 000 K"         ); break;
      case'K': strcpy(t,"T=  5 000 K"         ); break;
      case'M': strcpy(t,"T= 2 000 - 3 000K"   ); break;
      default: strcpy(t,"T=I don't know spectral class="); strncat(t,&f,1);
    }
  cout<<t<<endl; return 0;
}
```

Замечания:

1. Здесь в качестве селектора использовано значение типа **char**, трактуемое в СИ как однобайтовое значение целого типа.

2. В СИ любой целый тип может быть в двух формах: *данное со знаком* или *данное без знака* (описатели **signed** и **unsigned** соответственно). Первая трактует самый левый бит байта как знак числа (0 – плюс, 1 – минус), а вторая – как двоичный старший разряд целого положительного числа. Именно поэтому, максимальное целое типа **unsigned char** больше соответствующего значения типа **signed char** на $\mathbf{2^7}$ (сравни **255** и **127**; $\mathbf{255 = 127 + 128 = 127 + 2^7}$). Тип **unsigned char** удобно использовать для представления кодов литер клавиатуры, а константы **unsigned char** записывать символом, заключенным в одиночные кавычки (апострофы).

   Компиляторы **gcc** и **g++** толкуют описатель **char** по умолчанию согласно настройке опции компилятора **-fsigned-char** или **-funsigned-char**.

```
#include <stdio.h>                  // Данная программа из файла tstchr.c
int main()                          // в зависимости от указанных при вызове
{ char c;                           // компилятора опций выводит по форматам
  c=-5;                             // %i и %d  либо 251, либо -5.
  printf(" %c  %d  %i \n",c,c,c);
return 0; }
```

Команда вызова компилятора	Результат
gcc tstchr.c	**Ш -5 -5**
gcc -fsigned-char	**Ш -5 -5**
gcc -funsigned-char	**Ш 251 251**
g++ tstchr.cpp	**c=Ш -5**
gcc -fsigned-char	**c=Ш -5**
g++ -funsigned-char tstchr.cpp	tstchr.cpp: In function ‘int main()’: tstchr.cpp:5: warning: converting of negative value ‘-0x000000005’ to ‘char’
./a.out	c=Ш 251

   Во избежание зависимости от настройки режима умолчания компилятора рекомендуется в СИ-программах явно уточнять описание типа **char** (например, **unsigned char** или **signed char**).

   Наряду с элементарным типом данных **char** в программе впервые встретилось **char t[50]** – описание **массива**. Признаком описания массива в СИ служит наличие после имени массива парных квадратных скобок, в которых часто пишется количество элементов массива.

3. **Массив** – именованный набор конечного количества элементов одинакового типа. Все его элементы расположены в оперативной памяти непосредственно друг за другом. Доступ к отдельному элементу массива осуществляется по имени массива и порядковому номеру элемента. Часто порядковый номер называют **индексом**. В СИ **индекс** начального элемента массива всегда равен **нулю**; в ФОРТРАНе – по умолчанию **единице**. ФОРТРАН при описании массива использует круглые скобки. Число внутри скобок в операторе описания массива задает количество его элементов. Ограничимся пока простейшим случаем работы с массивом, задавая количество элементов в нем константой.

4. В альтернативе умолчания **default** к строке **t** посредством функции **strncat** добавляется значение ошибочно заданного селектора с целью информировать пользователя о нажатии им не той клавиши. Функция **strncat** в качестве первых двух аргументов требует указателей на строку, к которой добавляется подсоединяемая строка, и на саму подсоединяемую строку. Третий аргумент – количество подсоединяемых символов. Краткое описание функций **strcpy** и **strncat** можно увидеть посредством команд **man** и/или **info**, вызванных с соответствующим аргументом.

   Строка **t** объявлена массивом. Имя массива в СИ является **указателем**. Так что с первым аргументом **t** у **strncat** проблем не будет. В качестве второго аргумента имя **f** не подходит, поскольку функция **strncat** требует в качестве типа второго аргумента так же **указатель**, то есть адрес ячейки, в которой расположена переменная **f**. Поэтому при обращении к **strncat** при задании второго аргумента использована операция **&** указания адреса, по которому расположено значение переменной **f**.

   Обратим внимание, что в СИ при обращении к функции вовсе не обязательно явно передавать вырабатываемое ею значение какой-нибудь переменной или функции, как это необходимо в ФОРТРАНе. Например, обращение к функции **strncat**, если не надо запоминать возвращаемое через ее имя значение, может быть записано просто как **strncat(t,&f,1)**.

2.8.7 Пример использования селектора типа int на СИ++

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{ int n, npos, nzer, nneg, nout, i, ia;
  cout<<" введи количество чисел:"<<endl;  // подсказка пользователю;
  cin >>n;   cout<<" n="<<n<<endl;         // ввод и контрольная печать;
  npos=nzer=nneg=nout=0;                   // инициализация счетчиков;
  for (i=1;i<=n;i++)
    { cin>>ia;
      switch (ia)
        { case -10:case -9:case -8: case -7:case -6:
  case  -5:case -4:case -3: case -2:case -1: nneg+=1; break;
          case   0: nzer+=1; break;
          case  10:case 9:case 8: case 7:case 6:
  case   5:case 4:case 3: case 2:case 1: npos+=1; break;
          default : nout+=1;
        }
    }
  cout<<"nneg="<<nneg<<"  nzer="<<nzer
      <<"  npos="<<npos<<"  nout="<<nout<<endl;
  return 0;
}
```

2.9 О чем узнали из восьмого параграфа?

1. Синтаксис записи ФОРТРАН-оператора **select case**;
2. В качестве селектора возможны значения символьного, целого и булева типов;
3. **character*1** – ФОРТРАН-описатель символьного типа (в ФОРТРАНе-95 возможно и **character(1)**).
4. Описатель **char** в СИ соответствует однобайтовому значению целого типа.
5. О СИ-описателях **unsigned** и **signed**.
6. О возможности перенастройки режима работы программы с данными типа **char** посредством задания опций компилятора.
7. О возможности задания диапазона значений селектора для реализации одной ветви оператора варианта ФОРТРАНа.
8. О ФОРТРАН-операторе вычислимом **goto** (модель оператора варианта).
9. Синтаксис и назначение СИ-оператора **switch**.
10. Назначение СИ-оператора **break**.
11. Назначение функций **strcpy** и **strncat** из **string.h**.
12. Пример использования операции **&** при задании второго фактического аргумента при обращении к **strncat**.
13. О наличии и в СИ, и в ФОРТРАНе структуры данных **массив**, которая сопоставляет одному имени конечный набор элементов одинакового типа.
14. Все элементы массива располагаются в оперативной памяти непосредственно друг за другом.
15. При описании массива в СИ используются парные квадратные скобки.

    В ФОРТРАНе — парные круглые.
16. Целое число, заключенное между парными скобками в **описании массива**, задает количество его элементов.
17. Целое число или переменная целого типа, заключенная между парными скобками **при обращении к элементу массива**, задает номер элемента;
18. Номер элемента массива часто называют **индексом**.
19. В СИ индекс начального элемента массива всегда равен **нулю**. В ФОРТРАНе, если не предпринято явное изменение индекса начального элемента, – **единице**.
20. Слегка коснулись **операторов цикла** типа арифметической прогрессии (подробно об операторах цикла узнаем из параграфа **2.11**).

2.10 Третье домашнее задание

1. Написать программу, которая печатает **наименование** младшей цифры введенного целого десятичного числа.

   В СИ есть операция **%** – получение остатка от целочисленного деления операндов (например, **15 % 4 = 3**). В ФОРТРАНе аналогичной операции нет, но зато есть встроенная функция **mod(n,k)** (т.е. **mod(15,4)=3**). При нежелании пользоваться встроенной функцией **mod** остаток от деления нацело двух чисел можно найти по формуле **n - k * (n/k)**, но не n - k * n/k (последнее выражение принципиально отлично от предыдущего; *почему?*).

2. Написать программу, выводящую **наименование** младшей цифры введенного целого десятичного числа после перевода его в **Z**-ичную систему счисления.

3. Написать программу, которая по значению селектора, равному **C**, **G**, **M**, **S**, **d**, **h**, **m**, **s** переводит величину угла, заданного в радианной мере, в доли оборотов; градусы , минуты, секунды (угловой меры); доли суток; часы, минуты, секунды (часовой меры) **соответственно**.

4. Написать программу, которая вырабатывает ключ принадлежности точки части плоскости, согласно правилу:

Ключ	Смысловая нагрузка
0	началу координат;
1	правой полуоси абсцисс;
-1	левой полуоси абсцисс;
2	верхней полуоси ординат;
-2	нижней полуоси ординат;
3	первой четверти;
-3	третьей четверти;
4	второй четверти;
-4	четвертой четверти.

5. Известно, что светило восходит и заходит на широте φ, если абсолютное значение его склонения: $|\delta| < (\mathbf{90}^\circ - |\varphi|)$ иначе светило будет *незаходящим* или *невосходящим*. Написать программу, которая по введенным φ широте места и склонению δ светила определяет его тип: *незаходящее*, *невосходящее* или *восходящее и заходящее*.

6. Доказать, что любую целочисленную сумму **S** большую семи всегда можно разменять трешками и пятерками.

   Написать программу, которая по введенной сумме **S>7** находит вариант размена ее трешками и пятерками с наибольшим количеством трешек.

7. Задача совпадает с предыдущей, НО программа должна быть написана без использования какой-либо разновидности условного оператора или оператора варианта.

2.11 Операторы цикла

Операторы цикла предназначены для записи средствами языков программирования базовой алгоритмической структуры **повтор**. В **ФОРТРАНе** и **СИ** есть несколько вариантов записи циклических процессов в соответствие с рассмотренными ранее видами повтора (**повтор с предусловием**, **повтор с постусловием**), хотя в принципе всегда можно обойтись и одним из них (удобно ли только?). Прежде чем сформулировать правило выбора наиболее выгодной для решения задачи формы оператора цикла ознакомимся с их синтаксисом.

1. Оператор цикла с предусловием (ФОРТРАН)
2. Пример использования оператора цикла с предусловием
3. Осмысление результата
4. Оператор цикла с предусловием (СИ)
5. Оператор цикла с постусловием (ФОРТРАН, СИ)
6. Примеры использования оператора цикла с постусловием
7. ФОРТРАН-77, 90: Моделирование повтора с постусловием
8. СИ, C++: Моделирование повтора с постусловием
9. Оператор цикла с параметром
10. Примеры использования оператора цикла с параметром

2.11.1 Оператор цикла с предусловием (ФОРТРАН)

```
do while ( L )      !  Здесь L - логическое выражение,
                    !  задающее условие продолжения цикла;
     T              !  T - тело цикла (набор операторов,
                    !      которые надо повторять).
enddo
```

О работе конструкции **do while**:

1. вычисляется значение логического выражения **L**;

2. Пока значение **L** – **истина** (**.true.**) выполняется тело цикла.

3. Повтор прекращается только при **L=.false.**. Ясно, что значение **L**, обеспечивающее возможность первого выполнения тела цикла, должно быть подготовлено операторами, предшествующими открывающей рамке **do while**.

4. При начальном **L=.false.** тело цикла не выполнится ни разу.

5. Для обеспечения возможности прекращения повтора необходимо, чтобы значение **L** зависело определенным образом от величин, вычисляемых внутри тела цикла, так как только в этом случае возможно изменение значения **L** с **.true.** на **.false.**. При отсутствии изменения **L** программа **зациклится**. Таким образом, важно правильно записывать логическое выражение **L** и формулы расчета величин в теле цикла, влияющих на **L**.

В старых версиях ФОРТРАНа оператора цикла с предусловием не было. Его приходилось моделировать сочетанием условного оператора и оператора **goto**:

```
   L=.true.            ! Здесь  L - булева переменная, значение
10 continue            ! которой должно перевычисляться внутри
    if (not L) goto 20 ! тела T. В качестве L можно использовать
       T               ! и операцию отношения с операндами,
    goto 10            ! зависящими от вычисляемых в T величин.
20 continue
```

Такое моделирование возможно и в новых версиях ФОРТРАНа, хотя объективно уже не оправдано, вынуждая явно помечать операторы, на которые следует передавать управление, и предоставляя возможность совершить нелепую ошибку, передав управление не туда, куда хотели, что при использовании **do while** невозможно в принципе, т.к. нет ни меток, ни явных передач управления.

2.11.2 Пример использования оператора цикла с предусловием

В пункте **1.4** была сформулирована задача:
используя четыре переменные вещественного типа **x, y, z, t**, для значения аргумента **x=10** вычислить величины **y, z, t** по формулам **y=1+x; z=y-1; t=z/x**. Текст программы, решающей эту задачу, из пункта **1.4.4**:

```
program first1
implicit none
real(4) x, y, z, t
x=10;  y=1+x;  z=y-1;  t=z/x; write(*,*) x,y,z,t
end
```

В пункте **1.6.6** обсуждались причины, по которым результаты расчета величин **z** и **t** при очень малых значениях аргумента **x** оказывались абсолютно неверными. Включив правильно в текст программы оператор цикла **do while**, получим программу, которая выводит на экран таблицу, наглядно демонстрирующую ухудшение точности упомянутых **z** и **t** с уменьшением **x**. Оформим расчёт $\mathbf{x = 10^{-k}}$, $\mathbf{k = 0, 1, 2, ...10}$, вычисляя для каждого **k** соответствующие **x**, **y**, **z** и **t**.

Хотя в ФОРТРАНе и есть операция возведения в степень ******, сейчас использовать её для расчета по формуле **x=10.0**(-k)** невыгодно, так как будет вычисляться выражение $x = exp(-k \ln 10.0)$, вызывающее встроенные функции **exp** и **alog**, время и погрешность работы которых гораздо больше тех же характеристик одной арифметической операции (деления), которой можно обойтись. Присвоим до заголовка цикла аргументу **x** начальное значение **10**, а внутри цикла при каждом очередном **k** будем уменьшать текущее значение **x** в десять раз.

Перед открывающей рамкой цикла **do while** поместим печать заголовка таблицы по оператору **format**, помеченному меткой **1000**. Важно понимать, что *ругань* меток, звучавшая ранее, относится к меткам передачи управления, а не к меткам оператора **format**. Метки управления понижают наглядность текста, а метки оператора **format**, при удачной структуре программы, как правило, повышают.

Внедрение длинных форматных строк в оператор **write** ухудшает восприятие текста в целом, заостряя внимание на форме вывода, которая затеняет список печатаемых переменных — **за деревьями леса не видно**. Короткий же формат, внедренный в оператор **write**, объективно выгоден — меньше число строк при сохранении наглядности. Реализуем расчет так:

```
      program dowhile
      implicit none
      real*4 x, y, z, t
      integer k
      x=10.0; k=0;  write(*,1000)
      do while (k>=-10)
        x=x/10; y=1+x; z=y-1; t=z/x; write(*,1001) k,x,y,z,t
        k=k-1
      enddo
 1000 format(1x,1x,'#k',8x,'x',12x,'y=1+x',10x,'z=y-1',8x,'t=z/x=1')
 1001 format(1x,i3,4e15.7)
      end
```

2.11.3 Осмысление результата

```
 #k        x             y=1+x          z=y-1         t=z/x=1
  0  0.1000000E+01  0.2000000E+01  0.1000000E+01  0.1000000E+01
 -1  0.1000000E+00  0.1100000E+01  0.1000000E+00  0.1000000E+01
 -2  0.1000000E-01  0.1010000E+01  0.9999990E-02  0.9999990E+00
 -3  0.9999999E-03  0.1001000E+01  0.1000047E-02  0.1000047E+01
 -4  0.9999999E-04  0.1000100E+01  0.1000166E-03  0.1000166E+01
 -5  0.9999999E-05  0.1000010E+01  0.1001358E-04  0.1001358E+01
 -6  0.9999999E-06  0.1000001E+01  0.9536743E-06  0.9536744E+00
 -7  0.9999999E-07  0.1000000E+01  0.1192093E-06  0.1192093E+01
 -8  0.9999999E-08  0.1000000E+01  0.0000000E+00  0.0000000E+00
 -9  0.9999999E-09  0.1000000E+01  0.0000000E+00  0.0000000E+00
```

Содержимое столбца **t**, в каждой строчке которого должна стоять **единица** – очевидный верный ответ, наглядно демонстрирует по мере уменьшения **x** все большее и большее ухудшение точности результата. В последних трех строках вообще получен вместо единицы **нуль**, а при **k=-7** результат получен лишь с точностью до двух единиц второй значащей цифры (это разряд десятых), несмотря на то, что расчет велся с **7-8**-значной мантиссой.

Замечания:

1. На самом деле, именно для этой задачи оператор **do while** не так выгоден, как разбираемый далее оператор цикла с параметром. Здесь же просто учимся применять **do while**.

2. Результат в виде таблицы можно вывести не только на экран, но и в файл. Для этого нет нужды изменять текст программы – достаточно перенаправить поток стандартного вывода в файл текущей директории (см.параграф **1.3.3**): **./whiledo > whiledo.res**.

3. Наглядное графическое представление количества верных цифр просто получить, используя утилиту **gnuplot**. По оси абсцисс отложим $\mathbf{lg(x)}$, выбирая для наглядности равномерную шкалу по показателю степени десятки, а по оси ординат $\mathbf{1 - lg(9 \cdot |t - 1|/1)|}$ — грубая оценка количества верных цифр по относительной погрешности $\mathbf{|t - 1|}$ (**gnuplot** по умолчанию игнорирует точки, отражающие **бесконечность**, не нанося их, что удобно).

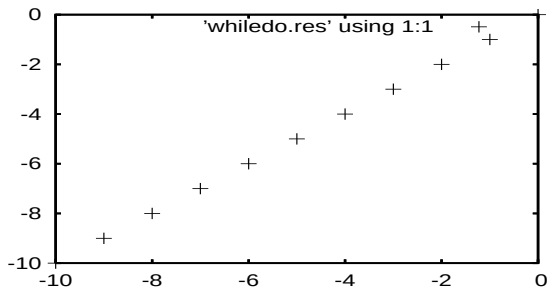
4. После вызова утилиты **gnuplot** на экране появится её заставка, информирующая о версии, авторах и т.д. Последняя строка заставки сообщает режим используемого терминала, после чего следует промпт утилиты, приглашающий к набору ее команд.

```
$ gnuplot
        G N U P L O T
        Version 4.0 patchlevel 0 ... ... ...
Terminal type set to 'x11'
gnuplot>
```

Напишем после промпта **gnuplot**> команду **plot** черечения графика

```
gnuplot> plot 'whiledo.res' using 1:1
gnuplot>
```

подав ей в качестве аргумента заключённое в апострофы имя файла, хранящего результаты работы нашей программы. После имени файла опция **using** информирует команду **plot** о номерах столбцов, хранящих значения абсциссы и ординаты. Вторая единица – не опечатка. Просто, приобретая навык работы с утилитой, хотим проверить, что точки графика разместятся по диагонали. Теперь нажимаем клавишу **enter** и

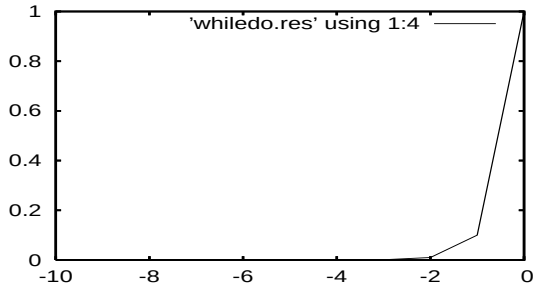


— график построен. Правда, построен он точками (красными значками *плюс*).

Нажав клавишу перевода курсора вверх (предварительно активируя указателем *мыши* экран с командной строкой **gnuplot**а), можем получить ранее данную **gnuplot**-команду, в которой можно поменять номер столбца ординат.

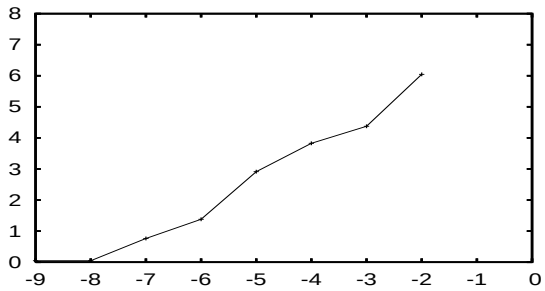
Для изображения графика непрерывной линией добавим в команде **plot** опцию **with** с модификатором **l**:

```
plot 'whiledo.res' using 1:4 with l
```



5. Начертим график, отражающий зависимость числа верных цифр столбца **5** от значения **x**. Вместо цифры **5** в содержимом опции **using** пишем формулу расчета нужной величины.

```
plot 'whiledo.res' using 1:(1-log10(9*abs($5-1))) with l
```



Значок доллара перед номером столбца (цифрой пять) означает, что десятичный логарифм надо брать не от числа *пять*, а от данных, находящихся в пятом столбце.

6. Наконец, заметим, что в файле **whiledo.res** заголовок таблицы начинается со значка *диеза*. Это неспроста – на языке утилиты **gnuplot** строка данных, начинающаяся с *диеза*, игнорируется, то есть считается комментарием. Вывод заголовка таблицы в файл результата явно напоминает нам о содержании столбца, а значок *диеза* не позволяет утилите воспринимать заголовок в качестве данных для графика.

7. Последний график демонстрирует, как по мере уменьшения аргумента уменьшается количество верных цифр в значениях **t** (и **z**). Если расчет ведется с одинарной точностью, то есть при восьмизначной десятичной мантиссе, то при $\mathbf{x = 10^{-7}}$ все цифры результата кроме самой старшей неверны, а при $\mathbf{x = 10^{-8}}$ уже потеряны все значащие цифры и порядок.

2.11.4 Оператор цикла с предусловием (СИ)

```
                    // Пока L не равно нулю, то есть
    while ( L )  T; // пока значение L - ИСТИНА выполняй
                    // тело T
```

Замечания:

1. Как и в ФОРТРАНе **L** – это условие продолжения цикла.
2. Если значение **L** в начальный момент работы заголовка цикла окажется равным **нулю**, то тело цикла не выполнится ни разу.
3. Если тело оператора цикла состоит более чем из одного оператора, то все их необходимо заключить в операторные скобки **{ }** (СИ-операторы цикла не обладают рамочной структурой, как операторы цикла ФОРТРАНа-95).

Текст предыдущей программы на СИ

```
#include <stdio.h>
int main()
{ float x, y, z, t;
 int k=0; x=10.0;
  printf(" %2c %9c %17s %15s %15s\n",'k','x',"y=1+x","z=y-1","t=z/x=1");
  while (k>=-8)
   { x=x/10; y=1+x; z=y-1; t=z/x;
     printf("%3i %15.7e %15.7e %15.7e %15.7e\n", k,x,y,z,t); k-=1;
   }
}
```

Вопрос: Как добиться в СИ того же эффекта, который дает повторитель ФОРТРАНа 8x? Писать восемь пробелов неудобно.

Ответ : В списке вывода указываем выводимые строки (“k”, “x”, “y=1+x”, “z=y-1”, “t=z/x=1”), сопоставляя им спецификацию формата **%s** с указанием нужной ширины поля вывода.

Результаты работы СИ-программы

k	x	y=1+x	z=y-1	t=z/x=1
0	1.0000000e+00	2.0000000e+00	1.0000000e+00	1.0000000e+00
-1	1.0000000e-01	1.1000000e+00	1.0000002e-01	1.0000002e+00
-2	9.9999998e-03	1.0100000e+00	9.9999905e-03	9.9999905e-01
-3	9.9999993e-04	1.0010000e+00	1.0000467e-03	1.0000468e+00
-4	9.9999990e-05	1.0001000e+00	1.0001659e-04	1.0001661e+00
-5	9.9999988e-06	1.0000100e+00	1.0013580e-05	1.0013582e+00
-6	9.9999988e-07	1.0000010e+00	9.5367432e-07	9.5367444e-01
-7	9.9999987e-08	1.0000001e+00	1.1920929e-07	1.1920930e+00
-8	9.9999991e-09	1.0000000e+00	0.0000000e+00	0.0000000e+00

Как видим, и СИ-, и ФОРТРАН-программы дают при малых значениях аргумента одинаково неверный результат.

2.11.5 Оператор цикла с постусловием (ФОРТРАН, СИ)

В старых версиях ФОРТРАНа не было специального оператора цикла с постусловием (как впрочем и с предусловием). Их всегда можно было сконструировать посредством оператора продолжения **continue**, условного оператора **if** и оператора передачи управления **goto**. Приведем типичные для старого ФОРТРАНа модели оператора цикла с постусловием:

```
                              ! Ф О Р Т Р А Н:
    10 continue               !  Точка приема передачи управления;
                              !
           T                  !  T - тело цикла (набор операторов,
                              !      которые надо повторять);
       if (.not. L)  goto 10  !  L - логическое выражение, определяющее
                              !      критерий ПРЕКРАЩЕНИЯ повтора.
```

или

```
    10 continue               ! Ф О Р Т Р А Н
                              !
           T                  !  Здесь L определяет критерий
                              !
       if (L)  goto 10        !  ПРОДОЛЖЕНИЯ повтора.
```

Уяснение ситуации

1. Выбирает модель оператора цикла с постусловием программист, исходя из своего опыта и постановки задачи.

2. При моделировании оператора цикла с постусловием последнее часто удобно рассматривать как условие **продолжения** цикла. В языке же программирования ПАСКАЛЬ с оператором цикла **repeat T until L** постусловие трактуется как условие **прекращения**. Поэтому, при переводе программы с ПАСКАЛя на ФОРТРАН перед паскалевским условием **L** можно ставить операцию отрицания **.not.** с тем, чтобы внешний вид **L** формально совпадал в обоих языках. Тем самым гарантируется правильность моделирования при переводе.

3. В СИ имеется оператор цикла с постусловием:

   ```
   do       // Цикл повторяется до тех пор, пока логическое
     T      // выражение L не примет значение 0 (то есть ложь).
   while L; // Другими словами в СИ-операторе цикла с постусловием
            // L - это критерий продолжения повтора:
            // цикл продолжается, пока L истинно.
   ```

   Безусловно, при желании, поставив перед условием **L** операцию отрицания **!**, получим паскалеобразную модель оператора цикла с постусловием, когда значение **истина** выражения **L**, будет служить критерием **прекращения** повтора.

2.11.6 Примеры использования оператора цикла с постусловием

ФОРТРАН: L - критерий прекращения Реализуем на ФОРТРАНе уже знакомую из пункта программу, используя оператор цикла с постусловием, когда **L** условие прекращения повтора (как в ПАСКАЛе):

```
      program repeat1
      implicit none
      real*4 x, y, z, t
      integer k
      x=10.0; k=0;  write(*,1000)
   10 continue
        x=x/10; y=1+x; z=y-1; t=z/x; write(*,1001) k,x,y,z,t
        k=k-1
      if (.not. (k.eq.-11) ) goto 10
 1000 format(1x,2x,'k',8x,'x',12x,'y=1+x',10x,'z=y-1',8x,'t=z/x=1')
 1001 format(1x,i3,4e15.7)
      end
```

ФОРТРАН: L - критерий продолжения

```
      program repeat1
      implicit none
      real*4 x, y, z, t
      integer k
      x=10.0; k=0;  write(*,1000)
   10 continue
        x=x/10; y=1+x; z=y-1; t=z/x; write(*,1001) k,x,y,z,t
        k=k-1
      if (k.ge.-10) goto 10
 1000 format(1x,2x,'k',8x,'x',12x,'y=1+x',10x,'z=y-1',8x,'t=z/x=1')
 1001 format(1x,i3,4e15.7)
      end
```

СИ: L - критерий прекращения

```
#include <stdio.h>
int main()
{ float x, y, z, t;
 int k=0; x=10.0;
  printf(" %2c %9c %17s %15s %15s\n",'k','x',"y=1+x","z=y-1","t=z/x=1");
  do
    {x=x/10;  y=1+x; z=y-1; t=z/x;
     printf("%3i %15.7e %15.7e %15.7e %15.7e\n", k,x,y,z,t);  k-=1;
    }
  while(k>=-11);
}
```

СИ: L - критерий продолжения

```
#(1(1include <stdio.h>
int main()
{ float x, y, z, t;
 int k=0; x=10.0;
  printf(" %2c %9c %17s %15s %15s\n",'k','x',"y=1+x","z=y-1","t=z/x=1");
  do
    {x=x/10;  y=1+x; z=y-1; t=z/x;
     printf("%3i %15.7e %15.7e %15.7e %15.7e\n", k,x,y,z,t);  k-=1;
    }
  while(!(k==-11));   // явно выделен критерий прекращения;
}
```

Замечание. Выводимый заголовок таблицы состоит из символов (**k** и **x**) и строковых констант (**y=1+x**, **z=1-x**, **t=z/x=1**). Спецификация формата для символов обозначена символом %c, а для строк — %s. При использовании %s вместо %c для вывода символа, заключённого в апострофы, компиляция завершится успешно, а пропуск – аварийно, сообщением: **Segmentation fault**.

2.11.7 ФОРТРАН-77, 90: Моделирование повтора с постусловием

При организации повторов с предусловием (с постусловием) в старых версиях ФОРТРАНа необходимо было явно указывать метку (перед оператором **continue** и после **goto**), что чревато возможной ошибкой (не то пометить, не туда передать).

Современный ФОРТРАН предоставляет возможность конструирования оператора цикла с постусловием (и с предусловием) **без постановки меток**, за счет наличия в нем оператора бесконечного повтора и оператора выхода из него **exit** на оператор, следующий непосредственно после оператора цикла:

```
do             ! Если бы тело этого оператора цикла не содержало
  T            ! расчета логического условия L прекращения цикла,
  if (L) exit  ! и оператора  exit (выхода из цикла), то получили бы
enddo          ! "бесконечно циклящуюся" алгоритмическую структуру.
```

С учётом изложенного первую программу из пункта **2.11.6** можно переписать так:

```
      program testexit
      implicit none
      real*4 x, y, z, t
      integer k
      x=10.0; k=0;  write(*,1000)
      do
        x=x/10; y=1+x; z=y-1; t=z/x; write(*,1001) k,x,y,z,t
        k=k-1
        if (k.eq.-10) exit
      enddo
 1000 format(1x,2x,'k',8x,'x',12x,'y=1+x',10x,'z=y-1',8x,'t=z/x=1')
 1001 format(1x,i3,4e15.7)
      end
```

2.11.8 СИ, C++: Моделирование повтора с постусловием

СИ позволяет организовать выход из **бесконечного** цикла посредством оператора **break**, который ранее (см. пункт **2.8.5**) применялся для завершения соответствующей альтернативы оператора **switch**. Например,

```
#include <stdio.h>
int main()
{ float x, y, z, t;
 int k=0; x=10.0;
  printf(" %2c %9c %17s %15s %15s\n",'k','x',"y=1+x","z=y-1","t=z/x=1");
  do
    {
     x=x/10;  y=1+x; z=y-1; t=z/x;
     printf("%3i %15.7e %15.7e %15.7e %15.7e\n", k,x,y,z,t);  k-=1;
     if (k==-10) break;
    }
  while(1<2));
}
```

2.11.9 ФОРТРАН-оператор цикла с параметром

Оператор цикла с параметром часто называют оператором цикла типа пересчета или типа арифметической прогрессии. **Параметр цикла** – переменная, которая согласно заголовку цикла автоматически принимает значения от начального до конечного при указанном приращении (т.е. изменяется с шагом равным разности арифметической прогрессии). Тип параметра определяется правилами языка программирования (целый, символьный, даже вещественный). Старый ФОРТРАН не допускал в качестве параметра цикла переменную **вещественного** типа. И это – правильно! (*Почему?*)

Оператор цикла с параметром состоит из заголовка и тела. Именно заголовок задает начальную настройку и правило изменения параметра цикла:

```
do i=i0,ik,ih ! Заголовок цикла с параметром i:
  Тело        ! i0 - начальное значение i;
enddo         ! ik - конечное;  ih - приращение параметра
```

Смысловая нагрузка: Для значения **i**, изменяющегося от значения **i0** с шагом **ih** вплоть до значения **ik** включительно, повторять выполнение **тела**. При **i0>ik** и **ih>0** тело цикла не выполняется, т.е. оператор цикла с параметром – частный случай оператора цикла с предусловием.

Достоинство оператора цикла с параметром по сравнению с последним в том, что в теле цикла **НЕ НАДО** явно применять оператор изменения параметра цикла. Параметр цикла пересчитывается автоматически в соответствии с заголовком по правилу арифметической прогрессии.

В старом ФОРТРАНе оператор цикла с параметром **всегда** выполнял тело цикла один раз (даже если начальное значение параметра цикла оказывалось больше конечного). Исполнимый код, полученный по умолчанию современными ФОРТРАН-компиляторами, не выполнит подобные циклы ни разу.

2.11.10 Примеры использования оператора цикла с параметром

Задача та же

```
      program for1
      implicit none
      real*4 x, y, z, t
      integer k
      x=10.0; k=0;  write(*,1000)
      do k=0,-10,-1
        x=x/10; y=1+x; z=y-1; t=z/x; write(*,1001) k,x,y,z,t
      enddo
 1000 format(1x,2x,'k',8x,'x',12x,'y=1+x',10x,'z=y-1',8x,'t=z/x=1')
 1001 format(1x,i3,4e15.7)
      end
```

или

```
      program for1
      implicit none
      real*4 x, y, z, t
      integer k
      x=10.0; k=0;  write(*,1000)
      do k=0,10
        x=x/10; y=1+x; z=y-1; t=z/x; write(*,1001) k,x,y,z,t
      enddo
 1000 format(1x,2x,'k',8x,'x',12x,'y=1+x',10x,'z=y-1',8x,'t=z/x=1')
 1001 format(1x,i3,4e15.7)
      end
```

В последнем варианте приращение по умолчанию полагается равным единице.

Синтаксис оператора цикла с параметром в СИ

```
for (начальное выражение;
            логическое выражение продолжения повтора;
                   выражение для приращения)
  {
     Тело цикла;
  }
```

Этот оператор эквивалентен следующему оператору **while**:

```
while ( логическое выражение продолжения повтора )
 {
   Тело цикла;
   изменение значения параметра согласно выражению для приращения;
 }
```

Примеры использования оператора FOR в СИ-программах.

```
#include <stdio.h>
int main()
{ float x, y, z, t;  int k=0; x=10.0;
  printf(" %2c %9c %17s %15s %15s\n",'k','x',"y=1+x","z=y-1","t=z/x=1");
  for (k=0;k>=-10;k--)                    // параметр цикла уменьшается
    { x=x/10;  y=1+x; z=y-1; t=z/x;
     printf("%3i %15.7e %15.7e %15.7e %15.7e\n", k,x,y,z,t);
    }
}
```

```
#include <stdio.h>
int main()
{ float x, y, z, t; int k=0; x=10.0;
  printf(" %2c %9c %17s %15s %15s\n",'k','x',"y=1+x","z=y-1","t=z/x=1");
  for (k=0;k<=10;k++)                    // параметр цикла увеличивается
    { x=x/10;  y=1+x; z=y-1; t=z/x;
     printf("%3i %15.7e %15.7e %15.7e %15.7e\n", k,x,y,z,t);
    }
}
```

Изменим тип параметра на вещественный и проанализируем результат.

```
#include <stdio.h>
int main()
{ float x=10.0, y, z, t,k;
  printf(" %2c %9c %17s %15s %15s\n",'k','x',"y=1+x","z=y-1","t=z/x=1");
  for (k=0.0;k<=10.0;k++)                // параметр цикла вещественный
    { x=x/10;  y=1+x; z=y-1; t=z/x;
      printf("%3i %15.7e %15.7e %15.7e %15.7e\n", k,x,y,z,t);
    }
}
```

```
  k         x             y=1+x           z=y-1         t=z/x=1
  0   0.0000000e+00  5.2998088e-315  5.3049895e-315  5.2998088e-315
  0 -1.4916685e-154 -1.4916685e-154  5.3003269e-315   2.0000005e+00
  0   2.0000005e+00  -2.0000005e+00  5.2998606e-315  5.2649948e-315
  0  -2.0000005e+00 -1.4916685e-154  5.2998140e-315  1.4916685e-154
  0  -2.0000005e+00 -2.6815622e+154  5.2998093e-315  1.4916685e-154
  0  2.6815622e+154 -5.2133474e-315  5.2998089e-315  1.4916685e-154
  0 -5.3127605e-315  5.1964474e-315  5.2998088e-315   2.0000005e+00
  0  2.6815622e+154  1.4916685e-154  5.2998088e-315  1.4916685e-154
  0  -2.0000005e+00  5.1617039e-315  5.2998088e-315   0.0000000e+00
  0  -2.0000005e+00  5.1447712e-315  5.2998088e-315   0.0000000e+00
  0 -1.4916685e-154  5.1273862e-315  5.2998088e-315   0.0000000e+00
```

В чем дело? – Неверно задана спецификация формата печати параметра цикла **k**. Она (**%i**) соответствует целому типу. После исправления (например, на **%4.1f**) получаем

```
#include <stdio.h>
int main()
{ float x=10.0, y, z, t, k;
  printf(" %2c %11c %15s %15s %15s\n",'k','x',"y=1+x","z=y-1","t=z/x=1");
  for (k=0.0;k<=8.0;k++)
    { x=x/10;  y=1+x; z=y-1; t=z/x;
     printf("%4.1f %15.7e %15.7e %15.7e %15.7e\n", k,x,y,z,t);
    }
}
```

```
  k           x           y=1+x           z=y-1          t=z/x=1
 0.0   1.0000000e+00   2.0000000e+00   1.0000000e+00   1.0000000e+00
 1.0   1.0000000e-01   1.1000000e+00   1.0000002e-01   1.0000002e+00
 2.0   9.9999998e-03   1.0100000e+00   9.9999905e-03   9.9999905e-01
 3.0   9.9999993e-04   1.0010000e+00   1.0000467e-03   1.0000468e+00
 4.0   9.9999990e-05   1.0001000e+00   1.0001659e-04   1.0001661e+00
 5.0   9.9999988e-06   1.0000100e+00   1.0013580e-05   1.0013582e+00
 6.0   9.9999988e-07   1.0000010e+00   9.5367432e-07   9.5367444e-01
 7.0   9.9999987e-08   1.0000001e+00   1.1920929e-07   1.1920930e+00
 8.0   9.9999991e-09   1.0000000e+00   0.0000000e+00   0.0000000e+00
```

Сравнение с работой аналогичных ФОРТРАН-программ:

```
      program for3
      implicit none
      real*4 k, x, y, z, t
      x=10.0; k=0;  write(*,1000)
      do k=0,8
        x=x/10; y=1+x; z=y-1; t=z/x; write(*,1001) k,x,y,z,t
      enddo
 1000 format(1x,2x,'k',8x,'x',12x,'y=1+x',10x,'z=y-1',8x,'t=z/x=1')
 1001 format(1x,i3,4e15.7)
      end
```

```
   k        x             y=1+x          z=y-1         t=z/x=1
   0  0.1000000E+01  0.2000000E+01  0.1000000E+01  0.1000000E+01
 ***  0.1000000E+00  0.1100000E+01  0.1000000E+00  0.1000000E+01
 ***  0.1000000E-01  0.1010000E+01  0.9999990E-02  0.9999990E+00
 ***  0.9999999E-03  0.1001000E+01  0.1000047E-02  0.1000047E+01
 ***  0.9999999E-04  0.1000100E+01  0.1000166E-03  0.1000166E+01
 ***  0.9999999E-05  0.1000010E+01  0.1001358E-04  0.1001358E+01
 ***  0.9999999E-06  0.1000001E+01  0.9536743E-06  0.9536744E+00
 ***  0.9999999E-07  0.1000000E+01  0.1192093E-06  0.1192093E+01
 ***  0.9999999E-08  0.1000000E+01  0.0000000E+00  0.0000000E+00
```

Как видим, ФОРТРАН работает более предсказуемо нежели СИ. Ошибка в выдаче касается только данных, для которых указан неверный формат. После исправления спецификации формата с **i3** на **f4.1** получаем, естественно, тот же результат, что и на СИ.

Рассмотрим программу с параметром цикла типа **char**, которая в **десятичной**, **восьмеричной** и **шестнадцатеричной** системах счисления выводит коды букв латиницы и кириллицы. Её результат, приводимый ниже, получен, когда в качестве базовой кодовой таблицы **Linux**-система использовала **koi8r**.

```
#include <stdio.h>
int main()
{ unsigned char k,kk, kkk; int i;
  printf(" %s %3s %5s %5s %7s %3s %5s %5s %7s %3s %5s %5s\n",
         "letter","dec","oct","hex","letter","dec","oct","hex",
 "letter","dec","oct","hex");
  for (kk='A',k='a',kkk='a';k<='z';  k++,kk++,kkk++)                    {
      printf("%5c %5d   %3o %5x %5c %5d %5o %5x %5c %5d %5o %5x\n",
                k,k,k,k,kk,kk,kk,kk, kkk,kkk,kkk,kkk);                 }
  printf(" %s %3s %5s %5s %8s %3s %5s %5s \n",
         "letter","dec","oct","hex","letter","dec","oct","hex");
  for (i=192;i<=223;i++)                                               {
    k=(unsigned) i;  kk=31+(unsigned) i ;
    printf("%5c %5d %5o %5x  %5c %5d %5o %5x\n",k,k,k,k,kk,kk,kk,kk); }
  printf("%24s %5c %5d %5o %5x\n"," ",255,255,255,255);
}
```

letter	dec	oct	hex	letter	dec	oct	hex	letter	dec	oct	hex
a	97	141	61	A	65	101	41	а	193	301	c1
b	98	142	62	B	66	102	42	б	194	302	c2
c	99	143	63	C	67	103	43	ц	195	303	c3
d	100	144	64	D	68	104	44	д	196	304	c4
e	101	145	65	E	69	105	45	е	197	305	c5
f	102	146	66	F	70	106	46	ф	198	306	c6
g	103	147	67	G	71	107	47	г	199	307	c7
h	104	150	68	H	72	110	48	х	200	310	c8
i	105	151	69	I	73	111	49	и	201	311	c9
j	106	152	6a	J	74	112	4a	й	202	312	ca
k	107	153	6b	K	75	113	4b	к	203	313	cb
l	108	154	6c	L	76	114	4c	л	204	314	cc
m	109	155	6d	M	77	115	4d	м	205	315	cd
n	110	156	6e	N	78	116	4e	н	206	316	ce
o	111	157	6f	O	79	117	4f	о	207	317	cf
p	112	160	70	P	80	120	50	п	208	320	d0
q	113	161	71	Q	81	121	51	я	209	321	d1
r	114	162	72	R	82	122	52	р	210	322	d2
s	115	163	73	S	83	123	53	с	211	323	d3
t	116	164	74	T	84	124	54	т	212	324	d4
u	117	165	75	U	85	125	55	у	213	325	d5
v	118	166	76	V	86	126	56	ж	214	326	d6
w	119	167	77	W	87	127	57	в	215	327	d7
x	120	170	78	X	88	130	58	ь	216	330	d8
y	121	171	79	Y	89	131	59	ы	217	331	d9
z	122	172	7a	Z	90	132	5a	з	218	332	da

```
letter dec   oct   hex   letter dec   oct   hex  // Данный результат
   ю   192   300    c0      ъ   223   337    df  // получен в те далёкие
   а   193   301    c1      Ю   224   340    e0  // времена, когда
   б   194   302    c2      А   225   341    e1  // основной кодировкой
   ц   195   303    c3      Б   226   342    e2  // была кодировка koi8r,
   д   196   304    c4      Ц   227   343    e3  // в которой все символы
   е   197   305    c5      Д   228   344    e4  // (в частности, и русские
   ф   198   306    c6      Е   229   345    e5  // буквы) кодировались
   г   199   307    c7      Ф   230   346    e6  // восьмиразрядным (одно-
   х   200   310    c8      Г   231   347    e7  // байтным) кодом.
   и   201   311    c9      Х   232   350    e8
   й   202   312    ca      И   233   351    e9
   к   203   313    cb      Й   234   352    ea  // Сегодня базовой
   л   204   314    cc      К   235   353    eb  // кодировкой является
   м   205   315    cd      Л   236   354    ec  // utf8 (вариация Unicode)
   н   206   316    ce      М   237   355    ed  // в которой русская буква
   о   207   317    cf      Н   238   356    ee  // требует для размещения
   п   208   320    d0      О   239   357    ef  // более одного байта.
   я   209   321    d1      П   240   360    f0  // Это так называемые
   р   210   322    d2      Я   241   361    f1  // "широкие" (многобайтные)
   с   211   323    d3      Р   242   362    f2  // символы.
   т   212   324    d4      С   243   363    f3  // Для работы с ними
   у   213   325    d5      Т   244   364    f4  // мы обязаны использовать
   ж   214   326    d6      У   245   365    f5  // только "широкие" версии
   в   215   327    d7      Ж   246   366    f6  // функций и "широкие"
   ь   216   330    d8      В   247   367    f7  // типы данных:
   ы   217   331    d9      Ь   248   370    f8  //
   з   218   332    da      Ы   249   371    f9  //  wchar_t  и  wint_t
   ш   219   333    db      З   250   372    fa  //
   э   220   334    dc      Ш   251   373    fb  //  На начальном этапе
   щ   221   335    dd      Э   252   374    fc  //  знакомства с языком C
   ч   222   336    de      Щ   253   375    fd  //  не будем более касаться
   ъ   223   337    df      Ч   254   376    fe  //  "широкой" темы.
                            Ъ   255   377    ff
```

Замечания:

1. Обратите внимание, что в СИ операция , позволяет в пределах одного оператора цикла использовать несколько управляющих параметров.

2. В СИ оператор цикла **for** может быть использован и вместо **while**, и вместо **do**. Например, эквивалентны конструкции:

```
for (; L ;)     //  while( L )   // Здесь L - условие продолжения
  { T; }        //     { T; }    //           повтора;
                                 //       T - тело цикла
for (T; L)      //  do { T; }    //
  { T; }        //  while ( L ); //
```

3. Ни в коем случае не ставим **точку с запятой** сразу после круглой скобки, закрывающей заголовок цикла в операторах **for** и **while**, ибо **точка с запятой** завершит оператор цикла с пустым телом (так решит СИ-компилятор) и программа либо отработает неверно, либо зациклится. Иначе говоря, при наличии упомянутой **точки с запятой**, составной оператор, стоящий после нее, не будет воспринят компилятором в качестве тела оператора цикла.

4. В ФОРТРАНе в качестве параметра цикла могут использоваться переменные целого или вещественного типа (переменные символьного типа, как в СИ или в ПАСКАЛе, в качестве параметра цикла недопустимы). Это не является принципиальным ограничением возможностей ФОРТРАНа, так как всегда можно использовать функцию **char(i)**, получающую по коду символа сам символ.

5. Приведем пример опасности, касающейся работы с вещественным параметром цикла (и в СИ, и в ФОРТРАНе она одинакова)

```
      program for5
      implicit none
      real*4 x, h
      do x=1.2, 3.6, 1.2
         write(*,'(e15.7)') x
      enddo
      write(*,*) ' '
      h=1.2
      do x=1.2, 3.6+h/2,h
         write(*,'(e15.7)') x
      enddo
      end

$ g77 for5.for -o for5
$ ./for5
  0.1200000E+01
  0.2400000E+01

  0.1200000E+01
  0.2400000E+01
  0.3600000E+01
```

Согласно заголовка оператора цикла можно ожидать, что отпечатаются три значения **1.2, 2.4, 3.6**. Из-за наличия погрешности округления при представлении данных типа **real** последнее значение **x=3.6** отпечатано не будет, поскольку в результате прибавления к **2.4** значения **1.2** получится значение чуть большее используемой двоичной модели значения **3.6**. Так что цикл для значения **3.6** не отработает. В ситуациях, когда по каким-то причинам предпочтительнее иметь дело с параметром цикла вещественного типа, в качестве конечного значения параметра цикла рекомендуется указывать значение чуть большее требуемого, например, на половину используемого приращения.

2.12 Классификация циклических процессов

Ранее были рассмотрены три способа организации повтора: **оператор цикла с предусловием**, **оператор цикла с постусловием**, **оператор цикла с параметром**. Наличие в языках программирования (ФОРТРАН, СИ, ПАСКАЛЬ) нескольких способов организации цикла, хотя в принципе всегда можно обойтись одним, позволяет обеспечить лучшее соответствие программы решаемой задаче.

Выделяют два типа циклических процессов: **итерационные** и **типа пересчета**.

Тип процесса	типа пересчета	итерационные
Определяющий признак	Количество повторов тела цикла **ИЗВЕСТНО** программе до начала работы тела цикла, т.е. количество повторов **НЕ ЗАВИСИТ** от величин, вычисляемых в теле цикла	Количество повторов тела цикла **НЕ ИЗВЕСТНО** программе к моменту начала работы тела цикла, т.е. количество повторов **ЗАВИСИТ** от величин, вычисляемых в теле цикла
Как правило, реализуется	опрератором цикла с параметром	операторами цикла с предусловием или с постусловием
Почему?	Не надо в тело цикла вставлять оператор изменения значения параметра цикла. Изменение обеспечивается автоматически заголовком цикла	Аналог параметра цикла в итеративных процессах иногда сам является одним из результатов работы, а иногда и вовсе не нужен.

2.13 Типичный пример задачи итерационного характера

(расчет квадратного корня)

Для расчета квадратного корня во многие языки программирования обычно встроена функция **sqrt**. В качестве метода расчета **sqrt** использует **итерационный метод** последовательных приближений. Не приводя математического обоснования алгоритма, просто опишем его, посмотрим, как он работает, напишем соответствующую программу и осмыслим результаты ее работы.

Алгоритм основан на формуле расчета очередного приближения x_k к квадатному корню из числа D через предыдущее (более грубое) x_{k-1}

$$x_k = \frac{1}{2}\left(x_{k-1} + \frac{D}{x_{k-1}}\right)$$

Естественно, для начала работы необходимо задать **x0** – начальное грубое приближение к искомому корню (причем можно достаточно грубо). Например, для любого положительного **D** можно положить **x0** равным единице. Если **x0** задать равным ближайшей к $\sqrt{\mathbf{D}}$ степени двойки, то потребуется меньшее количество уточнений.

Схема метода, при условии, что значение **D** уже введено программой:

1. задание начального приближения **x1=1**
2. новое грубое **x0** – это старое точное: **x0=x1**;
3. расчет нового уточненного по формуле: **x1=(x0+D/x0)/2**
4. Продолжать уточнение? Если да, то возврат на шаг **2**, иначе **x1** есть результат.

Неопределённым осталось условие продолжения уточнений. Выберем в качестве него логическое выражение **abs(x0-x1)>eps**, сравнивающее модуль разности двух последовательных приближений **x0** и **x1** с некоторым заданным, относительно малым по сравнению с **x1** (по модулю) числом **eps**. Уясним работу алгоритма, вычислив вручную по приведенному алгоритму $\sqrt{\mathbf{D}}$ при **D=4, x1=1, eps=0.001**:

Номер итерации	Грубое x0	Более точное x1	Примечания
1	1	$x1 = \frac{1+\frac{4}{1}}{2} = \frac{5}{2} = 2.5$	**2,5** ближе к 2, чем 1, но нужной точности пока нет: **2.5-1=1.5>0.001** Продолжаем уточнение.
2	2,5	$x1 = \frac{2.5+\frac{4}{2.5}}{2} = \frac{2.5+1.6}{2} = \frac{4.1}{2} = 2.05$	**2.5-2.05=0.45>0.001** Поэтому снова продолжаем уточнение.
3	2,05	$x1 = \frac{2.05+\frac{4}{2.05}}{2} = \frac{2.05+1.9512}{2} = \frac{4.0012}{2} = 2.00065$	**2.05-2.0006=0.0494**>0.001 Ясно, что корень с нужной точностью найден за три итерации, но критерий алгоритма менее точен
4	2,0006	$x1 = \frac{2.0006+\frac{4}{2.0006}}{2} = \frac{2.0006+1.999400}{2} = \frac{4.0000}{2} = 2.0000$	**2.0006-2.0000=0.0006** **0.0006 > 0.001 ?** Нет, меньше! Корень с нужной точностью найден. Завершаем цикл

Результат: **x1=2.0000**. Так что четырех уточнений оказалось достаточным для расчета корня с точностью даже не хуже 0.0001.

Текст программы на ФОРТРАНе-77

(пример стиля старых ФОРТРАН-программ)

Предложенную схему расчета в данном случае проще всего реализовать на старых версиях ФОРТРАНа посредством моделирования оператора цикла с постусловием операторами: **continue** – оператор приема передачи управления, **if (L)** – неполный условный нерамочный логический оператор и **goto** – оператор безусловной передачи управления, который ругают все, кому не лень. Поскольку вам, возможно, придется встречаться с очень древними ФОРТРАН-программами, то лучше один раз увидеть.

```
      program tsteps
      implicit none
      real*4 d,                   ! вводимое real, корень из которого ищется;
     >     eps,                   ! требуемая погрешность извлечения корня;
     >      x0,                   ! текущее грубое приближение;
     >      x1                    ! текущее уточненное приближение;
      integer k                   ! счетчик количества итераций (уточнений);
      integer ninp / 5 /, nres / 6 /
      open (unit=ninp,file='fsqrt.inp')                 ! настройка на ввод
                                                        ! из файла fsqrt.inp
      open (unit=nres,file='fsqrt.res',status='replace') ! на вывод в файл
                                                        ! fsqrt.res
      read (ninp, 100) d, eps
      write(nres,1100) d, eps
      write(nres,1200)     ! печать заголовка выводимой таблицы
      k=0                  ! начальная настройка количества уточнений;
      x1=1                 ! начальная настройка грубого приближения к корню;
   10 continue
        x0=x1                   ! текущее грубое - это старое уточненное;
        x1=(x0+d/x0)*0.5        ! расчет текущего уточненного;
        k=k+1                                 ! расчет номера итерации (k);
        write(nres,1201) x1, k, abs(d-x1*x1) ! вывод очередного приближения,
                                              !  числа итераций и абс.погр. x1**2
      if (abs(x0-x1).gt.eps) goto 10
  100 format(e15.7)                           ! Форматы ввода
 1100 format(1x,' d=',e15.7,'   eps=',e15.7) !      и  вывода
 1200 format(1x,8x,'x1',8x,'k',6x,'aer')
 1201 format(1x,e15.7,i4,3x,e10.3)
      end
```

Результат пропуска при D = 4.0; eps = 0.001

```
 d=  0.4000000E+01   eps=  0.1000000E-02
        x1         k       aer
  0.2500000E+01    1     0.225E+01
  0.2050000E+01    2     0.202E+00
  0.2000610E+01    3     0.244E-02
  0.2000000E+01    4     0.000E+00
```

Замечания:

1. Удобство алгоритма – минимальный объем начальной настройки. Можно даже не инициализировать счетчик количества итераций и не увеличивать его, если нас оно не интересует.

2. Повтор с постусловием оптимально подходит для записи алгоритма уточнения (ни одного лишнего действия).

3. Единственное, что можно критиковать – использование метки. Современный стиль записи ФОРТРАН-программ позволяет обойтись без них:

```
      program tsteps
      implicit none
      real*4 d,          ! вводимое real, корень из которого ищется;
     >     eps,          ! требуемая погрешность извлечения корня;
     >      x0,          ! текущее грубое приближение;
     >      x1           ! текущее уточненное приближение;
      integer k          ! счетчик количества итераций (уточнений);
      integer ninp / 5 /, nres / 6 /        ! номера программных устройств;
      open (unit=ninp,file='fsqrt.inp')     ! настройка ввода из fsqrt.inp
      open (unit=nres,file='fsqrt.res',status='replace') ! вывода в fsqrt.res
      read (ninp, 100) d, eps
      write(nres,1100) d, eps
      k=0; x0=0; x1=1   ! настройка числа уточнений, и приближений к корню;
      write(nres,1200)             ! печать заголовка выводимой таблицы
      do while (abs(x0-x1).gt.eps) ! Пока не достигнута требуемая точность:
        x0=x1                      ! текущее грубое - это старое уточненное;
        x1=(x0+d/x0)*0.5           ! расчет текущего уточненного;
        k=k+1                      ! расчет номера уточнения;
        write(nres,1201) x1, k,    ! печать текущего приближения,
     >              abs(d-x1*x1) ! номера уточнения и абс. погрешности
                                   ! квадрата текущего уточнения корня;
      enddo
  100 format(e15.7)                           ! Форматы ввода d и eps
 1100 format(1x,' d=',e15.7,'   eps=',e15.7)  ! и вывода: контроля ввода,
 1200 format(1x,8x,'x1',8x,'k',6x,'aer')      ! заголовка таблицы результата
 1201 format(1x,e15.7,i4,3x,e10.3)            ! и ее текущей строки.
      end
```

Здесь передачи управления на метку нет, но зато добавляется оператор затравочной инициализации переменной **x0**. Чисто интуитивно первая программа кажется более ясной. Правда, в последней использован **повтор с предусловием**, с необходимостью требующий задания начальных значений и грубого, и уточненного приближений. Так что приходиться выполнять одну лишнюю (по сравнению с **повтором с постусловием**) операцию **'затравочной'** инициализации и уточненного приближения. Таким образом, оператор цикла с постусловием позволяет обойтись меньшим количеством начальных настроек.

Текст программы на ФОРТРАНе-95 демонстрирует возможность организации цикла с **пред-** или **постусловием** с выходом **НЕ ЧЕРЕЗ** оператор **goto**, требующий указания **МЕТКИ**, а посредством оператора **exit**, который передает управление оператору, стоящему непосредственно после **enddo**, закрывающего тело цикла.

```
      program tsteps; implicit none
      real*4 d, eps, x0, x1; integer k
      integer ninp / 5 /,    nres / 6 /
      open (unit=ninp,file='fsqrt.inp'); open (unit=nres,file='fsqrt.res')
      read (ninp, 100) d, eps;           write(nres,1100) d, eps
      k=0; x1=1; write(nres,1200)
      do; x0=x1                      ! Переопределение грубого приближения;
         x1=(x0+d/x0)*0.5            ! Перерасчет уточненного;
         k=k+1                                  ! расчет номера уточнения;
         write(nres,1201) k, x1, abs(d-x1*x1)   ! печать результата
         if (abs(x0-x1)<eps) exit               ! выход из цикла при
      enddo                                     ! достижении нужной точности;
  100 format(e15.7)
 1100 format(1x,' d=',e15.7,'   eps=',e15.7)
 1200 format(1x,3x,'k',8x,'x1',11x,'aer')
 1201 format(1x,i4,e15.7,3x,e10.3)
      end
```

Ценность оператора **exit** в том, что метка не нужна, и, значит, в принципе отсутствует возможность **'послать не туда'** или **'принять не там'**. **Точка с запятой** после **do** обязательна: иначе оператор **x0=x1** воспримется компилятором как порча открытия оператора цикла, с выводом на экран сообщения о соответствующей синтаксической ошибке. Кроме того, появятся и две наведенные ошибки, именно:

1. поскольку заголовок оператора цикла не воспринят, то компилятор *полагает*, что оператора цикла нет вообще, и, потому, программист не имеет права использовать оператор **exit**, который предназначен для выхода из тела цикла;

2. по той же причине и закрывающая рамка **enddo** оператора цикла трактуется компилятором как ошибочное написание оператора завершения программы **end**, об ожидании которого и сообщается пользователю.

```
In file fsqrt.f95:9
       do x0=x1
                  1
   Error: Syntax error in iterator at (1)
In file fsqrt.f95:13
       if (abs(x0-x1)<eps) exit
                           1
                           Error: EXIT statement at (1) is not within a loop
In file fsqrt.f95:14
       enddo
         1
         Error: Expecting END PROGRAM statement at (1)
```

Текст и результаты пропуска программы fsqrt.c

```
#include <stdio.h>
int main()
{ double d, x0, x1, eps;   int k;
  printf(" введите исходное ( D ):\n"); scanf("%le",&d);
  printf(" введите критерий (eps):\n"); scanf("%le",&eps);
  printf(" d=%le eps=%le\n", d, eps);
  x1=1; k=0;
  do { x0=x1; x1=(x0+d/x0)/2; k++;
       printf("%i\t%le\t%le\n",k,x1,fabs(d-x1*x1));
     }
  while (fabs(x0-x1)>eps);
  return 0;
}
```

```
 d=4.000000e+00 eps=1.000000e-05
 1       2.500000e+00    2.250000e+00
 2       2.050000e+00    2.025000e-01
 3       2.000610e+00    2.439396e-03
 4       2.000000e+00    3.716892e-07
 5       2.000000e+00    8.881784e-15
```

Текст и результаты пропуска программы fsqrt.cpp

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ double d, x0, x1, eps; int k;
  cout << " введите исходное ( D ):"; cin >> d;
  cout << " введите критерий (eps):"; cin >> eps;
  cout << " D=" << d << "   eps=" << eps << endl;
  x1=1; k=0;
  cout <<"  k\tx\t|D-x1*x1|\n";
  do { x0=x1; x1=(x0+d/x0)/2; k++;
       cout <<"  "<< k<<"\t"<<x1<< "\t"<<fabs(d-x1*x1)<<endl;
     }
  while (fabs(x0-x1)>eps);
  return 0;
}
```

```
 введите исходное ( D ):4
 введите критерий (eps):1e-3
 D=4   eps=0.001
  k     x       |D-x1*x1|
  1     2.5     2.25
  2     2.05    0.2025
  3     2.00061 0.0024394
  4     2       3.71689e-07
```

2.14 О чем узнали из параграфов 2.11 – 2.13?

1. Наличие в одном языке программирования нескольких разновидностей оператора цикла позволяет упростить запись алгоритма.

2. В языке СИ тело оператора цикла, если оно состоит из нескольких операторов, должно быть заключено в операторные скобки.

3. В операторах цикла с предусловием и параметром в языке СИ **не надо** ставить точку с запятой после заголовка оператора цикла (после круглой скобки, закрывающей заголовок; *Почему?*).

4. **Повтор с предусловием** и (или) **повтор с постусловием** выгодны при программировании итерационных циклов, количество повторов которых заранее **НЕ ИЗВЕСТНО**, но зависит от величин, вычисляемых в теле цикла.

5. Если количество повторов цикла до начала его работы **ИЗВЕСТНО** программе, то предпочтительнее использовать **оператор цикла с параметром**.

6. И ФОРТРАН-, и СИ-параметр цикла может быть целого или вещественного типа, НО вещественный использовать **не рекомендуется!** (*Почему?*).

7. В качестве инициализирующих значений параметра цикла СИ допускает литеральные константы; ФОРТРАН - нет.

8. В древних версиях языка ФОРТРАН оператор цикла с параметром требовал использования метки, помечающей последний оператор тела цикла, а конструирование операторов цикла с пред- или постусловием требовало еще и употребление оператора безусловной передачи управления **goto**.

9. Современный ФОРТРАН позволяет записывать операторы цикла, в принципе не используя метки и оператор **goto**.

10. Утилита **gnuplot** предоставляет удобную возможность построения графиков по численным данным, полученным в результате работы пользовательской программы и записанным, например, в файл в виде таблицы.

11. Непосредственно построением графика в среде утилиты **gnuplot** ведает команда **plot**. В частности, команда **plot 'имя файла' using 1:4 with p** построит график, взяв в качестве абсциссы и ординаты значения, хранящиеся соответственно в первом и четвертом столбцах таблицы из указанного файла.

12. Опция **with p** – указывает, что график желательно изобразить дискретными значками; опция **with l** обеспечит проведение графика непрерывной линией.

13. Значок доллара перед любой из цифр в опции **using**, указывающей номер столбца, позволяет сослаться на содержимое столбца в формуле расчета координаты. Например, **using 1:(log10(4))** означает, что график представляет собой константу равную десятичному логарифму четверки, а **using 1:(log10($4))** соответствует графику логарифма от содержимого четвертого столбца.

2.15 Четвертое домашнее задание

1. Программа расчета **n!**.
2. Программа расчета **n!!**.
3. Моделирование операции возведения в степень умножением ($\mathbf{a^n}$).
4. Экономичное решение предыдущей задачи.
5. Расчёт значения полинома **n**-ой степени по коэффициентам, хранящимся в первых **n+1** элементов одномерного массива **a**, и аргументу **x**

$$\mathbf{P_n(x) = a_0 \cdot x^n + a_1 \cdot x^{n-1} + \cdots + a_{n-1} \cdot x + a_n}$$

6. Написать программу, которая проверяет:

   “Является ли простым введенное натуральное число?”

7. Программа поиска по алгоритму Евклида наибольшего общего делителя двух введенных целых значений.
8. Программа поиска корня кубичного из введенного данного типа **real*8** по итеративной схеме:

$$\mathbf{x_0 = 1;} \qquad \mathbf{x_1 = \frac{2x_0 + \frac{D}{x_0^2}}{3}}$$

   Уточнения корня прекратить, как только модуль разности между двумя последовательными приближениями окажется меньше некоторого заданного **eps**.
9. Написать программу поиска наибольшего **eps**, сумма которого с **единицей** точно равна (*с точки зрения ЭВМ*) **единице**.
10. Написать программу поиска наибольшего **eps**, сумма которого с введенным числом типа **real(8)** в точности равна (*с точки зрения ЭВМ*) введенному числу.
11. Написать программу расчета суммы всех десятичных цифр введенного данного вещественного типа из диапазона **[0.1,1)**.
12. Написать программу получения всех цифр введенного вещественного числа **a**, после его перевода **Z**-ичную систему счисления. Ограничения: **2<=Z<=35**, $\mathbf{a} \in [\mathbf{0.1}, \mathbf{1})$].
13. Написать программу, находящую минимальное **n** – количество слагаемых усеченного гармонического ряда

$$\mathbf{H_n} = \sum_{\mathbf{i=1}}^{\mathbf{n}} \frac{\mathbf{1}}{\mathbf{i}},$$

    при котором сумма ряда $\mathbf{H_n}$ окажется больше некоторого введенного числа **a**.

3 Подпрограммы и функции.

Подпрограмма и функция – это разновидности оформления алгоритма, который можно вызывать по его имени. Их часто называют **процедурами**.

В ФОРТРАНе различие между **функцией** и **подпрограммой** закреплено синтаксически (служебные слова для описания заголовка **function** и **subroutine** – разные). В СИ описание заголовка подпрограммы по форме соответствует заголовку функции, не возвращающей никакого значения. Поэтому в СИ говорят только о функциях.

Функция возвращает результат в вызывающую её программу формально через **имя функции**, а **подпрограмма** – через свои **аргументы** (хотя последняя может и не возвращать значения).

ФОРТРАН, и СИ имеют богатые наборы встроенных (или библиотечных) функций (например, **exp**, **abs**, **atan** и др.). Тем не менее нередко возникает необходимость оформлять алгоритмы, отсутствующие среди упомянутых стандартных наборов, в виде процедур, которые можно вызывать по имени. ФОРТРАН и СИ предоставляют такую возможность. Оформление алгоритма процедурой осуществляется посредством его описания в виде **функции** или **подпрограммы**, а обращение к нему – через вызов его по имени. Процедуры выгодны для

1) разбиения сложной задачи на ряд более мелких подзадач, каждую из которых можно транслировать независимо, что упрощает отладку, тестирование и конструирование новых типов данных при использовании технологии объектно-ориентированного программирования;

2) накопления объектных кодов оттранслированных процедур для использования в будущих программах.

3.1 Понятие тестирующей программы.

Пусть при разработке программы выяснилось, что в ее исходном тексте встречаются выражения **2*x-3**, **2*y-3**, **2*z-3**, **2*a-3**, **2*b-3**. Сначала оформим расчет выражения **2*(аргумент) - 3** функцией. Ясно, что после создания новой функции нужно убедиться в правильности ее работы. Поэтому наряду с функцией потребуется и программа, тестирующая ее.

К сожалению, в качестве тестирующей программы часто используют ту проблемную, для которой нужна разрабатываемая функция. При таком подходе неточности, возможно существующие в других частях проблемной программы, могут проявиться при работе тестируемой функции, что, как правило, приводит к напрасной трате времени на поиск ошибок там, где их нет.

Рекомендация:

При разработке новой процедуры всегда пишем отдельную главную программу, которая нацелена на тестирование и выяснение наиболее оптимальной формы вызова нового алгоритма.

1. *Уяснение ситуации. Директива INCLUDE.*
2. *Тестирующая программа.*
3. *Тестируемая функция.*
4. *Создание загрузочного модуля.*
5. *Оценка правильности результата.*
6. *Пример использования директивы INCLUDE.*

3.1.1 Уяснение ситуации. Директива INCLUDE.

Исходные тексты тестирующей программы и функции размещаем в физически разных файлах (например, **testfun.for** и **fun.for** соответственно) с целью освоить наиболее оптимальную технологию разработки и сборки программ.

Размещение исходных текстов в одном файле, как правило, неудобно:

1) утомительна прокрутка текста при поиске и исправлении ошибок;

2) создание исполнимого файла требует перекомпиляции всех единиц компиляции, исходные тексты которых собраны в одном файле.

3) использование одной и той же процедуры в разных программах потребует дублирования ее текста в файл с остальными единицами компиляции.

Нередко единый исходный текст собирается из текстов разных программных единиц, размещенных в разных файлах, посредством директивы **include имя_файла**.

include – это не оператор ФОРТРАНа, а директива ФОРТРАН-компилятору — вставить на место ее расположения текст из указанного файла (см. пример в **3.1.6**).

Подобный подход создает лишь иллюзию краткости исходного текста, сохраняя упомянутые выше недостатки. Изначальная цель директивы **include** – внедрение в некоторые из единиц компиляции нужных по сути дела одинаковых фрагментов, ручной перенабор которых может внести ошибки.

При **первом знакомстве с описанием и вызовом функций** нет объективного основания для использования директивы **include**, хотя субъективное (простота подсоединения) имеется. Наша цель – освоить все этапы оптимальной технологической цепочки сборки программы. Именно поэтому пока обойдемся без директивы **include**.

3.1.2 Тестирующая программа

```
      program testfun; use my_prec
      implicit none               ! отключаем действие правила умолчания.
      real(mp) fun, x, y, z, a, b  ! указываем тип значений переменных и fun
      x=0.51_mp; y=1.0_mp; z=0.75_mp                  ! инициализация
      a=1.5_mp; b=1.499999999999999_mp                !   переменных;
      write(*,1000) x, fun(x); write(*,1000) y, fun(y) ! вывод значения каждой
      write(*,1000) z, fun(z)                         ! из них и результата
      write(*,1000) a, fun(a); write(*,1000) b, fun(b) ! соответствующего
 1000 format(1x,'fun(',d23.16,')=',d23.16)            ! вызова функции fun
      end
```

Если сейчас попытаться создать исполнимый файл посредством вызова компилятора **gfortran testfun.for**, то на экране получим несколько (пять) сообщений типа:

```
/tmp/ccxORdsS.o(.text+0x89): In function ‘MAIN__’:
: undefined reference to ‘fun_’
%/tmp/ccxORdsS.o(.text+0xdc): In function ‘MAIN__’:
%: undefined reference to ‘fun_’
%/tmp/ccxORdsS.o(.text+0x1d5): In function ‘MAIN__’:
%: undefined reference to ‘fun_’
%collect2: ld returned 1 exit status
```

Ясно, что компоновщик пытался при встрече в главной программе очередного вызова функции **fun** найти её объектный код, но не находил (ведь **fun** вообще не описана). Итог — пять сообщений: **неопределенная ссылка на внешнюю функцию fun**. Однако, если компилятор нацелить лишь на получение объектного файла главной программы, то компиляция пройдет удачно:

```
$ gfortran -c -Wall my_prec.f testfun.for
```

Таким образом, создавая тестирующую программу, мы

1) придумали имя функции и осмыслили тип возвращаемого ею значения;

2) уяснили тип аргумента функции и количество её аргументов;

3) убедились в отсутствии синтаксических ошибок в тексте главной программы;

4) получили ее объектный код (так что, если исходный текст главной программы не будем менять, то и перекомпиляция ее не потребуется).

3.1.3 Тестируемая функция

```
function fun(x)
use my_prec
implicit none
real(mp) fun, x
fun=2.0_mp*x-3.0_mp
return
end
```

Здесь:

1. Первый оператор – оператор описания заголовка функции, который определяет ее имя – **fun** и имя ее аргумента – **x** (эти имена придумывает программист).

2. **x** называется **формальным аргументом** функции. **Формальный аргумент** – это посредник, через который функция при вызове реализует доступ к данному вызвавшей ее программы.

В **ФОРТРАН**е (если нет каких-то особых ограничений) функция всегда будет работать с содержимым переменной, расположенной **по адресу**, на который ссылается через посредство **формального аргумента**.

Переменная (вызывающей программы), **адрес** (или **имя**) которой передается при вызове функции **формальному аргументу** называется **фактическим аргументом** функции.

Так что в операторах вызова **fun(x)**, **fun(y)**, **fun(z)**, **fun(a)**, **fun(b)** переменные главной программы **x**, **y**, **z**, **a** и **b** – это **фактические аргументы**. При вызове функции именно их адреса передаются **формальному аргументу** и, тем самым, функция нацеливается на работу с указанными таким образом переменными программы, которая вызывает функцию.

Формальный аргумент – это собственность вызываемой функции.

Фактический аргумент – собственность вызывающей программы.

3. Структура функции после завершения оператора ее заголовка подобна структуре главной программы: подключаем (если нужно) модуль **my_prec**, устанавливающий размер объекта типа **real**, отключаем действие правила умолчания, указываем тип значения возвращаемого функцией и тип формального аргумента. Далее следует расчетная часть. Если для её работы нужны какие-то рабочие переменные, то все они должны быть описаны и как-то вычислены внутри функции.

4. В список аргументов функции включаем лишь те, что подаются ей на вход

5. Расчетная часть нашей функции – это один оператор **fun=2.0_mp*x-3.0_mp** (найденное значение необходимо присвоить имени функции).

6. Оператор **return** — выход из функции на оператор следующий за оператором её вызова. Современный ФОРТРАН обычно обходится без оператора **return**.

7. Оператор **implicit none** всегда включаем в каждую единицу компиляции.

8. Команда **gfortran -c my_prec.f fun.for** при отсутствии синтаксических ошибок создаст в текущей директории объектный файл **fun.o**.

3.1.4 Создание исполнимого файла

После появления в текущей директории объектных файлов **testfun.o** и **fun.o** создадим из них исполнимый файл и инициируем его вызов:

```
$ gfortran testfun.o fun.o -o testfun // Сборка исполнимого файла testfun
$./testfun                                                  //    Его вызов
 fun( 0.5100000000000000E+00)=-0.1980000000000000E+01  //        и
 fun( 0.1000000000000000E+01)=-0.1000000000000000E+01  //
 fun( 0.7500000000000000E+00)=-0.1500000000000000E+01  //    результаты
 fun( 0.1500000000000000E+01)= 0.0000000000000000E+00  //
 fun( 0.1499999999999999E+01)=-0.2220446049250313E-14  //      работы
```

3.1.5 Оценка правильности результата.

Перед запуском загрузочного файла полезно составить тестовую таблицу, в которой отразить конкретное назначение теста, входные данные и **ожидаемый результат**, оставив последнюю колонку для **фактического**. Например,

N	Назначение	Исходные данные	Ожидаемый результат	Фактический результат
1	Проверка расчёта	x=0.51	-1.98	+
2	– ” –	y=1.00	-1.00	+
3	– ” –	z=0.75	-1.50	+
4	– ” –	a=1.5	-1.00	+
5	– ” –	b=1.499999999999999	-2.00d-14	**-0.222044604925E-14**

Если для каждого из тестов фактический результат совпадёт в пределах используемой разрядности с ожидаемым, то, вероятно, серьёзных проколов в алгоритме нет. Тестовые примеры, в частности, должны либо выявить опасные области значений аргументов, либо продемонстрировать вычислительную устойчивость модифицированного алгоритма в ранее опасных областях. Из таблицы заключаем, что результат, полученный **fun** для аргументов **0.51**, **1.00**, **0.75** и **1.5** верен в пределах 15-16 значащих цифр. Результат же для значения **b настораживает:**

ведь на первый взгляд **1.499999999999999 * 2 = 2.999999999999998**
и разность **2.999999999999998 - 3 = -0.200000000000000e-14**
явно не совпадает с машинной разностью равной **-0.222044604925E-14**.

1. Если машинная разность будет использована как критерий почти точного равенства уменьшаемого и вычитаемого, то результат приемлем.

2. Если же важны верные цифры разности, то результат сомнителен (еще бы – считали с 16 значащими цифрами, а верна только одна старшая).

3. Причина – конечная разрядность машинной арифметики. Да и человек, ведя расчет исключительно с 16-значной мантиссой, без априорных данных относительно остальных младших её цифр, не знает чему они равны.

   Результат **-0.200000000000000e-14**, основан на предположении, что все десятичные цифры и уменьшаемого и вычитаемого, начиная с 17-ой, равны нулю. Предположение объективно обосновано: мы сами задавали уменьшаемое и вычитаемое и знали заранее о равенстве нулю всех младших разрядов. Однако, когда уменьшаемое и вычитаемое (почти равные) заражены погрешностью округления (после обширных получающих их расчётов), то странно ожидать результата **0.200000000000000e-14**, так как ни нам, ни ЭВМ не известно содержимое соответствующих разрядов операндов.

Рекомендация:
Без нужды не вычитать на ЭВМ почти равные числа типов REAL.

3.1.6 Пример использования директивы INCLUDE.

Тестирование функции можно было бы осуществить и так:

```
$gfortran my_prec.f testfun.for fun.for -o testfun
$./testfun
```

При этом в текущей директории после создания загрузочного файла **./testfun** не окажется объектных модулей **testfun.o** и **fun.o**. Так что воспользоваться функцией **fun.for** в какой-то другой программе можно было бы только перекомпилировав ее заново совместно с вызывающей ее программой. Недостатки подобного подхода уже отмечались в пункте **3.1.1**. Тем не менее, приведем пример программы, тестирующей функцию из файла **fun.for** при использовании директивы **INCLUDE**.

```
      program testfun                 !            файл   test1.for
      implicit none
      real*8 fun
      real*8 x, y, z, a, b
      x=0.51d0; y=1d0; z=0.75d0;
      a=1.5d0; b=1.4999999999999999d0
      write(*,1000) x, fun(x)
      write(*,1000) y, fun(y)
      write(*,1000) z, fun(z)
      write(*,1000) a, fun(a)
      write(*,1000) b, fun(b)
 1000 format(1x,'fun(',d23.16,')=',d23.16)
      end
      include 'fun.for'
```

Соответствующий вызов компилятора может выглядеть и так:

```
$ gfortran test1.for
```

1. Подобный способ компиляции приемлем, если не предполагаем создавать и использовать объектный файл функции. Единственное достоинство способа — внешняя краткость исходного текста. Помним, что директива **include** лишь добавляет текст из указанного файла, но не экономит время компиляции.

2. При такой схеме подключения функции **fun** приказ компилятору еще и явно подсоединить объектный модуль **fun.o** приведет к аварийному завершению задания из-за попытки двукратного подключения объектного кода с одним именем. Например,

```
$ gfortran test1.for fun.o                        !   У функции fun
fun.o(.text+0x0): In function `fun_':             !    многократное
: multiple definition of `fun_'                   !    определение
/tmp/cciA1bBz.o(.text+0x204): first defined here  !
collect2: ld returned 1 exit status               !
```

Рекомендация:
Не пользуемся директивой include без особой надобности.

3.2 Оформление алгоритма ФОРТРАН-подпрограммой

1. Тестирующая программа.
2. Тестируемая подпрограмма subfun.
3. Описание функции и подпрограммы в стиле ФОРТРАНа-95
4. О чем узнали из первых двух параграфов? 5. Пятое домашнее задание.

3.2.1 Тестирующая программа.

Оформим алгоритм расчета функции $\mathbf{y(x) = 2x-3}$ подпрограммой с именем **subfun**. Возврат результата происходит не через имя подпрограммы, а через параметр. Поэтому у **subfun** два формальных параметра: один входной – для аргумента; другой выходной – для результата. Обращение к ФОРТРАН-подпрограмме осуществляет оператор вызова **call**, например, **call subfun(x,rx)**.

```
      program tstsfun                          !    Файл tstsfun.for
      implicit none
      real*8 fun, x,  y,  z,  a,  b, rx, ry, rz, ra, rb
      x=0.51d0; y=1d0; z=0.75d0; a=1.5d0; b=1.4999999999999999d0
      call subfun(x,rx); call subfun(y,ry); call subfun(z,rz)
      call subfun(a,ra); call subfun(b,rb)
      write(*,1000) x, rx;  write(*,1000) y, ry; write(*,1000) z, rz
      write(*,1000) a, ra;  write(*,1000) b, rb
 1000 format(1x,'fun(',d23.16,')=',d23.16)
      end
```

Переменные главной программы **x**, **y**, **z**, **a**, **b** хранят значения входных аргументов при вызове подпрограммы, а **rx**, **ry**, **rz**, **ra**, **rb** предназначены для приёмки соответствующих результатов. Объектный модуль тестирующей программы получаем командой **$ gfortran -c tstsfun.for**.

3.2.2 Тестируемая подпрограмма subfun

```
      subroutine subfun(x,res)                 !    Файл subfun.for
      implicit none
      real*8 x, res
      res=2d0*x-3d0
      end
```

Преобразование функции в подпрограмму достаточно формально:

a) заменили слово **function** на **subroutine** и придумали имя для подпрограммы.

b) добавили **формальный аргумент** (**res**) и присвоили значение результата по указанному в **res** адресу, исключив естественно присваивание результата имени функции.

Объектный код подпрограммы получаем командой **gfortran -c subfun.for**.
Исполнимый файл создается командой **gfortran tstsfun.o subfun.o -o tstsfun**.
Вызов **./tstsfun** приводит к результатам из пункта **3.1.4**.

3.2.3 Интерфейс в ФОРТРАНа-95

Интерфейс (в программировании) – минимально необходимая для вызова процедуры совокупность сведений о ней (см., например, [10]), именно: подпрограмма это или функция, каково количество её параметров, их типы и атрибуты.

В ФОРТРАНе-77 интерфейс – неявный, определяется оператором вызова процедуры (т.е. полагается, что соответствие типа фактического и формального параметров с непогрешимостью обеспечивается программистом). Так, если бы в исходном тексте программы из пункта **3.2.1** переменной **y** случайно сопоставили тип **integer**:

```
program tersfun                          !    Файл tersfun.for
implicit none
real*8 fun, x,       z,  a,  b
real*8     rx,  ry, rz, ra, rb
integer y                                ! <--= модель опечатки
y=1
x=0.51d0;     z=0.75d0; a=1.5d0;  b=1.499999999999999d0
call subfun(x,rx); call subfun(y,ry); call subfun(z,rz)
call subfun(a,ra); call subfun(b,rb)
write(*,*) x, rx;
write(*,*) y, ry                         ! формат по списку вывода
write(*,*) z, rz
write(*,*) a, ra;  write(*,*) b, rb
end
```

при прежнем описании подпрограммы **subfun**, то после

```
$ gfortran -c tersfun.for
$ gfortran -c subfun.for
$ gfortran tersfun.o subfun.o -terrible
$ ./terrible
```

получили бы следующий результат

```
 0.510000000000000       -1.98000000000000
         1  -3.00000000000000                    //     2*1-3 = -3 ???
 0.750000000000000       -1.50000000000000
 1.50000000000000        0.00000000000000
 1.50000000000000      -2.220446049250313E-015
```

И сколько времени мы искали бы ошибку в формулах или в значении аргумента, удивляясь результату (типичному для ФОРТРАНа-77), из-за того, что не воспользовались возможностями ФОРТРАНа-95, в котором есть средства автоматического контроля соответствия формальных и фактических аргументов процедур — два способа описания **интерфейса** внешних процедур: **явное** и **модульное**.

1. Например, программа, вызывающая **fun** и **subfun**, при **явном** описании интерфейса может выглядеть так

```
program testfun                  !               Файл testfun.f95
implicit none                    !
interface                        ! Начало интерфейсного блока
  function fun(x)                !   Его тело состоит здесь из
    real(8) fun                  !   описания интерфейсов двух
    real(8) x                    !   внешних процедур:fun и subfun.
  end function fun               !   Завершение интерфейса   fun
  subroutine subfun(x,y)         !   Начало     интерфейса subfun
    real(8) x                    !
    real(8) y                    !
  end subroutine subfun          !   Завершение интерфейса subfun
end interface                    ! Завершение интерфейсного блока
real*8 x, y, z, a, b             !
x=0.51d0; y=1d0; z=0.75d0        !
a=1.5d0; b=1.4999999999999999d0  !
write(*,*) x, fun(x)             ! Пять
write(*,*) y, fun(y)             !     тестовых
write(*,*) z, fun(z)             !           вызовов
write(*,*) a, fun(a)             !                 функции fun
write(*,*) b, fun(b)             ! и
call subfun(b,y)                 ! один вызов подпрограммы subfun.
write(*,*) b, y
end
```

После компиляции, сборки исполнимого кода и вызова его на исполнение

```
$ gfortran -c testfun.f95
$ gfortran -c subfun.for
$ gfortran -c fun.for
$ gfortran  testfun.o subfun.o fun.o -o testfun
$ ./testfun
```

получаем

```
  0.510000000000000       -1.98000000000000
   1.00000000000000       -1.00000000000000
  0.750000000000000       -1.50000000000000
   1.50000000000000        0.00000000000000
   1.50000000000000      -2.220446049250313E-015
   1.50000000000000      -2.220446049250313E-015
```

Достоинство указания интерфейса в том, что в случае опечатки, подобной той, которую намеренно допустили в программе **tersfun** (в начале пункта **3.2.3**, — сопоставления переменной **y** типа отличного от **real*8**:

```
      program ttssfun                 !               Файл ttssfun.f95
      implicit none                   !
      interface                       ! Начало интерфейсного блока
        function fun(x)               !
          real(8) fun                 !
          real(8) x                   !
        end function fun              !
        subroutine subfun(x,y)        !
          real(8) x                   !
          real(8) y                   !
        end subroutine subfun         !
      end interface                   ! Завершение интерфейсного блока
      real*8 x,    z, a, b            !
      integer   y                     !  <--= случайная опечатка
      y=1d0                           !
      x=0.51d0;        z=0.75d0       !
      a=1.5d0; b=1.499999999999999d0  !
      write(*,*) x, fun(x)            ! Пять
      write(*,*) y, fun(y)            !     тестовых
      write(*,*) z, fun(z)            !           вызовов
      write(*,*) a, fun(a)            !                 функции fun
      write(*,*) b, fun(b)            ! и
      call subfun(b,y)                ! один вызов подпрограммы subfun.
      write(*,*) b, y
      end
```

ещё на шаге компиляции **ttssfun** получим сообщение:

```
$ gfortran -c ttssfun.f95
 In file ttssfun.f95:19
      write(*,*) y, fun(y)
                        1
Error: Type/rank mismatch in argument 'x' at (1)
 In file ttssfun.f95:23
      call subfun(b,y)
                  1
Error: Type/rank mismatch in argument 'y' at (1)
```

так как при наличии интерфейса компилятор в состоянии проверить все ли **фактические** аргументы процедур **fun** и **subfun** по типу соответствуют **формальным**.

2. При большом количестве процедур обширность **интерфейсного блока** понижает наглядность программы, использующей его. ФОРТРАН предоставляет возможность ужать описание интерфейсного блока (как бы велик он не был) до одной строки путём размещения его в отдельном файле и директивы компилятору **include** указанной в вызывающей программе. Например,

```
program testfun                  !               Файл testfun.f95
implicit none                    !
include 'my_interf'              ! Вставка интерфейсного блока
                                 ! через директиву include
real*8 x, y, z, a, b             !
x=0.51d0; y=1d0; z=0.75d0        !
a=1.5d0; b=1.499999999999999d0   !
write(*,*) x, fun(x)             ! Пять
write(*,*) y, fun(y)             !     тестовых
write(*,*) z, fun(z)             !           вызовов
write(*,*) a, fun(a)             !                 функции fun
write(*,*) b, fun(b)             ! и
call subfun(b,y)                 ! один вызов подпрограммы subfun.
write(*,*) b, y
end


interface                        !                Файл myinterf
  function fun(p)
    real(8) fun
    real(8) p
  end function fun
  subroutine subfun(x,y)
    real(8) x
    real(8) y
  end subroutine subfun
end interface
```

В описании внешних процедур и их интерфейсов допустим несущественный разнобой. Так, при описании **fun** и **subfun** типы имен указаны посредством **real*8**, а при описании интерфейса — посредством **real(8)**. Имена формальных параметров при описании интерфейса могут отличаться от их имён при описании самой процедуры (сравните описания самой **fun** и её интерфейса).

3. Описание интерфейса можно разместить и в **модуле** (**module**) — новой единице компиляции, которой не было в ФОРТРАНе-77, и обеспечивающей для программных единиц (**program**, внешних **function** и **subroutine**), подсоединяющих её через оператор **use**, возможность пользоваться именами типов, переменных, процедур и **интерфейсов**, описанных в ней.

   В терминах *исходный модуль, объектный модуль, загрузочный модуль* слово *модуль* можно заменить на слово *файл*. Аналогично единицу компиляции

module можно назвать **модульным файлом**. Так что в программировании термин **модуль** в зависимости от контекста может обозначать несколько разные понятия. Не вдаваясь пока в продробности, приведём лишь пример размещения интерфейса в **модуле**:

```
program testfun                 !               Файл testfun.f95
use modinterf                   ! Подключение модуля
implicit none
real*8 x, y, z, a, b
x=0.51d0; y=1d0; z=0.75d0
a=1.5d0; b=1.4999999999999999d0
write(*,*) x, fun(x);  write(*,*) y, fun(y)
write(*,*) z, fun(z);  write(*,*) a, fun(a); write(*,*) b, fun(b)
call subfun(b,y)
write(*,*) b, y
end

module modinterf                !               Файл myinterf.for
implicit none
interface
  function fun(p)
    real(8) fun
    real(8) p
  end function fun
  subroutine subfun(x,y)
    real(8) x
    real(8) y
  end subroutine subfun
end interface
end module modinterf
```

```
$ gfortran -c modinterf.for  # Модуль должен быть откомпилирован до
$ gfortran -c testfunm.f95   # единиц компиляции, к которым подсоединяется.
$ gfortran -c subfun.for     # Его объектный файл необходим для сборки
$ gfortran -c fun.for        # исполнимого файла.
$ gfortran  testfunm.o  modinterf.o subfun.o fun.o -o testfunm
```

Замечания:

1. **Интерфейсный блок** помещается среди операторов описания.

2. Процедура, заголовок которой указан в интерфейсном блоке, считается **внешней** по отношению к вызывающей её программе. Если ее имя совпадает с именем встроенной процедуры, то встроенная недоступна для использования.

3. Условия, при которых необходимо явное задание интерфейса: наличие у процедуры необязательных формальных параметров, возврат через имя функции массива, доступ к внешним процедурам, написанных на других языках, задании новых операций, перегрузке процедур и др. (подробнее см., например, [9]).

3.2.4 О чем узнали из первых двух параграфов?

1. **Подпрограммы** и **функции** (или **процедуры**) предоставляют возможность
   а) разбиения сложной задачи на ряд более мелких независимых подзадач, что принципиально облегчает отладку и тестирование.
   б) накопления объектных кодов автономно откомпилированных процедур.
2. Процедура – это именованный алгоритм, который можно вызывать по имени.
3. Оператор описания процедуры сопоставляет алгоритму имя.
4. **Функция** - всегда возвращает результат через свое имя.
5. **Подпрограмма** возвращает результат либо через свои аргументы, либо не возвращает вообще.
6. В **ФОРТРАНе** для описания заголовка **функции** и **подпрограммы** используются соответственно служебные слова **function** и **subroutine**.
7. В **СИ** говорят только о функциях. (*Почему?*).
8. При разработке новой процедуры пишем тестирующую ее программу.
9. Исходные тексты процедуры и вызывающей её программы невыгодно размещать в одном файле (наиболее разумно в разных).
10. В ФОРТРАНе **include** – не оператор ФОРТРАНа, а директива компилятору внедрить текст нужного файла на место ее расположения.
11. **Формальный аргумент процедуры** – посредник, через который процедура при вызове получает доступ к данному программы, её вызвавшей.
12. **Фактический аргумент процедуры** – это данные программы, которые фактически подаются процедуре при вызове через список аргументов.
13. В ФОРТРАНе и СИ разные схемы доступа к фактическому аргументу: **ФОРТРАН** по умолчанию обычно работает непосредственно по адресу данного программы, вызвавшей процедуру (то есть по адресу фактического аргумента); **СИ** – только с копией содержимого фактического аргумента, которая передана формальному.
14. Оператор **implicit none** обязательно включаем в каждую единицу компиляции ФОРТРАНа.
15. В ФОРТРАНе-77 контроль за соответствием фактических и формальных параметров осуществляет программист. В ФОРТРАНе-95 есть оператор **interface**, позволяющий этот контроль осуществлять компилятору.
16. В ФОРТРАНе-95 **интерфейс** может быть описан **явно** или подсоединён через **модуль**.

3.2.5 Пятое домашнее задание (ФОРТРАН).

Оформить алгоритмы функцией и подпрограммой (не забыть о их тестировании).

1. Анализ и решение линейного уравнения вида **ax=b** (одинарная точность).
2. Расчёт $\mathbf{a^n}$.
3. Поиск наибольшего общего делителя по алгоритму Евклида.
4. Поиск с заданной точностью ϵ кубичного корня из вещественного **D** (**real(mp)**).
5. Поиск наименьшего ϵ, для которого *с точки зрения ЭВМ* $1+\epsilon \neq 1$ (**real(mp)**).
6. Нахождение суммы всех (хранимых в мантиссе) цифр введенного десятичного числа из диапазона **[0.1,1)**. Результаты для значений **0.25**, **0.5**, **0.1** и **0.11** письменно прокомментировать и обосновать (**real(mp)**, **mp=4, 8, 10, 16**).
7. Нахождение **n**, наименьшего количества слагаемых усеченного гармонического ряда, при котором соответствующая частичная сумма ряда окажется больше заданного вещественного **a** (**real(mp), mp=4**).
8. Расчёт значения функции $\mathbf{f(x)} = \dfrac{\mathbf{1.6} - \sqrt{\mathbf{2.56 - x^{21}}}}{\mathbf{x^{21}}}$. Главная программа должна обеспечить вызов соответствующей функции для значений аргумента **x** от **0.1** до **1** с шагом **0.1** (**real(mp), mp=8**). Письменно сформулировать:
   а) какие неожиданности обнаружились при оценке правильности результатов расчета, выполненного непосредственно по приведенной формуле?
   б) аналитически найти численное значение предела: $\lim\limits_{\mathbf{x\to 0}} \mathbf{f(x)}$.
   в) совпадает ли оно с f(0.1)?
   г) какова объективная причина несовпадения?
   д) какова субъективная причина несовпадения?
   е) каким образом можно значительно уменьшить влияние последней?
   ж) разработать функцию точного расчета искомого отношения и добавить её тестирование в главную программу.
9. Для функции $\sin \mathbf{x}$ известен ряд Маклорена

$$\sin \mathbf{x} = \sum_{\mathbf{k=0}}^{\infty} (-\mathbf{1})^{\mathbf{k}} \frac{\mathbf{x^{2k+1}}}{\mathbf{(2k+1)!}} \quad ,$$

   сходящийся при любом $|x| < \infty$. Известно, что остаток этого ряда меньше по модулю первого неучтенного в частичной сумме слагаемого. Иначе говоря, гарантируется, что при любом конечном значении аргумента **x** накопленная частичная сумма будет отличаться по модулю от истинного значения $\sin \mathbf{x}$ не

более чем на ϵ, как только очередное слагаемое, вычисленное в процессе ее накопления, окажется меньше этого наперед заданного малого $\epsilon > \mathbf{0}$.

Разработать процедуру расчета функции sin(x) по указанной схеме. (одинарная точность)

В качестве входных параметров процедуры использовать аргумент $\mathbf{x}$ и абсолютную погрешность ϵ; а в качестве выходных накопленную сумму, величину наибольшего по модулю из учтенных в сумме слагаемых и их количество.

Тестирующая программа должна после ввода ϵ обеспечить получение таблички результатов для значений аргумента $\mathbf{x} = \dfrac{\pi}{\mathbf{6}} + \mathbf{2}\pi\mathbf{k}$ при $\mathbf{k}$=**0(1)20**.

Каждая строка таблички должна содержать значения аргумента, встроенной функции $\sin \mathbf{x}$, суммы, вычисленной разработанной процедурой, величину наибольшего по модулю из учтенных в ней слагаемых и их количество.

Письменно ответить на вопросы:

А. При всех ли $\mathbf{x}$ результат совпадает с правильным?

Б. Как изменяется точность результата с ростом аргумента?

В. В чем причина рассогласования точности расчета на ЭВМ с точностью, гарантированной математическим анализом – ведь суммирование прекращается именно тогда, когда остаток ряда меньше заданного ϵ?

Г. Может ли помочь и как уяснению причин рассогласования вывод значения максимального по модулю слагаемого, учтенного в сумме?

Д. Почему **встроенная** функция при любом из заданных значений аргумента $\mathbf{x}$ дает результат верный в пределах используемой разрядности, хотя ее алгоритм суммирования принципиально не отличается от алгоритма разработанной подпрограммы?

Е. при каких значениях аргумента $\mathbf{x}$ встроенные функции $\sin \mathbf{x}$ и $\cos \mathbf{x}$ будут давать результат неверный в старшей цифре мантиссы?

Ж. Как следует подправить нашу подпрограмму, чтобы результат ее работы практически не отличался от результата работы встроенной функции?

З. Какую встроенную функцию ФОРТРАНа можно употребить для указанной коррекции?

10. Разработать для режима удвоенной точности функцию расчета выражения

$$\mathbf{f(x)} = \mathbf{5040} * \frac{\sin \mathbf{x} - \mathbf{x} + \dfrac{\mathbf{x^3}}{\mathbf{6}} - \dfrac{\mathbf{x^5}}{\mathbf{120}}}{\mathbf{x^7}}$$

которая получает верный (в пределах используемой разрядности) результат. Пропустить тестирующую программу для трёх диапазонов аргумента $\mathbf{x}$:

1) $x = 0.001(0.001)0.011$; 2) $x = 1.0(1.0)11.0$; 3) $x = 100(100)1100$.

3.3 Тестирование СИ-функций.

1. *Тестирующая программа для СИ-функции.*
2. *Тестируемая функция.*
3. *Компиляция, компоновка и пропуск СИ-программы.*
4. *Первая попытка оформления алгоритма СИ-функцией типа void.*
5. *Использование указателя для адресации результата СИ-функции.*
6. *Понятие о скрытом указателе C++.*
7. *Два способа доступа функции к переменным главной программы.*
8. *О чем узнали из третьего параграфа?*
9. *Шестое домашнее задание.*

3.3.1 Тестирующая программа для СИ-функции.

Поместим программу, тестирующую функцию **fun**, в файл **tstfun.c**:

```
#include <stdio.h>                                  //   Файл tstfun.c
double fun(double);
int main()
{ double x, y, z, a, b;
  x=0.51; y=1; z=0.75; a=1.5; b=1.499999999999999;
  printf(" fun(%23.16e)=%23.16e\n",x,fun(x));
  printf(" fun(%23.16e)=%23.16e\n",y,fun(y));
  printf(" fun(%23.16e)=%23.16e\n",z,fun(z));
  printf(" fun(%23.16e)=%23.16e\n",a,fun(a));
  printf(" fun(%23.16e)=%23.16e\n",b,fun(b));
  return 0;
}
```

1. Посредством директивы #**include** <**stdio.h**> подключен интерфейс функций стандартного ввода/вывода (в частности, и **printf**).

2. Для функции **fun**, которая вызывается в тестирующей программе, явно указан **прототип** функции **double fun(double);** (в С интерфейс функции часто называют её прототипом), что позволяет компилятору проконтролировать соответствие свойств фактических и формальных аргументов. В СИ перед единицей компиляции с обращениями к функциям, всегда помещаем описания их прототипов. При независимом получении объектного файла **tstfun.o** и отсутствии в исходном **tstfun.c** прототипа **fun** СИ по умолчанию *полагает*, что **fun** возвращает значение целого типа (а это нас не устроит). **g++**, правда, при компиляции **tstfun.c** потребует объявления имени **fun**.

3. Описание прототипа можно поместить явно, как сделано в данной программе. Но можно и через файл-посредник, подключаемый еще одной директивой **include**. Выбор способа подключения прототипа за программистом. Обычно в качестве расширения таких файлов-посредников часто используется литера **h** (от **header**, то есть файл с определениями *заголовков* функций и атрибутов их аргументов).

4. Оператором присваивания инициализируются переменные **x**, **y**, **z**, **a** и **b**.

5. Печатается значение каждой из них и результат соответствующего вызова **fun**

3.3.2 Тестируемая функция

Исходный текст тестируемой функции **fun** поместим в файл **fun.c**

```
 double fun(double x)                          //     Файл  fun.c
{ return (2*x-3); }
```

1. Первый оператор – оператор описания заголовка функции, который определяет ее имя – **fun**, имя ее аргумента – **x**, его тип и тип значения, возвращаемого функцией. В ФОРТРАНе тип функции можно указать непосредственно перед словом **function**, например, **real*8 function fun(x)** или даже **double precision function fun(x)** (как в СИ), но тип аргумента синтаксически указывается отдельным оператором. Так что СИ-заголовок функции выглядит элегантнее и информативнее ФОРТРАН-заголовка.

2. Как и в ФОРТРАНе, **x** называется **формальным аргументом** функции, через посредство которого функция при вызове реализует доступ к данному вызвавшей ее программы. В отличие от ФОРТРАНа СИ реализует иную схему доступа к **фактическому аргументу**. В ФОРТРАНе по умолчанию **формальному аргументу** передается адрес **фактического**, а в СИ – значение **фактического**. Подробнее о смысловой нагрузке двух схем доступа функции (**по адресу** и **по значению**) чуть ниже.

3. Возврат результата через имя функции и передача управления программе, вызвавшей функцию, в СИ осуществляется одним оператором **return(2*x-3);**.

4. Тело СИ-функции располагается после ее заголовка в фигурных скобках. При этом после заголовка перед фигурной скобкой, открывающей тело, ни в коем случае не ставим **;**. Ее наличие превращает заголовок функции в объявление прототипа.

3.3.3 Компиляция, компоновка и пропуск СИ-программы.

```
$ gcc -c tstfun.c                  // получение объектного файла tstfun.o;
$ gcc -c fun.c                     // получение объектного файла fun.o;
$ gcc tstfun.o fun.o -lm -o tstfun // получение исполнимого файла tstfun;
                                   // опция -lm подключение матем. библиотеки;
$ ./tstfun                         // загрузка и выполнение исполнимого файла.
```

```
 fun( 5.1000000000000001e-01)=-1.9800000000000000e+00
 fun( 1.0000000000000000e+00)=-1.0000000000000000e+00
 fun( 7.5000000000000000e-01)=-1.5000000000000000e+00
 fun( 1.5000000000000000e+00)= 0.0000000000000000e+00
 fun( 1.4999999999999989e+00)=-2.2204460492503131e-15
```

3.3.4 Первая попытка оформления алгоритма СИ-функцией типа *void*.

СИ-функция типа **void** не возвращает ничего через свое имя вызывающей программе (это аналог ФОРТРАН-подпрограммы). Возврат результата **void**-функция (если он нужен) выполняет через дополнительный формальный параметр (возможен возврат и через глобальные параметры, но пока не будем их использовать).

Если переложим запись ФОРТРАН-подпрограммы **subfun.for** на язык СИ по интуиции, то получим следующие исходные тексты:

```
//#include <stdio.h>                                //    файл testsubf.c
void subfun(double,double);
int main()
{ double x, y, z, a, b;
  double resx, resy, resz, resa, resb;
  x=0.51; y=1; z=0.75; a=1.5; b=1.499999999999999;
  subfun(x,resx);  printf("   main: x=%23.16e  resx=%23.16e\n",x,resx);
  subfun(y,resy);  printf("   main: y=%23.16e  resy=%23.16e\n",y,resy);
  subfun(z,resz);  printf("   main: z=%23.16e  resz=%23.16e\n",z,resz);
  subfun(a,resa);  printf("   main: a=%23.16e  resa=%23.16e\n",a,resa);
  subfun(b,resb);  printf("   main: b=%23.16e  resb=%23.16e\n",b,resb);
  return 0;
}
```

```
void subfun(double x, double y)                 //    файл subfun.c
{ y=2*x-3; printf(" subfun: x=%23.16le     y=%23.16le\n",x,y);}
```

Осознать результаты работы СИ-функции проще при наличии в теле функции вывода ее аргументов. После компоновки исполнимого файла и его вызова получим:

```
$ gcc -c testsubf.c
$ gcc -c subfun.c
$ gcc testsubf.o subfun.o -lm -o testsubf
$ ./testsubf
 subfun: x= 5.1000000000000001e-01     y=-1.9800000000000000e+00
         x= 5.1000000000000001e-01  resx=4.8528751426136228e-270
 subfun: x= 1.0000000000000000e+00     y=-1.0000000000000000e+00
         y= 1.0000000000000000e+00  resy=4.8729129281732584e-270
 subfun: x= 7.5000000000000000e-01     y=-1.5000000000000000e+00
         z= 7.5000000000000000e-01  resz= 0.0000000000000000e+00
 subfun: x= 1.5000000000000000e+00     y= 0.0000000000000000e+00
         a= 1.5000000000000000e+00  resa=2.8054667906184338e-317
 subfun: x= 1.4999999999999989e+00     y=-2.2204460492503131e-15
         b= 1.4999999999999989e+00  resb=-2.9634328625631385e+303
```

Как видим, результат, получаемый внутри функции (содержимое второго аргумента), – верен, но это значение не передается соответствующему фактическому аргументу при вызове функции. Причина:

в СИ и ФОРТРАНе разные механизмы доступа функции к фактическим аргументам (пока функция использовала аргументы только для приема входных данных, это различие механизмов доступа внешне не проявлялось).

3.3.5 Использование указателя для адресации результата СИ-функции.

```
#include <stdio.h>                                    // файл testsub1.c
void subfun(double,double*);
int main()
{ double x, y, z, a, b;
  double resx, resy, resz, resa, resb;
  x=0.51; y=1; z=0.75; a=1.5; b=1.499999999999999;
  subfun(x,&resx);  printf("         x=%23.16e  resx=%23.16e\n",x,resx);
  subfun(y,&resy);  printf("         y=%23.16e  resy=%23.16e\n",y,resy);
  subfun(z,&resz);  printf("         z=%23.16e  resz=%23.16e\n",z,resz);
  subfun(a,&resa);  printf("         a=%23.16e  resa=%23.16e\n",a,resa);
  subfun(b,&resb);  printf("         b=%23.16e  resb=%23.16e\n",b,resb);
  return 0;
}
```

```
void subfun(double x, double* y)                     // файл subfun1.c
{ (*y)=2*x-3;
  printf(" subfun: x=%23.16le     y=%23.16le\n",x,*y);
}
```

Здесь при описании прототипа функции **subfun** указано, что

1. тип ее первого аргумента **double**. В СИ при вызове функции формальному аргументу передается НЕ АДРЕС (как в ФОРТРАНе), а **копия значения** соответствующего фактического аргумента. В этом случае можем изменять внутри тела функции ее формальный аргумент, но понимаем, что при вызове изменение фактического аргумента функции не произойдет – функция работает не с переменной вызывающей программы, не по адресу фактического аргумента.

2. тип второго аргумента функции – **double***. **Звездочка** (в данном контексте по синтаксису СИ) указывает, что второй формальный аргумент — это **адрес** памяти, в которой находится данное типа **double**. Поскольку аргументы СИ-функции всегда передаются по значению (в частности, и адреса), то функция не может изменить адрес, переданный ей в качестве аргумента, но может работать с адресуемой им памятью (в частности, и изменять содержимое памяти).

3. В СИ описание **имя_типа *имя_переменной** означает переменную для хранения **адреса**, по которому находится значение указанного типа (такая переменная называется **указателем**). Например,

   ```
   #include <stdio.h>
   int main()      //   a  - переменная для хранения данного целого типа;
   { int a, *pint; // pint - указатель на область памяти для данного типа int;
     a=10;         // присвоили переменной a целочисленную десятку;
     pint=&a;      // присвоили переменной pint адрес переменной a;
     printf("a=%d a=%d\n", a, *pint); // вывели значение переменной a двумя
                                      // способами: через имя переменной и
     return 0;                        // через указатель, ссылающийся на нее.
   }
   ```

Символ * в этой программе используется в двух местах: при описании указателя (третья строка) и при использовании значения, хранящегося по адресу, на который ссылается указатель (шестая строка). ***pint** в шестой строке — **операция разыменования** указателя **pint**, т.е. доступ к содержимому ячейки, на которую ссылается указатель. Именно так, функция **subfun** заносит вычисленное значение в ячейку адресуемую фактическим аргументом ***y=2*x-3**.

4. При вызове функции **subfun** из главной программы в качестве первого фактического аргумента через имена **x**, **y**, **z**, **a**, **b** подаются **значения** соответствующих переменных, а в качестве второго фактического аргумента подаются **значения АДРЕСОВ** тех переменных, в которые намечаем поместить результат. Операция взятия адреса в СИ обозначается символом **&**.

3.3.6 Понятие о скрытом указателе C++.

Для моделирования передачи аргумента по адресу в C++ имеется (наряду с обычным **указателем**) родственный последнему элемент – **ссылка** (*скрытый указатель* см., например, [16, параграф 4.6]). Предыдущая программа и **void**-функция с использованием параметра-ссылки и результаты пропуска могут выглядеть так

```
#include <stdio.h>                    // файл testsub1.cpp
void subfun1(double,double &);        // описание прототипа subfun1
int main()                            // с параметром-ссылкой
{ double x, y, z, a, b;
  double resx, resy, resz, resa, resb;
  x=0.51; y=1; z=0.75; a=1.5; b=1.499999999999999;
  subfun1(x,resx);  printf("         x=%23.16e  resx=%23.16e\n",x,resx);
  subfun1(y,resy);  printf("         y=%23.16e  resy=%23.16e\n",y,resy);
  subfun1(z,resz);  printf("         z=%23.16e  resz=%23.16e\n",z,resz);
  subfun1(a,resa);  printf("         a=%23.16e  resa=%23.16e\n",a,resa);
  subfun1(b,resb);  printf("         b=%23.16e  resb=%23.16e\n",b,resb);
  return 0;
}
```

```
void subfun1(double x, double &y)                    // файл subfun1.c
{ y=2*x-3;}
```

```
$ c++ testsub1.cpp -c                       // Получение объектных
$ c++ subfun1.cpp -c                        // и исполнимого файлов;
$ c++ testsub1.o subfun1.o -lm  -o testsub1 // Вызов исполнимого и
$ ./testsub1                                // результат его работы:
        x= 5.1000000000000001e-01  resx=-1.9800000000000000e+00
        y= 1.0000000000000000e+00  resy=-1.0000000000000000e+00
        z= 7.5000000000000000e-01  resz=-1.5000000000000000e+00
        a= 1.5000000000000000e+00  resa= 0.0000000000000000e+00
        b= 1.4999999999999989e+00  resb=-2.2204460492503131e-15
```

При оформление аргумента функции параметром-ссылкой не указываем явно операцию взятия адреса соответствующего фактического аргумента, а при описании функции не нужно использовать операцию разыменования (компилятор автоматически берет все на себя).

3.3.7 Режимы доступа функции к переменным главной программы.

Доступ функции к фактическому аргументу можно организовать по разному:

1. Передать формальному аргументу **адрес** фактического аргумента – и тогда функция работает с областью памяти, адрес которой передан.
   Передача аргумента по адресу (или по ссылке) нужна, если фактический аргумент должен получить результат работы подпрограммы.

   ФОРТРАН, как правило, нацелен на передачу аргумента по адресу.

2. Передать формальному параметру лишь копию **значения** фактического аргумента, так что функция будет работать только с этой копией, и изменение внутри функции значения формального аргумента не приведет к изменению значения фактического.

   Передача аргумента по значению нужна, если по каким-то причинам **хотим**, чтобы функция не смогла изменить значение фактического аргумента, даже если мы изменяем ее соответствующий формальный аргумент.

   СИ и С++ всегда настроены на передачу аргумента по значению.

3. В принципе есть и другие способы передачи, например, передать формальному параметру не адрес и не значение фактического, а выражение, по которому значение фактического перевычисляется заново столько раз сколько имя формального встречается в теле функции.

Какой из первых двух способов лучше? (других пока не касаемся)

1. ФОРТРАН использует **передачу параметра по адресу**. Она не требует дополнительной памяти под размещение содержимого, хранящегося по этому адресу, и не тратит время на пересылку, не требует, как Си, явного указания на то, что именно передается: адрес или содержимое. Фактический параметр изменяет свое содержимое при изменении соответствующего формального аргумента. Так подпрограмма **subfun**, протестированная в **3.2**, брала исходное данное по адресу первого фактического аргумента и помещала результат по адресу второго.

2. Однако, есть вопрос: *Чего ожидать в результате работы программы:*

```
program terrible1          ! Результат работы:
write(*,*) 1               ! 1
call terr(1)               !  terr1: i=1
write(*,*) 1               !  Segmentation fault
end                        ! Так что, когда на вход попдпрограмме,
subroutine terr(i)         ! изменяющей содержимое своего формального
write(*,*) 'terr1: i=',i   ! аргумента, подается константа, gfortran
i=777                      ! не допустит её изменения и программа
write(*,*) 'terr2: i=',i   ! завершиться аварийным сообщением.
end
```

На старых ФОРТРАН-компиляторах таким способом можно было изменить значение константы **1** (или любой другой), что, хотя иногда и оказывалось удобным, тем не менее, вполне обосновано подвергалось критике. Аналогичная программа на языке СИ не завершается аварийно, как фортрановская:

```
#include <stdio.h>
void terr(int);           // Результат работы:
int main()                //
{ printf(" 1=%i\n",1);    //       1=1
  terr(1);                //       i=1     Внутри функции формальный
  printf(" 1=%i\n",1);    //       i=777   аргумент меняется, но
  return 0;               //       i=1     фактический аргумент,
}                         //               несмотря на изменение
void terr(int i)          //               формального остается
{  printf(" i=%i\n",i);   //               в неприкосновенности.
   i=777;                 // Так что налицо достоинство передачи
   printf(" i=%i\n",i);   // аргумента по значению.
}
```

3. Вопрос, аналогичный предыдущему, возникает и в случае, когда в качестве фактического аргумента подается выражение, а подпрограмма изменяет соответствующий формальный аргумент. Например,

```
      program terrible2         ! Результат работы
      m=1                       !
      write(*,*) '1:  m=',m     !   1: m= 1
      call terr2(m+5)           !   terr1: m= 6
      write(*,*) '2:  m=',m     !   terr2: m= 777
      end                       !   2: m= 1
      subroutine terr2(m)       ! Как видно, при использовании в качестве
      write(*,*) 'terr1: m=',m  ! параметра выражения ФОРТРАН по умолчанию
      m=777                     ! как бы имитирует переход на режим
      write(*,*) 'terr2: m=',m  ! передачи по значению (адрес рабочей
                                ! ячейки, хранящей сумму, передается
      end                       ! формальному параметру)
```

4. **Передача параметра по адресу** может быть опасна и при успешном первом вызове функции, получающей верный результат. Так функция поиска изолированного корня методом деления отрезка пополам за счёт передачи параметра по адресу сузит отрезок с корнем до величины требуемой точности. Поэтому, если на исходном отрезке предстоит искать ещё корни других уравнений, то необходимо особо позаботиться о восстановлении его первоначальной длины, иначе рискуем искать корень там, где его нет, а при отсутствии должной внимательности есть шанс очень долго проискать ошибку в формулах, описывающих расчет левой части уравнения $\mathbf{f(x)=0}$, т.е. не там, где надо ее искать.

   При передаче аргумента по значению подобное невозможно в принципе.

5. Для моделирования в ФОРТРАНе передачи фактического аргумента по значению проще всего **не допускать изменения внутри тела процедуры соответствующего формального аргумента**. Если оно нужно по ходу дела, то можно заблаговременно переприсвоить значение формального аргумента некоторой рабочей переменной, не используя далее имя формального.

6. Если формальный аргумент – простая переменная, то вызов процедуры с обрамлением имени фактического аргумента круглыми скобками толкуется компилятором как необходимость сначала вычислить выражение с участием фактического аргумента и использовать найденное значение при ее работе. В этом случае ФОРТРАН как бы переходит на **передачу по значению**. Например, пусть подпрограмма **obmen**:

```
      subroutine obmen(a,b)
      implicit none
      real a, b, w
      write (*,*) '                obmen: a=', a,'  b=', b
      w=a; a=b; b=w
      write (*,*) '                obmen: a=', a,'  b=', b
      end
```

   обменивает содержимым две переменных типа **real**. Программа, которая при вызове **obmen** демонстрирует оба способа передачи: и **по значению**, и **по адресу** может выглядеть так:

```
      program testvalue
      implicit none
      real x, y
      x=5.0; y=3.0
      write(*,*) '  x=', x,'  y=', y
      call obmen((x),(y))             ! обращение с передачей по значению!
      write(*,*) '  x=', x,'  y=', y
      call obmen(x,y)
      write(*,*) '  x=', x,'  y=', y
      stop
      end
```

```
$ gfortran testvalue.for -c
$ gfortran obmen.for -c
$ gfortran testvalue.o obmen.o -testvalue
$ ./testvalue                       ! Результаты работы: несмотря на то,
  x=  5.  y=  3.                    ! что формальные аргументы обменялись
            obmen: a=  5.  b=  3. ! значениями фактические остались
            obmen: a=  3.  b=  5. ! неизменными. При заключении фактиче-
  x=  5.  y=  3.                    ! ского аргумента в круглые скобки
                                    ! ФОРТРАН толкует его как выражение,
                                    ! значение которого вычисляет.
                                    ! Формальному аргументу передается адрес
            obmen: a=  5.  b=  3. ! рабочей ячейки, хранящей найденное,
            obmen: a=  3.  b=  5. ! имитируя передачу фактического аргумента
  x=  3.  y=  5.                    ! по значению.
```

3.3.8 О чем узнали из третьего параграфа?

1. **ФОРТРАН** и **СИ** используют при вызове процедур разные схемы доступа к фактическим аргументам: ФОРТРАН – **по адресу**, СИ – **по значению**.

2. ФОРТРАН имитирует передачу параметра по значению, если фактическим параметром является выражение.

3. **При передаче параметра по адресу** имя фактического аргумента трактуется процедурой **как адрес** ячейки, с которой надо работать, и именно этот адрес передается формальному аргументу. ФОРТРАН, как правило, *в этом смысле понимает* имена фактического и формального аргументов.

4. **При передаче параметра по значению** имя фактического аргумента трактуется процедурой **как значение**, хранящееся по адресу фактического аргумента, которое сначала надо скопировать в соответствующий формальный аргумент, а уж затем начать обработку формального аргумента. СИ (в отличие от ФОРТРАНа) всегда *именно в этом смысле понимает* имена фактического и формального аргументов. В частности, даже изменение формального аргумента внутри тела СИ-функции не приведет к изменению фактического.

5. Для изменения содержимого фактического аргумента СИ-функции необходимо передать **адрес** фактического. При таком вызове СИ-функции перед именем фактического аргумента указывается значок **&** (операция взятия адреса).

6. Признак нацеленности формального аргумента на прием **адреса** — наличие значка ***** перед именем формального аргумента при описании последнего. В этом случае говорят, что по типу формальный аргумент является **указателем**.

7. Описание **double *p;** означает, что переменная **p** отводится для хранения адреса ячейки, в которой намечается хранить значение типа **double**. В операторе присваивания ***p=*p+5** правая часть означает: извлечь значение из ячейки, на которую указывает указатель **p** и сложить это значение с пятеркой; левая часть означает: поместить найденную сумму в ту же ячейку.

8. Обозначения подобные ***p** в контексте выполняемых операторов языка СИ часто называют **операцией разыменования указателя**.

9. В **C++** для возврата результата через фактический параметр можно использовать механизм указателей языка **СИ**. **C++** помимо доступа к фактическому аргументу через механизм указателей предоставляет родственный механизм передачи через **ссылку**. В этом случае имя формального параметра предваряется значком **&** (а не звездочкой, как в случае указателя), в теле функции не нужна операция разыименования, а соответствующий фактический параметр не требует операции получения его адреса.

10. Передача аргумента функции по значению никогда не может изменить значения фактического аргумента и, тем самым, свободна от наведения побочных эффектов, которыми бывает чревата передача аргумента по адресу.

3.3.9 Шестое домашнее задание.

Оформить решения задач **пятого домашнего задания** на языке **C++** через обращения к соответствующим функциям.

Тестирующие программы и тестируемые функции расположить в разных файлах.

Каждая тестирующая программа должна обращаться к трём функциям:

1. первая должна возвращать результат через свое имя;
2. вторая (типа **void**) – запоминать результат в фактическом аргументе, используя для доступа к нему в качестве соответствующего формального аргумента **указатель**.
3. третья (типа **void**) – запоминать результат в фактическом аргументе, используя для доступа к нему в качестве соответствующего формального аргумента **ссылку**.

4 Кое-что об утилите GNU make (часть первая).

4.1 Уяснение ситуации.

1. Исполнимый файл – это не только загрузочный.
2. Понятие об утилите make.

4.1.1 Исполнимый файл – это не только загрузочный

При отладке и пропуске программ предыдущей главы, когда приходилось в угоду преподавателю по отдельности компилировать главную программу и тестируемые функции, затем особо вызывать компоновщик для получения по опции **-o** загрузочного файла – почти наверняка хотелось сразу же получить последний:

1) или объединяя исходные тексты в одном файле (директивой **include**);

2) или указывая при вызове компилятора все необходимые исходные файлы.

Вся аргументация о невыгодности подобного подхода, хотя и осознавалась, тем не менее, казалась несколько надуманной – ведь набирать вместо трех команд одну **gfortran tstsub.for -o tstsub** или **gfortran tstsub.for subfun.for -o tstsub** проще. При наличии же в текущей директории соответствующих объектных файлов (**tstsub.o** и **subfun.o**) возможен и вызов **gfortran tstsub.o subfun.o -o tstsub** без затрат времени на перекомпиляцию.

Если объектных файлов много, то вызов (или вызовы) **gfortran** удобно записать в **особый отдельный командный файл** (скрипт, сценарий), который для исполнения можно было бы вызвать по имени как новую **UNIX**-команду, и в котором наряду с вызовами нужных команд присутствовали бы и имена необходимых файлов. С примерами скриптов уже знакомились в параграфах **1.8.7** – **1.8.10**.

По форме скрипт не отличается от обычного текстового файла, который можно редактировать, хотя по статусу должен быть **исполнимым** (как получаемый компоновщиком при сборке объектных **загрузочный файл**).

Поместим в файл **tstsfun** команды из параграфа **3.2.2**, готовящие загрузочный файл программы, тестирующей подпрограмму **subfun**.

```
                                    # Файл tstsfun . Его вызов ./tstsfun
gfortran -c tstsfun.for             # приведет к выводу сообщения
gfortran -c subfun.for              # $ bash: ./tstsfun: Permission denied
gfortran tstsfun.o subfun.o -o tstsfun # Причина - у файла tstsfun нет статуса
                                    # исполнимого файла.
```

Изменить статус на нужный можно, например, командой **chmod a+x tstsfun** (от change mode – изменить режим файла; подробно о возможностях и синтаксисе **chmod** см. в курсе **„Операционные системы”**). Теперь запуск новой команды **tstsfun** из текущей директории возможен. Несмотря на несомненную выгоду по сравнению с ручным набором команд, наш скрипт **tstsfun** будет перекомпилировать все указанные в нем исходные файлы и перекомпоновывать соответствующие объектные вне зависимости от целесообразности этих действий.

4.1.2 Понятие об утилите make

В процессе отладки и тестирования при сборке загрузочного файла наряду с объектными файлами ФОРТРАН- или СИ-функций, давно отлаженных и протестированных, используются исходные тексты главной программы и, возможно, некоторых новых процедур. Ясно, что перекомпилировать нужно лишь те исходные файлы, в которые вносились изменения, не тратя время на перекомпиляцию остальных. Поэтому чрезвычайно выгодно наличие в математическом обеспечении утилиты, способной

1) учитывать зависимости одних файлов от других;

2) осуществлять только необходимую перекомпиляцию;

3) получать загрузочный файл и файлы с результатами;

4) выполнять некоторые полезные для программиста дополнительные действия.

Такая утилита есть практически во всех **UNIX**-подобных системах [20], [21], [22, глава GNU Make]. Это – программа-координатор, вызываемая командой **make**. Инструкции для себя она ищет в файле с именем **Makefile**, в котором программист **на языке утилиты** может определить более оптимальный режим сборки программного продукта нежели, например, **bash**-скрипт из предыдущего параграфа. Искомым программным продуктом в данном случае (по задумке), то есть **главной целью** сборки является **загрузочный файл** с именем **tstsfun**. Это имя при записи первой строки **Make**-файла можно указать в качестве имени **главной цели** нашего проекта. Поскольку **tstsfun** получается из объектных файлов **tstsfun.o** и **subfun.o**, то достижение цели **tstsfun** непосредственно **зависит** от их наличия. Отражение факта этой зависимости можно записать в первой строке **Make**-файла:

```
tstsfun:   tstsfun.o subfun.o
```

Здесь **tstsfun** – имя **главной цели**, отделяемое от **списка зависимостей** (имён объектных файлов, из которых должен компоноваться загрузочный файл **tstsfun**) **двоеточием**. Во второй строке **Make**-файла поместим **команду**

```
[  TAB  ]  gfortran tstsfun.o subfun.o -o tstsfun
```

которая из указанных **зависимостей** компонует **главную цель**. Признаком того, что это именно **команда оболочки**, обеспечивающая достижение **цели**, а не какая-то другая инструкция **Make**-файла, служит наличие перед командой кода табуляции (в зависимости от настройки редактора – одно или два нажатия клавиши **TAB**). Редактор **mcedit** подкрашивает поле признака такой **команды** в **Make**-файле красным цветом. В итоге содержимое **Make**-файла, компонующего загрузочный файл из указанных двух объектных, в простейшем случае имеет вид:

```
  tstsfun : tstsfun.o subfun.o                    # значок "диез" - признак
[  TAB  ]  gfortran tstsfun.o subfun.o  -o tstsfun # начала комментария
```

1. Обозначение **[TAB]** соответствует полю, занятому кодом табуляции.

2. Запуск утилиты **make** при отсутствии в текущей директории файла **tstsfun** и наличии файлов **tstsfun.o** и **subfun.o**

```
 gfortran tstsfun.o subfun.o -o tstsfun # высветит на экран отработавшую
$ make                                  # команду.
make: 'tstsfun' не требует обновления.  # Результат повторного вызова make.
```

   т.е. утилита *обнаружила*, что время последнего создания **tstsfun** более позднее нежели времена создания объектных файлов, и перекомпоновка не нужна.

3. Если дать команду **touch subfun.o**, которая изменит время модификации файла **subfun.o** на текущее (более позднее, чем время прежней компоновки файла **tstsfun**), то новый запуск **make** приведет к замене старой версии файла **tstsfun**. Изменить время модификации можно и принудительным сохранением файла при работе в редакторе), хотя посредством команды **touch** проще.

4. Вообще, имя цели не обязано совпадать с именем файла, получающегося в результате отработки команды, реализующей достижение цели. Например, при содержимом **Make**-файла

```
mytest    : tstsfun.o subfun.o
[  TAB  ]  gfortran tstsfun.o subfun.o -o tstsfun
```

   каждый запуск утилиты **make** независимо от времени модификации объектных кодов производит перекомпоновку файла **tstsfun**, что не согласуется с первоначальной идеей экономии времени. В данном случае получается, что цель **mytest** формально никогда не может быть достигнута (т.е. воплотиться в файл с таким именем), поскольку файл **mytest** никогда не будет создан.

5. Если по каким-то причинам в текущей директории не будет файла **subfun.o**, то вызов **make** завершится сообщением

```
make: *** Нет правила для сборки цели 'subfun.o', требуемой для
          'tstsfun'. Останов.
```

6. Утилита **make** может в качестве аргумента использовать файл с именем отличным от **Makefile**, если при вызове **make** включена опция **-f** с указанием после неё имени файла, замещающего **Make**-файл (например, **make -f mymake**).

Таким образом, на очень простой задаче компоновки из двух объектных файлов загрузочного познакомились с различными вариантами работы утилиты **make**, когда **Make**-файл состоит всего из одного **правила**, задающего

1) имена цели и файлов, от которых непосредственно зависит ее достижение;

2) и команды, которая обеспечивает это достижение.

Правда, пока посредством нашего **make**-файла нельзя выполнить полностью ту работу, которую делал **bash**-скрипт из параграфа **4.1.1**. Именно: наш **make**-файл не получает объектные модули из исходных.

4.2 Первые усовершенствования первого make-файла.

Добавим в полученный **make**-файл ещё два **правила**: первое с описанием цели **tstsfun.o** получения объектного файла из исходного **tstsfun.for** и указанием команды вызова компилятора для ее достижения; второе – с описанием решения аналогичной задачи для получения файла **subfun.o**.

```
   tstsfun : tstsfun.o subfun.o
[  TAB ]  gfortran tstsfun.o subfun.o -o tstsfun
tstsfun.o: tstsfun.for
[  TAB ]   gfortran -c tstsfun.for
 subfun.o: subfun.for
[  TAB ]  gfortran -c subfun.for
```

Здесь

1) **tstsfun.o** слева от двоеточия – имя дополнительной цели **make**-файла;

2) после двоеточия – список **зависимостей**, т.е. имен исходных файлов, от которых зависит достижение цели (сейчас такой файл один – **tstsfun.for**);

3) в следующей строке располагается команда достижения цели (вызов компилятора **gfortran**, нацеленный на получение объектного файла **tstsfun.o**);

4) очередная пара строк содержит описание цели **subfun.o** и команды, получающей объектный файл **subfun.o**.

Теперь наш **make**-файл способен полностью выполнять ту же работу, что и **bash**-скрипт из параграфа **4.1.1**, но более оптимальным образом. Поэкспериментируем с новым **make**-файлом, используя при необходимости утилиту **touch**.

1. ```
   $ touch *.for                      # Моделируем изменение исходных файлов.
   $ make                             # Вызываем утилиту make, которая
   gfortran -c tstsfun.for            # создаёт файлы         tstsfun.o,
   gfortran -c subfun.for                                  #  subfun.o
   gfortran tstsfun.o subfun.o -o tstsfun                  #  tstsfun.
   $ ./tstsfun                                             # Вызываем tstsfun
    fun( 0.5100000000000000D+00)=-0.1980000000000000D+01  # Получаем
    fun( 0.1000000000000000D+01)=-0.1000000000000000D+01  #    на
    fun( 0.7500000000000000D+00)=-0.1500000000000000D+01  #  экране
    fun( 0.1500000000000000D+01)= 0.0000000000000000D+00  # результаты.
    fun( 0.1499999999999999D+01)=-0.2220446049250313D-14
   ```

   Сейчас работа утилиты просто повторяет работу **bash**-скрипта из **4.1.1**.

2. Однако при повторном запуске становится видно преимущество **make**, которая

   ```
   $ make                                  # сообщает, что нет нужды в
   make: ‘tstsfun’ не требует обновления.  # перекомпиляции и перекомпоновке.
   ```
   ```

3.
```
$ touch subfun.for                    # Смоделировали изменение subfun.for.
$ make                                # При вызове make перекомпилирует
gfortran -c subfun.for                # только измененный исходный файл.
gfortran tstsfun.o subfun.o           #
$
$ touch tstsfun.for                   # Моделируем изменение tstsfun.for
$ make                                # и еще раз убедились, что make
gfortran -c tstsfun.for               # не тратит время на компиляцию
gfortran tstsfun.o subfun.o           # неизмененного исходного файла.
```

4.
```
$ make tstsfun.o                        # Демонстрируем, что make можно
make: ‘tstsfun.o’ не требует обновления. # вызывать и для выполнения
$ touch subfun.for                      # конкретной второстепенной
$ make subfun.o                         # цели, указывая ее имя
gfortran -c subfun.for                  # в качестве аргумента утилиты.
$ make                                  #
gfortran tstsfun.o subfun.o -o tstsfun
```

5.
```
  tstsfun : tstsfun.o subfun.o                    # Для перекомпиляции
[  TAB ]    gfortran tstsfun.o subfun.o -o tstsfun # всех исходников
tstsfun.o : tstsfun.for                         # можем описать новую
[  TAB ]    gfortran -c tstsfun.for             # дополнительную цель
 subfun.o : subfun.for                          #
 [  TAB ]    gfortran -c subfun.for             # touch, реализуемую
  touch   :                                     # одноименной командой.
[  TAB ]    touch *.for
```

   Некоторые варианты пропуска этого **make**-файла:

```
$ make touch                    # Вызов make с указанием цели touch.
touch *.for                     # Отработка этого вызова.
$ make                          # Видим, что make работает по
gfortran -c tstsfun.for         # полной программе, ибо предыдущий
gfortran -c subfun.for          # вызов make touch изменил времена
gfortran tstsfun.o subfun.o     # модификации исходных файлов.
```

   Так что **имя цели** в **make**-файле может быть не только именем файла, но и именем (придуманным нами) некоторого действия, которое реализуется подходящей по смыслу командой. Цели, имена которых не являются именами файлов часто называют **псевдоцелями** (pseudotargets) или **абстрактными целями** (phony targets) [22], [20].

6. Полезным примером использования **абстрактной цели** может служить цель уничтожения в текущей директории объектных и загузочного файлов. При выполнении этой операции вручную НЕТ гарантии того, что думая про **rm *.o**, не наберём **rm *.*** и, не заметив *кошмарной очипатки*, не нажмём на ввод. Аналогичная ситуация вполне может возникнуть и при уничтожении файлов посредством клавиш среды **mc** при ошибочной пометке. Наличие отлаженного **make**-файла такую гарантию дает.

```
  tstsfun : tstsfun.o subfun.o                      # Опция -f в команде
[  TAB  ]   gfortran tstsfun.o subfun.o -o tstsfun # реализации псевдоцели
tstsfun.o : tstsfun.for                           # rmoout гасит реакцию
[  TAB  ]   gfortran -c tstsfun.for               # утилиты rm, если в
 subfun.o : subfun.for                            # текущей директории НЕТ
[  TAB  ]   gfortran -c subfun.for                # указываемых файлов.
   rmoout :                          #
[  TAB  ]   rm -f *.o tstsfun       # При пропуске этого make-файла ИМЕЕМ:
$ make rmoout
rm *.o tstsfun
$ make
gfortran -c tstsfun.for
gfortran -c subfun.for
gfortran tstsfun.o subfun.o -o tstsfun
```

7. В пунктах **5** и **6** после имен псевдоцелей **touch** и **rmoout** не указан список зависимостей как в случае описания целей **tstsfun**, **tstsfun.o**, **subfun.o**. Дело в том, что без наличия файлов **tstsfun.o** и **subfun.o** принципиально невозможно построить загрузочный файл **tstsfun** (невозможно достичь цели **tstsfun**). Аналогично без наличия исходного файла **tstsfun.for** невозможно получить объектный файл **tstsfun.o** и без исходного файла **subfun.for** не достичь цели **subfun.o**. Поэтому при определении целей **tstsfun**, **tstsfun.o** и **subfun.o** необходимо указать список имен файлов, без которых цели нельзя достичь.

   Достижение абстрактных целей **touch** и **rmoout** не требует с необходимостью наличия конкретных файлов. Если же опишем цель **rmo : tstsfun.o subfun.o**, то процесс ее достижения напомнит анекдот о сведении задачи кипячения воды в наполненном чайнике к решению той же задачи при пустом:

```
$ make rmo                # При наличии объектных модулей все пройдет
rm -f  *.o                # по задумке.
$ make rmo                # Если же их уже нет, то наличие зависимости
gfortran -c subfun.for    # потребует их создания исключительно для
gfortran -c tstsfun.for   # последующего удаления (вряд ли это разумно).
rm -f  *.o                #
```

8. При указании зависимости для одной абстрактной цели можно указать имя другой абстрактной цели. Пусть нас не устраивает совместное удаление объектных и загрузочного файлов одной командой **rm -f *.o tstsfun**, а требуются две возможности: 1) удалить только объектные файлы; 2) удалить и объектные и и загрузочный. Реализовать это можно по разному:

```
     rmo :                        #      rmo :
[  TAB  ]  rm -f *.o              # [ TAB  ]   rm -f *.o
  rmoout : rmo                    #  rmoout  :
[  TAB  ]  rm -f  tstsfun         #[  TAB  ]   rm -f *.o rm -f *.out
```

9. Наш **make**-файл можем дополнить абстрактной целью пропуска программы:

```
   tstsfun : tstsfun.o subfun.o                    # Запуск программы
[  TAB  ]  gfortran tstsfun.o subfun.o -o tstsfun  # можно инициировать
tstsfun.o: tstsfun.for                             # командой make exe.
[  TAB  ]  gfortran -c tstsfun.for
subfun.o : subfun.for              #  Для вывода результата не на экран
[  TAB  ]  gfortran -c subfun.for  #  а в файл myres.dat достаточно
     rmo :                         #  дать команду
[  TAB  ]  rm -f *.o               #              make exe > myres.dat
  rmoout : rmo
[  TAB  ]  rm -f  tstsfun
     exe :
[  TAB  ]  ./tstsfun
$ make exe
./tstsfun
 fun( 0.5100000000000000D+00)=-0.1980000000000000D+01
 fun( 0.1000000000000000D+01)=-0.1000000000000000D+01
 fun( 0.7500000000000000D+00)=-0.1500000000000000D+01
 fun( 0.1500000000000000D+01)= 0.0000000000000000D+00
 fun( 0.1499999999999999D+01)=-0.2220446049250313D-14
```

10. Подключение цели **exe** из пункта **9** — небезупречно, так как после модификации исходного кода вызов **make exe** не приведет к перекомпоновке загрузочного кода. Поэтому, внося изменения в исходный код, важно не забывать перед запуском **make exe** особо выполнять главную цель. Это очень неудобно.

11. Для устранения последнего недостатка можно при описании абстрактной цели **exe** указать ее зависимость от главной цели, а значит и от входящих в нее зависимостей от времен создания исходных и объектных файлов:

```
  tstsfun: tstsfun.o subfun.o
[  TAB  ]  gfortran tstsfun.o subfun.o -o tstsfun
tstsfun.o: tstsfun.for
[  TAB  ]  gfortran -c tstsfun.for
subfun.o : subfun.for
[  TAB  ]  gfortran -c subfun.for
     rmo :
[  TAB  ]  rm -f *.o
  rmoout : rmo
[  TAB  ]  rm -f  *.out
     exe : tstsfun                #  Подправка цели    exe
[  TAB  ]  ./tstsfun              #  зависимостью от главной цели.
```

Теперь вызов **make exe** *чувствует* необходимость перекомпоновки загрузочного кода, если исходный код был модифицирован:

```
$ touch *.for              # Моделируем модификацию исходных текстов.
$ make exe                 # Теперь цель exe перед запуском загрузочного
gfortran -c tstsfun.for    # файла  перекомпилирует  тексты  исходных и
gfortran -c subfun.for     #                      перекомпонует объектные,
gfortran tstsfun.o subfun.o  -o tstsfun  #       получая загрузочный файл
./tstsfun                                #       самой   свежей   версии,
 fun( 0.5100000000000000E+00)=-0.1980000000000000E+01  #     с  выводом
 fun( 0.1000000000000000E+01)=-0.1000000000000000E+01  #     результатов
 fun( 0.7500000000000000E+00)=-0.1500000000000000E+01  #     расчета на
 fun( 0.1500000000000000E+01)= 0.0000000000000000E+00  #     экран.
 fun( 0.1499999999999999E+01)=-0.2220446049250313E-14
$ make exe                                             # Повторный вызов
./tstsfun                                              #    make exe
 fun( 0.5100000000000000E+00)=-0.1980000000000000E+01  # уже не требует
 fun( 0.1000000000000000E+01)=-0.1000000000000000E+01  # перекомпиляции
 fun( 0.7500000000000000E+00)=-0.1500000000000000E+01  # исходных кодов
 fun( 0.1500000000000000E+01)= 0.0000000000000000E+00  # и перекомпоновки
 fun( 0.1499999999999999E+01)=-0.2220446049250313E-14  # объектных
```

12. Обычно результаты работы программы выводить только на экран не очень удобно. Для осмысления или использования в качестве исходных данных другими программами результаты выгодно помещать в файл, который можно посмотреть посредством какого-нибудь вьюера или редактора. Однако для тренировки можно и в **make**-файл включить псевдоцель вывода результата на экран. Для уяснения ситуации рассмотрим работу следующего **make**-файла:

```
  tstsfun : tstsfun.o subfun.o
[  TAB  ]  gfortran tstsfun.o subfun.o -o tstsfun
tstsfun.o : tstsfun.for
[  TAB  ]  gfortran -c tstsfun.for
 subfun.o : subfun.for
[  TAB  ]  gfortran -c subfun.for
     rmo :
[  TAB  ]  rm -f  *.o
   rmoout : rmo
[  TAB  ]  rm -f *.out
     exe : tstsfun
[  TAB  ]  ./tstsfun > tstsfun.res
 tstsfun.res : tstsfun                # Цель - перерасчёт результата лишь при
[  TAB  ]    ./tstsfun > tstsfun.res  # модификации  (исходных или объектных
                                      # файлов), которая выполнена после
                                      # получения предыдущего результата.
   resold :                    # Вывод старого результата (если он не стерт).
[  TAB  ]  cat tstsfun.res     #
   resold1: tstsfun            # Вывод старого результата. Загрузочный файл
[  TAB  ]   cat tstsfun.res    # при необходимости будет получен новый, но
                               # не вызовется для нового расчёта.
   resnew : tstsfun.res        # Всегда высвечивает результат самой свежей
[  TAB  ]   cat tstsfun.res    # версии программы. При необходимости
                               # загрузочный файл модифицируется и
                               # выполняется перед выводом результата.
```

Демонстрация работы последнего make-файла:

```
$ rm -f *o *.out                    # Уничтожили объектные и загрузочный коды
$ make                              # Обращаемся к главной цели:
gfortran -c tstsfun.for             #
gfortran -c subfun.for              #
gfortran tstsfun.o subfun.o         # получаем загрузочный код программы.
$ make                              # Повторное обращение к главной цели
make: ‘a.out’ не требует обновления.# приводит к сообщению о его ненужности
$ make resold                       # Попытка вывести содержимое файла,
cat tstsfun.res                             # которого нет приводит, к
cat: tstsfun.res: No such file or directory  # аварийному завершению
make: *** [resold] Ошибка 1                 # целей resold и resold1,
$ make resnew          # Однако, вызов цели  resnew с задачей справляется:
./a.out > tstsfun.res  # отрабатывает загрузочный код, создавая нужный файл,
cat tstsfun.res        # и содержимое последнего высвечивается на экран.
 fun( 0.5100000000000000E+00)=-0.1980000000000000E+01
 fun( 0.1000000000000000E+01)=-0.1000000000000000E+01
 fun( 0.7500000000000000E+00)=-0.1500000000000000E+01
 fun( 0.1500000000000000E+01)= 0.0000000000000000E+00
 fun( 0.1499999999999999E+01)=-0.2220446049250313E-14
$ touch *.for                  # Смоделировали модификацию всех исходных кодов
$ make resnew                  # Вызов цели resnew опять справляется:
gfortran -c tstsfun.for        # генерируются новые объектные коды,
gfortran -c subfun.for         #
gfortran tstsfun.o subfun.o    # генерируется новый загрузочный код,
./a.out > tstsfun.res          # который создает файл с результатом и
cat tstsfun.res                # последний высвечивается:
 fun( 0.5100000000000000E+00)=-0.1980000000000000E+01 ... ... ...
$ make exeold          # Вызов цели exeold приводит только к отработке
./a.out > tstsfun.res  # имеющегося загрузочного модуля так, что,
touch *.for            # если модифицировать исходные коды, то
$ make exeold          # цель exeold не создаст новый ./a.out, а
./a.out > tstsfun.res  # заставит отработать его старый вариант.
$ make exenew               # Демонстрируется, что псевдоцель exenew
gfortran -c tstsfun.for     # чувствует ситуацию так же, как и resnew,
gfortran -c subfun.for      #  получая свежие версии и объектных файлов,
gfortran tstsfun.o subfun.o #                        и загрузочного файла,
./a.out > tstsfun.res       #                        и файла-результата.
$ touch *.o                 # Как бы модифицировали объектные коды.
$ make exeold               # exeold не распознает этого и просто
./a.out > tstsfun.res       # вызывает старый загрузочный модуль.
$ make exenew               # Исходники не перекомпилируются, но из объектных
gfortran tstsfun.o subfun.o # получается новый загрузочный и по нему свежий
./a.out > tstsfun.res       # результат. Вывод результата не требуется.
$ touch *.out          # Как бы изменили загрузочный код. exenew не будет
$ make exenew          # его перекомпоновывать (ведь он и так самый свежий),
./a.out > tstsfun.res  # а просто инициирует его работу, получая результат.
```

Некоторое обсуждение.
Несмотря на простоту нашей программы оказалось, что **make**-файл для нее сравним с ней по объему исходного текста. Правда, в нем описано довольно много странных абстрактных целей. Например,

1. **make resold**. Зачем загромождать **make**-файл требованием вывода результата на экран? Если нужно, удобнее воспользоваться возможностями утилиты **mc**?

2. **make resold1**. Вряд ли кому-то захочется увидеть на экране результат старой версии программы, хотя к моменту его вывода уже создан новый загрузочный файл, который еще не отработал.

3. **make rmo**. Зачем вообще удалять объектные и загрузочный файлы, если ими намерены пользоваться при дальнейшей работе.

4. **make exeold**. Имеет ли смысл проводить расчет по старой версии загрузочного файла, если менялись исходные тексты. Если же они не менялись, то можно воспользоваться и **make exenew**, которая в случае модификации автоматически перекомпонует загрузочный файл и выполнит по нему надлежащий расчет.

5. **make exenew**. Если сравнить ее описание с целью **make tstsfun.res**, то увидим, что и список зависимостей, и командная часть у этих двух частных целей – одинаковы. Единственное их отличие в статусе. **exenew** – псевдоцель, так как не являентся именем файла. **tstsfun.res** – настоящая цель. А раз так, то она имеет перед **exenew** одно преимущество. Именно, вызов цели **make tstsfun.res** в случае, когда файл-результат **tstsfun.res** уже был получен ранее, просто выведет на экран сообщение о нецелесообразности повторного вызова загрузочного кода, если вместо этого сразу можно посмотреть, ранее полученный им результат.

Все упомянутые здесь псевдоцели послужили просто примерами для уяснения характера работы утилиты **make**. Теперь можем резко сократить текст **make**-файла:

```
    tstsfun : tstsfun.o subfun.o
[  TAB  ]     gfortran tstsfun.o subfun.o -o tstsfun
  tstsfun.o : tstsfun.for
[  TAB  ]     gfortran -c tstsfun.for
   subfun.o : subfun.for
[  TAB  ]     gfortran -c subfun.for
      clear :
[  TAB  ]     rm -f  *.o tstsfun
tstsfun.res : tstsfun                # Цель - получение файла с результатом только
[  TAB  ]    ./tstsfun > tstsfun.res  # в том случае, если модификация кодов выполнена
                                      # после получения предыдущего результата.
   resnew : tstsfun.res     # Всегда высвечивает результат свежей версии программы.
[  TAB  ]   cat tstsfun.res # При необходимости загрузочный код модифицируется
                            # и выполняется перед выводом результата.
```

Сохраним последнюю цель, поскольку она все-таки позволяет сразу увидеть результат. Если его объем превышает размер экрана, то можно использовать или другие вьюеры, или даже какой-нибудь редактор, тот же **mcedit**, например.

4.3 Еще один элементарный make-файл для простой задачи.

1. Текст главной программы на ФОРТРАНе и пояснения.
2. Включение имен файлов с данными в зависимости make-файла.
3. Проверка работы make-файла. Текстовый файл с результатом.
4. GNUPLOT-скрипт выдачи результата в виде графика.
5. Тестирование псевдоцели высветки рисунка.

Пусть функция **fun.for** и подпрограмма **subfun.for** нас устраивают. Требуется посредством их вызова для значений аргумента **x**, равномерно распределенных по промежутку **[a,b]**, **табулировать** функции $(\mathbf{2x-3})/\mathbf{2}$ и $(\mathbf{2x-3})/\mathbf{4}$. Число участков дробления **n**, значения **a** и **b** главная программа должна вводить из файла **probtask.par**, а результат расчета – выводить в файл **result** в виде таблицы, каждая строка которой содержит номер точки дробления с соответствующими значениями аргумента и упомянутых функций. Необходимо, чтобы без каких-либо изменений файл **result** мог использоваться утилитой **gnuplot** для построения соответствующих графиков.

4.3.1 Текст главной программы на ФОРТРАНе и пояснения.

```
      program probtask      ! Модель проблемной программы.  Файл probtask.for
      include 'probtask.hdr'!.............  Подключение описаний.
      open(unit=ninp,file='probtask.par')            ! Открытие файла  ввода.
      open(unit=nres,file='result',status='replace') ! Открытие файла вывода.
      read(ninp,100) n,a,b                  ! Ввод числа узлов и границ отрезка.
      write(nres,110) n, a, b               ! Контрольная печать введенного.
      h=(b-a)/(n-1)                         ! Расчет шага дробления [a,b].
      write(nres,1000)                      ! Печать заголовка первой таблицы.
      do i=1, n                             ! Для каждого из узлов
        x=a+(i-1)*h                         !   вычисляем текущий аргумент,
        y=fun(x)/2                          !   фиксируем результат fun(x),
        call subfun(x,z)                    !   получаем результат subfun
        write(nres,1001) i, x, y, z/4       !   вывод текущего результата
      enddo                                 !
      close(nres)                           ! Закрытие файла результата.
  100 format(i10/d10.3/d10.3)               ! Список форматов ввода-вывода:
  110 format(1x,'#  Число узлов дискретизации (n)=',i4/
     >       1x,'# Левая граница промежутка  (a)=',d23.16/
     >       1x,'# Правая граница промежутка (b)=',d23.16)
 1000 format(1x,'# N',15x,'x',21x,'fun(x)/2',19x,' z/4 ')
 1001 format(1x,i3,2x,d23.16,2x,d23.16,2x,d23.16)
      end
```

1. **2-я строка.** Через указание **include** в текст программы включено описание имен, используемых в ней. Конечно, можно, не используя файл **probtask.hdr**, просто явно описать их. Однако, такой подход хорош при работе с небольшими программами. Когда описания обширны, то они резко снижают наглядность алгоритма. Сейчас вся программа уместилась менее чем на странице, а при необходимости можно просто заглянуть в файл **probtask.hdr**:

```
c Переменные главной программы                    :   Файл probtask.hdr
      implicit none                    ! Отмена правила умолчания ФОРТРАНа.
      integer ninp / 5 /, nres / 6 / ! Номера файлов ввода и вывода.
      integer i                        ! Номер текущего узла.
      integer n                ! Число узлов дробления промежутка [a,b].
      real*8 a, b, h           ! Границы [a,b] и шаг его дискретизации.
      real*8 x, y, z           ! Текущие аргумент и результаты fun и subfun.
      real*8 fun               ! Тип значения возвращаемого функцией fun.
```

2. **3-я строка.** Оператор **open(unit=ninp,file='probtask.par',status='old')** означает, что к формальному устройству с номером **ninp** подсоединяется файл **probtask.par**, который к моменту вызова исполнимого кода должен быть сформирован в текущей директории, чтобы оператор **read(ninp,...)** мог прочесть из **probtask.par** данные, подготовленные для ввода.

3. **4-я строка.** Программный номер **nres** сопоставляется файлу с именем **result**, в который намечаем вывод результатов работы программы. Спецификатор статуса **'replace'** означает, что открываемый файл замещает существующий.

4. **5- строка.** Оператор **read(ninp,100) n,a,b** читает значения переменных из файла с программным номером **ninp** под управлением оператора **format**. Обычно оператор **format** считается неудобным, так как расположение данного в конкретных позициях строки требует от человека почти безошибочной работы. На самом деле фиксация места чтения – ДОСТОИНСТВО: в остальной части вводимой строки можно поместить комментарии, например, так:

```
        9<--= (n) Число узлов дискретизации.    :   Файл probtask.par
 2.100+00<--= (a) Левая абсцисса промежутка.    :   -----------------
 3.100+00<--= (b) Правая граница промежутка     :.....................
```

   Без знания смысловой нагрузки и очередности данных вероятность ошибится при их наборе гораздо выше чем при наличии очевидного примера их записи.

5. **6-я строка.** Контрольная печать вводимых данных объективно обоснована:

   1) для проверки значений введенных параметров (вдруг случайно ошиблись при наборе данных или, не заметив, перепутали их очередность);
   2) для анализа получаемых результатов (разумно ли их анализировать без документального подтверждения значений исходных параметров).

   Контрольная печать и печать заголовка таблицы начинаются со значка „диез“ (#). Включение его в печатаемый текст вызвано намерением применить в дальнейшем для построения графиков функций утилиту **gnuplot**. По синтаксису утилиты „диез“ – признак начала комментария. Поэтому строки, начинающиеся с него, **gnuplot** игнорирует, то есть они не мешают работе утилиты. А при выводе таблицы на экран эти строки могут содержать, например, заголовок таблицы, указывающий, что в каком столбце находится:

```
#  Число узлов дискретизации (n)=   5
# Левая граница промежутка  (a)= 0.0000000000000000E+00
# Правая граница промежутка (b)= 0.1000000000000000E+01
# N               x                     fun(x)/2                    z/4
  1   0.0000000000000000E+00  -0.1500000000000000E+01  -0.7500000000000000E+00
  2   0.2500000000000000E+00  -0.1250000000000000E+01  -0.6250000000000000E+00
  3   0.5000000000000000E+00  -0.1000000000000000E+01  -0.5000000000000000E+00
  4   0.7500000000000000E+00  -0.7500000000000000E+00  -0.3750000000000000E+00
  5   0.1000000000000000E+01  -0.5000000000000000E+00  -0.2500000000000000E+00
```

6. **13-я строка.** Закрытие файла, в который выводился результат.

4.3.2 Включение имен файлов с данными в зависимости make-файла.

```
   tstsfun : tstsfun.o subfun.o                    # Слева make-файл, предложенный
[  TAB  ]  gfortran tstsfun.o subfun.o -tstsfun    # в завершении пункта 4.2.
tstsfun.o: tstsfun.for                             # Модифицируем его для решения
[  TAB  ]  gfortran -c tstsfun.for                 # задачи, изложенной в пункте 4.3.1.
subfun.o : subfun.for
[  TAB  ]  gfortran -c subfun.for
   clear :
[  TAB  ]  rm -f  *.o *.out
 tstsfun.res : tstsfun
[  TAB  ]      ./a.out > tstsfun.res
   resnew : tstsfun.res
[  TAB  ]   cat tstsfun.res
```

1. Назовём главную цель нового **make**-файла именем **probtask**.

2. Включим имя файла с объектным кодом функции **fun.o** в список зависимостей главной цели и в аргументы команды компоновки, не забыв указать после опции **-o** и имя файла **probtask** с получаемым загрузочным кодом.

3. Для получения файла **fun.o** добавим цель **fun.o**. Ее достижение требует наличия в текущей директории файла **fun.for** с исходным кодом функции **fun**.

4. Программе тестирования подпрограммы **subfun.for** из пункта **4.2** ввод не требовался (она только печатала результат). Программа же, решающая настоящую задачу, по условию должна вводить параметры из файла **probtask.par**. Поэтому необходимо указать, что достижение цели **result** зависит от наличия не только загрузочного файла **probtask**, но и файла **probtask.par**, хранящего вводимые данные и подготовленного заранее в текущей директории.

5. Цель **resnew make**-файла из пункта **4.2** заменим двумя новыми псевдоцелями:

   restab – вывод на экран результата расчета в виде числовой таблицы;
   resplt – обзор графиков табулированных функций (через утилиту **gnuplot**).

   Именно ради подключения этой цели мы и устроили себе ранее тренировку, описав цель **resnew**. Пояснения к содержимому **gnuplot**-скрипта **probtask.gnu** даются в пункте **4.3.4**. Таким образом, модифицированный для новой задачи **make**-файл в первом приближении имеет вид:

```
    probtask : probtask.o subfun.o fun.o
[  tab  ]       gfortran probtask.o subfun.o fun.o -o probtask
  probtask.o : probtask.for
[  TAB  ]       gfortran -c probtask.for
    subfun.o : subfun.for
[  TAB  ]       gfortran -c subfun.for
       fun.o : fun.for
[  TAB  ]       gfortran -c fun.for
       clean :
[  TAB  ]       rm -f  *.o *.out
      result : probtask probtask.par
[  TAB  ]       ./probtask
      restab : result                # Цели restab и resplt
[  TAB  ]       cat result             # зависят от probtask.par
      resplt : result                # неявно - через имя цели
[  TAB  ]       gnuplot probtask.gnu   #                  result
```

4.3.3 Проверка работы make-файла. Текстовый файл с результатом.

1. ```
   $ make                                  # Здесь демонстрируется отработка
   gfortran -c probtask.for                # главной цели:
   gfortran -c subfun.for                  #.........  сборка загрузочного файла,
   gfortran -c fun.for                             #  когда в текущей директории
   gfortran fun.o probtask.o subfun.o -o probtask #  нет объектных файлов
   $ make                                       # Если теперь снова вызвать make
   make: ‘probtask’ не требует обновления.     # то получим ожидаемое сообщение.
   ```

2. ```
   $ make result    #  Проверка работы make-файла по цели result.
   ./probtask       #  Если файл probtask.par не модифицировался, то вторичный
   $ make result                           # вызов цели result не приводит к
   make: ‘result’ не требует обновления.  # повторному вызову загрузочного файла.
   $ touch *.par                           # Моделируем модификацию probtask.par.
   $ make result                           # Цель result адекватно реагирует
   ./probtask                              # на неё, вызывая загрузочный код.
   $ touch fun.for                         # Моделируем изменение fun.for
   $ make result        # Налицо адекватная реакция: перекомпилирован только
   gfortran -c fun.for                               #  изменённый исходник,
   gfortran fun.o probtask.o subfun.o -o probtask  # собран загрузочный файл
   ./probtask                                        # и получен новый результат.
   ```

3. ```
   $ make result                           # Проверка make-файла по цели restab.
   make: ‘result’ не требует обновления.  # Если результат не требует обновления,
   $ make restab                           # то make restab просто высвечивает
   cat < result                            # содержимое файла result:
    #  Число узлов дискретизации (n)=   5
    # Левая граница промежутка  (a)= 0.0000000000000000E+00
    # Правая граница промежутка (b)= 0.1000000000000000E+01
    # N               x                     fun(x)/2                  z/4
      1   0.0000000000000000E+00  -0.1500000000000000E+01  -0.7500000000000000E+00
      2   0.2500000000000000E+00  -0.1250000000000000E+01  -0.6250000000000000E+00
      3   0.5000000000000000E+00  -0.1000000000000000E+01  -0.5000000000000000E+00
      4   0.7500000000000000E+00  -0.7500000000000000E+00  -0.3750000000000000E+00
      5   0.1000000000000000E+01  -0.5000000000000000E+00  -0.2500000000000000E+00
   ```
   ```

```
$ make restab             # Если же изменили содержимое файла probtask.par,
./probtask                #                   то перед высветкой результат
cat result                #                   будет заново пересчитан:
 #  Число узлов дискретизации (n)=   9
 # Левая граница промежутка  (a)= 0.0000000000000000E+00
 # Правая граница промежутка (b)= 0.1000000000000000E+01
 # N               x                     fun(x)/2                    z/4
   1   0.0000000000000000E+00  -0.1500000000000000E+01  -0.7500000000000000E+00
   2   0.1250000000000000E+00  -0.1375000000000000E+01  -0.6875000000000000E+00
   3   0.2500000000000000E+00  -0.1250000000000000E+01  -0.6250000000000000E+00
   4   0.3750000000000000E+00  -0.1125000000000000E+01  -0.5625000000000000E+00
   5   0.5000000000000000E+00  -0.1000000000000000E+01  -0.5000000000000000E+00
   6   0.6250000000000000E+00  -0.8750000000000000E+00  -0.4375000000000000E+00
   7   0.7500000000000000E+00  -0.7500000000000000E+00  -0.3750000000000000E+00
   8   0.8750000000000000E+00  -0.6250000000000000E+00  -0.3125000000000000E+00
   9   0.1000000000000000E+01  -0.5000000000000000E+00  -0.2500000000000000E+00
$ touch subfun.for       # После модификации исходного файла subfun.for запуск
$ make restab            #  make restab отреагирует адекватно, так как restab
gfortran -c subfun.for   #                              зависит от цели result
gfortran fun.o probtask.o subfun.o -o probtask.out
./probtask
cat result
 #  Число узлов дискретизации (n)=   9
 # Левая граница промежутка  (a)= 0.0000000000000000E+00
 # Правая граница промежутка (b)= 0.1000000000000000E+01
 # N               x                     fun(x)/2                    z/4
   1   0.0000000000000000E+00  -0.1500000000000000E+01  -0.7500000000000000E+00
   2   0.1250000000000000E+00  -0.1375000000000000E+01  -0.6875000000000000E+00
   3   0.2500000000000000E+00  -0.1250000000000000E+01  -0.6250000000000000E+00
   4   0.3750000000000000E+00  -0.1125000000000000E+01  -0.5625000000000000E+00
   5   0.5000000000000000E+00  -0.1000000000000000E+01  -0.5000000000000000E+00
   6   0.6250000000000000E+00  -0.8750000000000000E+00  -0.4375000000000000E+00
   7   0.7500000000000000E+00  -0.7500000000000000E+00  -0.3750000000000000E+00
   8   0.8750000000000000E+00  -0.6250000000000000E+00  -0.3125000000000000E+00
   9   0.1000000000000000E+01  -0.5000000000000000E+00  -0.2500000000000000E+00
$ make result                          # Еще раз убедились, что заново
make: ‘result’ не требует обновления.  # считать результат не нужно.
$ make restab                          # Изменили содержимое файла probtask.par
./probtask                             # и запуск make restab требует пересчета
cat result                             # результата перед показом.
 #  Число узлов дискретизации (n)=   5
 # Левая граница промежутка  (a)= 0.0000000000000000E+00
 # Правая граница промежутка (b)= 0.1000000000000000E+01
 # N               x                     fun(x)/2                    z/4
   1   0.0000000000000000E+00  -0.1500000000000000E+01  -0.7500000000000000E+00
   2   0.2500000000000000E+00  -0.1250000000000000E+01  -0.6250000000000000E+00
   3   0.5000000000000000E+00  -0.1000000000000000E+01  -0.5000000000000000E+00
   4   0.7500000000000000E+00  -0.7500000000000000E+00  -0.3750000000000000E+00
   5   0.1000000000000000E+01  -0.5000000000000000E+00  -0.2500000000000000E+00
```

4. В нашем **make**-файле не указана зависимость файла **probtask.for** от файла **probtask.hdr** — изменение последнего не приведёт при вызове **make** к перекомпиляции **probtask.for**. Как устранить этот недостаток узнаем позже.

4.3.4 GNUPLOT-скрипт выдачи результата в виде графика.

Псевдоцель **resplt** обеспечивает вычерчивание графиков посредством вызова утилиты **gnuplot**, которая выводит на один рисунок графики функций, табулированных в файле **result**. Настройка утилиты на нужный режим черчения указывается в **gnuplot**-скрипте, который удобно записать в отдельном файле. Таким образом, полезно уметь не только писать программы на языках программирования **ФОРТРАН** или **СИ**, но и **make**-файлы, и скрипты для утилиты **gnuplot** или интерпретатора оболочки (см. пункты **1.8.9 – 1.8.10** с примерами скриптов для переформатирования и перекодировки файлов).

Разместим наш **gnuplot**-скрипт в файле **probtask.gnu**:

```
set terminal x11                              # установка порта приписки
set output                                     # вывод на экран

plot 'result' using 2:3 with lines,\
        ''    using 2:4 with linespoints ps 3
pause 5 "the picture 1."                      # задержка рисунка на 5 секунд
set terminal postscript eps enhanced          # смена порта приписки
set output "result.eps"                       # имя файла с рисунком
replot                                        # перевод рисунка через новый порт
reset                                         # восстановление настроек умолчания
```

Здесь

1. **Первая строка.** Установка типа терминала **x11**, на который выводим рисунок. Возможно сокращение **set term x11**.

2. **Вторая строка.** Намечаем вывод рисунка непосредственно на экран.

3. **Третья строка.** Вызов команды **plot** черчения графиков по данным из таблицы, хранящейся в файле **result**. Опция **using 2:3** после имени файла, заключенного в апострофы, указывает, что для первого графика в качестве абсцисс используем данные, хранящиеся во втором столбце таблицы, а в качестве ординат – данные из третьего столбца. Опция **with line** настраивает команду **plot** на черчение графика путем соединения каждой пары двух соседних точек графика отрезком прямой. **Запятая** служит указанием, что далее расположена информация о том, как данной команде **plot** на том же рисунке следует чертить второй график. Если эту информацию хотим поместить в следующей строке, то текущую завершаем символом **обратный слэш**, который должен быть в текущей строке последним (после него не допускается даже комментарий).

4. **Четвертая строка.** На один рисунок можно вывести много графиков. Если все они чертятся по данным из одного файла, то явно имя файла можно указать только для первого графика, используя для остальных более короткий синоним имени – два апострофа, стоящие рядом друг с другом. В качесте абсцисс второго графика используются данные из того же столбца, что и для первого, а в качестве ординат берутся данные из четвертого. Опция **with linepoints**

(сокращенная форма записи **w lp**) информирует команду **plot** о необходимости наряду с прочерчиванием линии еще наносить и точки, соответствующие непосредственно табличным данным.

5. **Пятая строка.** Оператор **pause 5 "the picture 1"** задержит рисунок на экране на пять секунд, а **pause -1** – до нажатия на клавишу **enter**. В двойных кавычках указывается текст, который хотим видеть на экране консоли.

6. **Шестая строка.** Помимо обзора рисунка на экране его полезно сохранить и в файле на диске для возможного включения в печатный отчет. **gnuplot** работает со множеством графических форматов. Здесь выбран **eps**-формат. Поэтому намечаем вывод через терминал **postscript eps enhanced**.

7. **Седьмая строка.** Задаем имя файла, в котором хотим сохранить рисунок.

8. **Восьмая строка.** Команда **replot** перечерчивает рисунок в желаемом формате, направляя в указанный выше файл.

9. **Девятая строка.** Восстанавливаем значения параметров утилиты, соответствующие режиму их умолчания.

4.3.5 Тестирование абстрактной цели вывода рисунка.

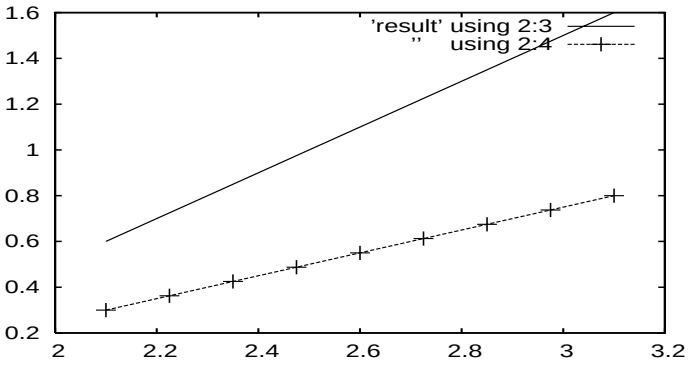


Рис. 1: Графики функций $(\mathbf{2x-3})/\mathbf{2}$ и $(\mathbf{2x-3})/\mathbf{4}$.

```
$ make resplt              # После вызова псевдоцели resplt в течение
gnuplot probtask.gnu       # 5 секунд обозреваем рисунок, а на экране консоли
the picture 1.             # видим сообщение: the picture 1.
```

В правом верхнем углу рисунка видна *легенда*, из которой ясно, какой график соответствует третьему столбцу, а какой – четвертому. Если включим в **gnuplot**-скрипт оператор **set nokey**, то легенда исчезнет. При модификации установки **set key** легенду можно переместить в любую часть рисунка, окружить рамкой и изменить названия кривых. О подробностях работы в среде утилиты **gnuplot** см., например, в [10], а начальное освоение можно провести по методичке **"Азы GNUPLOTa"** (в библиотеке и на сайте Астрономического института СПбГУ).

4.3.6 О чем узнали из четвертого параграфа?

1. Утилита **make** позволяет организовать процесс получения исполнимого файла наиболее оптимальным образом.

2. **make**-файл – это набор инструкций, управляющий работой утилиты **make** и записанный в файл с именем **makefile**, который в простых случаях удобно расположить в директории с исходными файлами, входящими в проект. Работа утилиты активируется через командную строку командой **make**.

3. **make**-файл можно назвать именем отличным от **makefile**. В этом случае отработка инструкций, записанных в нем, инициируется вызовом утилиты с включением опции **-f**, после которой указывается желаемое имя **make**-файла.

4. **Правило** – основная структурная единица **make**-файла; состоит из определения цели, зависимостей и команд, которые позволяют ее достичь.

5. Достижение **цели** может зависеть или не зависеть от наличия других файлов. Список файлов, от которых зависит достижение цели называют списком зависимостей или **пререквизитом.**

6. Имя цели может совпадать с именем файла, а может не совпадать (в последнем случае цель называют **абстрактной**).

7. Примитивная запись **make**-файла в явном виде содержит имена исходных и объектных файлов. Поэтому его длина, грубо говоря, пропорциональна количество исходных файлов. Синтаксис утилиты **make** предоставляет средства записи **make**-файла с автоматической генерацией имен файлов. Так что **make**-файл проекта, состоящего из большого количества исходных файлов, по размеру будет невелик (подробнее см. курс “Операционные системы” или главу **Кое-что об утилите make (часть вторая)**).

8. Главной целью **make**-файла является его первая цель. Обычно такой целью служит компоновка исполнимого файла из объектных.

9. Наряду с главной целью в **make**-файл удобно включать и дополнительные цели: получение из исходных кодов соответствующих объектных, очистка текущей директории от объектных и исполнимого кодов, вывод результата работы программы в табличном и графическом виде.

10. Для вывода результатов работы программы в графическом виде выгодно использовать утилиту **gnuplot**, которая **проста в освоении**, имеет для богатый арсенал графических средств и, кроме того, сама позволяет вести расчет за счет богатого набора встроенных операций и функций.

11. Инструкции, управляющие работой утилиты можно записать в файл (так называемый **gnuplot**-скрипт) с тем, чтобы автоматически инициировать их работу, вызывая утилиту либо из командной строки, либо, например, в качестве команды **make**-файла, которая реализует достижение цели построения графиков.

4.3.7 Седьмое домашнее задание.

Задача N 1:

1. Написать элементарный **make**-файл для отладки и пропуска программы табулирования функции:

$$\mathbf{f(t) = e^t \cdot (1.1 - \sqrt{1.21 - e^{-t}})}.$$

   Главная программа и функция должны быть расположены в разных файлах одной директории.

2. **make**-файл должен обеспечить сборку программного продукта (главная цель), запуск программы с целью получения результата, обзор результата на экране в виде таблицы и соответствующих графиков.

3. **Главная программа должна в режиме одинарной точности:**

   A. Ввести **n**, **a** и **b** из файла **testfun.inp**, используя оператор **format**.
   Здесь **[a,b]** промежуток задания аргумента **t**.
   n – число участков равномерной дискретизации **[a,b]**.

   B. Обеспечить вывод результатов пропуска в файл **result**.

   C. Организовать **n**-кратный вызов функции **fun1(x)**, которая вычисляет одно значение **f(t)**, для соответствующего вызову значения аргумента **t**.

   D. В каждой строке таблицы результата поместить номер аргумента, значение аргумента **t** и значение **f(t)**, вычисленное **fun1(x)** непосредственно по заданной формуле.

4. **При тестировании:**

   (a) Оценить правильность работы **fun1** при **n=8**, **a=1.2**, **b=1.6**.

   (b) Оценить правильность работы **fun1** при **n=8**, **a=12**, **b=16**.

   (c) Аналитически найти $\lim\limits_{t\to\infty} f(t)$.

   (d) Аналитически получить иную формулу для расчета требуемой функции.

   (e) Сформулировать причину, по которой расчётные свойства новой формулы предпочтительнее.

   (f) Оформить расчет по новой формуле подпрограммой **subfun(x,res)**.

   (g) Подправить соответствующим образом **make**-файл.

   (h) Модифицировать главную подпрограмму так, чтобы она выводила в одну таблицу результаты работы и **fun1**, и **subfun**.

   (i) Убедиться в правильности результатов, получаемых **subfun**.

   (j) Модифицировать **gnuplot**-скрипт так, чтобы он позволял выводить на один рисунок оба графика.

   (k) Исследовать работу программы на типах **REAL(mp)** при **mp=4, 8, 10, 16**.

5 Контрольная работа N 1 (часть 1)

Работа нацелена на практическое освоение и закрепление тем:

1. Процедурно-ориентированное программирование на ФОРТРАНе и СИ.
2. Базовые алгоритмические структуры обоих языков.
3. Простейшие приемы ввода и вывода данных из файла.
4. Оформление алгоритмов подпрограммами и функциями.
5. Элементарное использование **make**-файла и утилиты **gnuplot**.
6. Элементарное использование утилиты **bc** и системы **maxima**.
7. Приведение простых формул к виду удобному для расчета.

Предлагается задача табулирования в режиме одинарной точности некоторой простой функции. Условие содержит синтаксически верный (**но неудовлетворительный по качеству расчета**) вариант её решения.

5.1 План выполнения контрольной

1. Устранить основные недостатки текста главной ФОРТРАН-программы.
2. Используя вычислительные возможности утилиты **bc** и системы **maxima**, убедиться в неверности работы ФОРТРАН-функции, данной в условии.
3. Найти предел функции, соответствующей задаче (вручную и используя аналитические возможности системы **maxima**). Пояснить причины его отличия от результата табулирования.
4. Преобразовать исходную формулу к виду удобному для расчета и оформить соответствующий алгоритм ФОРТРАН-функцией.
5. Главная программа должна вызывать эту функцию и выводить результат в файл в виде удобном для применения утилиты **gnuplot**.
6. Написать **make**-файл для создания и уничтожения исполнимого и объектных файлов, вызова программы и вывода её результатов на экран.
7. Написать **gnuplot**-скрипт создания **eps**-файла с графиками обоих вариантов расчета (а также абсолютной и относительной погрешности исходного способа по отношению к новому), и включить псевдоцель его вызова в **make**-файл.
8. ФОРТРАН-решение задачи в режиме удвоенной точности (**см. пункт 5.5**).
9. СИ-решение задачи в режиме удвоенной точности. Для ввода и вывода использовать функции **fscanf** и **fprintf** (**см. пункт 5.6**).

Замечания:

1. Запись ФОРТРАН-функций, прилагаемых к условию задач, часто содержит рабочие переменные, использующиеся для хранения промежуточных результатов, которые можно было бы и не запоминать, добиваясь более высокой точности, поскольку в этом случае промежуточные значения хранятся на многоразрядных регистрах процессора. На самом деле в реальных объемных расчетах, когда алгоритм записывается множеством формул, без вспомогательных переменных не обойтись. Их наличие в предложенных текстах просто моделирует реальную ситуацию, позволяя проявить эффекты, сглаживаемые многоразрядностью.

2. Некоторые задачи на первый взгляд кажутся искусственными. Например, рассматриваемая ниже задача, об отношении разности между функцией **sin(x)** и несколькими членами ее разложения в ряд Маклорена к аналогичной разности для функции **cos(x)**. Тем не менее, в предложенном виде задача позволяет наглядно уяснить и причину практически полной потери точности при ведении расчета по исходной формуле, и суть ее преобразования, которое обеспечивает получение верного результата.

3. Можно привести немало примеров, когда простые на вид формулы оказываются непригодными для расчета по ним на ЭВМ так, что приходится их аналитически преобразовывать к более подходящему виду.

4. Элементарные функции **sin(x)** и **cos(x)** хорошо знакомы со школы. Поэтому столкновение с "*неожиданной*" потерей точности "*на ровном месте*" заставит неформально прочувствовать ситуацию на простом материале и осознанно подойти к программированию формул, которые встретятся при выполнениии курсовых и дипломных работ.

5. Все задачи контрольной демонстрируют один эффект – потерю точности при вычитании на ЭВМ почти равных чисел. Для получения формулы, выгодной для расчета, как правило, достаточно знаний, полученных в средней школе. Умение аналитически преобразовывать формулы важно не только для их приведения к более элегантному академическому виду, но и к виду удобному для расчёта, т.е. важно для практики вычислений на ЭВМ.

6. Для получения зачета по первой части контрольной достаточно сдать письменный отчёт и грамотно продемонстрировать работу программ, написанных на СИ и ФОРТРАНе-95 (в стиле ФОРТРАНа-77), а также соответствующих **bc**- и **maxima**-скриптов.

7. Конечно, ФОРТРАН-77 и СИ – достаточно древние языки. Однако, при при выполнении курсовых и дипломных работ часто приходится иметь дело с комплексами программ, написанными программистами старшего поколения именно на этих языках. Поэтому полезно познакомится на практике с базовыми элементами и приемами программирования на ФОРТРАНе-77 и СИ. В дальнейшем, при выполнении второй части контрольной освоим и закрепим некоторые альтернативные базовые возможности языков ФОРТРАН-95 и C++.

5.2 Пример условия контрольной задачи.

Необходимо разработать алгоритм расчета функции

$$\mathbf{w(x)} = \frac{\cos \mathbf{x} - \mathbf{1} + \dfrac{\mathbf{x^2}}{\mathbf{2}} - \dfrac{\mathbf{x^4}}{\mathbf{24}} + \dfrac{\mathbf{x^6}}{\mathbf{720}}}{\dfrac{\sin \mathbf{x}}{\mathbf{x}} - \mathbf{1} + \dfrac{\mathbf{x^2}}{\mathbf{6}} - \dfrac{\mathbf{x^4}}{\mathbf{120}} + \dfrac{\mathbf{x^6}}{\mathbf{5040}}}$$

для $\mathbf{x} = \mathbf{e^{-t}}$ при $\mathbf{t} \in [\mathbf{1.0}, \mathbf{2.0}]$. Была составлена подпрограмма-функция $\mathbf{w0(x)}$:

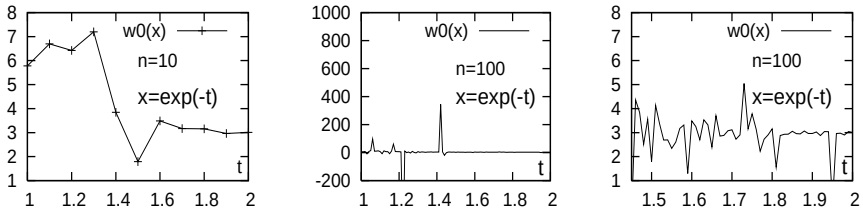
```
      function w0(x)                                    !    Файл w0.for
      x2=x*x
      a=cos(x) - 1 + x2* 0.5*(1 - x2/12 * ( 1 - x2 / 30))
      b=sin(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42))
      w0=a/b
      end
```

которая вела расчет непосредственно по заданной формуле. Тестирование функции проводилось программой

```
      program tsfs2p30                        !    Файл  tsfs2p30.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result', status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' # t0=',t0,'  tn=',tn,'  n=',n
      pi4=atan(1.0)
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=(t0+i*ht)
        x=exp(-t)
        r0=w0(x)
        write(nres,1001) i, t, r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

При значениях аргумента $\mathbf{t}$ из диапазона $\mathbf{t} = [\mathbf{-1.0}, \mathbf{-2.0}]$ (то есть при $\mathbf{x} = \mathbf{e}$ и $\mathbf{x} = \mathbf{e^2}$) дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Табулирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{1.0}, \mathbf{2.0}]$ привело к следующему:

```
#   t0=   1.000000         tn=   2.000000         n=          10
#  i            t              w0
   0      0.1000000E+01  0.5782932E+01   Ниже приведены графики w0(x)
   1      0.1100000E+01  0.6694451E+01   при x=exp(-t)   и   1<=t<=2.
   2      0.1200000E+01  0.6426376E+01
   3      0.1300000E+01  0.7200233E+01  Левый график построен посредством
   4      0.1400000E+01  0.3847253E+01  соединения отрезками прямых 11 точек
   5      0.1500000E+01  0.1791706E+01  с абсциссами и ординатами из второй
   6      0.1600000E+01  0.3489712E+01  третьей колонок настоящей таблицы.
   7      0.1700000E+01  0.3166201E+01
   8      0.1800000E+01  0.3158754E+01  Остальные два графика построены
   9      0.1900000E+01  0.2960747E+01  аналогично, но по 100 точкам.
  10      0.2000000E+01  0.3008335E+01
```



В данном случае проводить аппроксимирующую кривую бессмысленно – в указанном диапазоне неверна уже старшая цифра, полученная **w0(x)**.

План выполнения первой части контрольной

1. Устранить основные недостатки текста главной ФОРТРАН-программы.
2. Используя вычислительные возможности утилиты **bc** и системы **maxima**, убедиться в неверности работы ФОРТРАН-функции, данной в условии.
3. Найти предел функции, соответствующей задаче (вручную и используя аналитические возможности системы **maxima**). Пояснить причины его отличия от результата табулирования.
4. Преобразовать исходную формулу к виду удобному для расчета и оформить соответствующий алгоритм ФОРТРАН-функцией.
5. Главная программы должна вызывать эту функцию и выводить результат в файл в виде удобном для применения утилиты **gnuplot**.
6. Написать **make**-файл для создания и уничтожения исполнимого и объектных файлов, вызова программы и вывода её результатов на экран.
7. Написать **gnuplot**-скрипт создания **eps**-файла с графиками обоих вариантов расчета (а также абсолютной и относительной погрешности исходного способа по отношению к новому), и включить псевдоцель его вызова в **make**-файл.
8. ФОРТРАН-решение задачи в режиме удвоенной точности (**см. пункт 5.5**).
9. СИ-решение задачи в режиме удвоенной точности. Для ввода и вывода использовать функции **fscanf** и **fprintf** (**см. пункт 5.6**).

5.3 Исходное ФОРТРАН-решение (уяснение ситуации)

1. Первый недостаток полученного исходного текста.
2. Второй и главный недостаток предложенного решения.
3. Объективная и субъективная причины неверности результата.
4. Новая подпрограмма-функция расчета искомого отношения.
5. Черновой вариант модернизированной главной программы.
6. Make-файл модернизированной программы.
7. Результаты пропуска модернизированной программы.
8. Что должны знать и уметь после сдачи первой части?

5.3.1 Первый недостаток полученного исходного текста

— использование правила умолчания ФОРТРАНа. , по которому любое имя, начинающееся с букв $\mathbf{i}, \mathbf{j}, \mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n}$ трактуется компилятором, как имя данного целого типа; а имя, начинающееся с любой другой буквы, как имя данного вещественного типа.

1. Использование правила умолчания можно оправдать (с некоторой натяжкой) в очень коротких программах, когда опечатки в именах просто обнаружить.

2. Если при наборе программы вместо оператора $\mathbf{t = t0 + (i - 1) * h}$ напишем $\mathbf{t = tO+(i-1)*h}$, не заметив, что "нуль" в имени переменной **t0** заменила буква **O**, то компиляция пройдёт удачно и даже получим некий числовой результат:

```
#   t0=  1.    tn=  2.    n= 10
#  i            t              w0
   0      0.2970153E-38  0.8981886E+01
   1      0.1000000E+00  0.8986946E+01
   2      0.2000000E+00  0.8958036E+01
   3      0.3000000E+00  0.8969361E+01
   4      0.4000000E+00  0.8898288E+01
   5      0.5000000E+00  0.8972149E+01
   6      0.6000000E+00  0.8877596E+01
   7      0.7000000E+00  0.9807290E+01
   8      0.8000000E+00  0.8068257E+01
   9      0.9000000E+00  0.7077634E+01
  10      0.1000000E+01  0.5782932E+01
```

3. Он отличается от предыдущего. Если не заметить, что значение аргумента на единицу меньше требуемого, и проверить, что $\mathbf{w0(1) = 8.981886}$, то сомнений в правильности работы $\mathbf{w0(x)}$ может и не возникнуть.

4. Отмена действия правила умолчания посредством оператора **implicit none** выявит опечатку еще при компиляции. Поэтому используем его всегда.

5. Обычно компиляторы имеют опцию, отключающую действие правила умолчания. К сожалению, имена этих опций у разных компиляторов могут различаться (**-Wimplicit** у **g77** и **-fimplicit-none** у **gfortran**). Так что проще в каждой единице компиляции явно отключать действие традиционного правила умолчания, чем запоминать для разных компиляторов соответствующие опции.

5.3.2 Второй и главный недостаток предложенного решения.

В исходной формуле в числителе находится разность между функцией $\cos \mathbf{x}$ и ее приближения первыми четырьмя членами разложения в ряд Маклорена. При значениях аргумента $\mathbf{x} \approx \mathbf{0.1}$ значения функции и ее приближения совпадут с точностью до восьми значащих цифр мантиссы так, что операция вычитания их на одинарной точности приведет к потере всех верных цифр результата.

Подобная ситуация наблюдается и в знаменателе: из функции $\frac{\mathbf{sinx}}{\mathbf{x}}$ вычитаются четыре старших члена ее разложения с аналогичной потерей верных цифр. Так что значения, полученные функцией $\mathbf{w0(x)}$ при $\mathbf{x} \approx \mathbf{0.1}$ – абсолютно неверны, хотя и кажутся правдоподобными. Убедиться в этом можно и аналитически, и практически. Аналитический подход требует нахождения предела вычисляемой функции при стремлении аргумента к нулю; практический – либо аналитического приведения исходной формулы к виду более удобному для ведения расчета на ЭВМ, либо перехода на более многозначную арифметику.

Последнее кажется менее хлопотным ввиду наличия утилиты **bc**, предоставляющей возможность ведения расчёта практически с произвольным количеством значащих цифр. Приведём элементарные примеры работы с **bc**:

```
$ bc -l        # Опция -l предоставляет bc возможность использовать
bc 1.06        # при расчётах функции математической библиотеки.
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.

scale=50                       /* установили число цифр после запятой */
1.0/3.0
.33333333333333333333333333333333333333333333333333
t=sqrt(2.0)                                          /*   запомнили   */
t                                                    /*  в t sqrt(2)  */
1.41421356237309504880168872420969807856967187537694
t*t                                                  /* проверка t*t  */
1.99999999999999999999999999999999999999999999999997
pi=4*a(1)                                              /* a(1)=arctg(1) */
3.14159265358979323846264338327950288419716939937508
pi6=4*a(1)/6                                         /*   pi6=pi/6    */
s(pi6)                                               /*  s(x)=sin(x)  */
.49999999999999999999999999999999999999999999999999
c(pi6)                                               /*  c(x)=cos(x)  */
.86602540378443864676372317075293618347140262690519
s(pi6)^2+c(pi6)^2                                    /* s(x)^2+c(x)^2 */
.99999999999999999999999999999999999999999999999998
e(1)                                                 /*  e(x)=exp(x)  */
2.71828182845904523536028747135266249775724709369995
l(e(1/3))                                            /*   l(x)=ln(x)  */
.33333333333333333333333333333333333333333333333332
quit                                          /* команда выхода из bc */
```

Применим утилиту **bc** для проверки расчётов, полученных ФОРТРАН-программой. Потребуем, чтобы **bc** вела расчёт с пятьюдесятью цифрами после запятой.

Составим **bc**-скрипт для нашей задачи, разместив его, например, в файле **fun.bc**:

```
scale=50
define w(x){        /* Определяем заголовок функции в bc        */
auto x2, a, b       /* Указываем локальные переменные функции w */
      x2=x*x
      a=c(x) - 1 + x2* 0.5*(1 - x2/12 * ( 1 - x2 / 30))
      b=s(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42))
      return(a/b)                          /* возврат результата */
}

t0=1                          /* левая граница промежутка [1,2] */
t1=2                          /* правая                         */
n=10                          /* число шагов дробления          */
h=(t1-t0)/n                   /* расчёт шага                    */
for (i=1;i<10;i++){           /* заголовок оператора цикла      */
t=t0+(i-1)*h                  /* расчёт аргумента t             */
 x=e(-t)                      /* расчёт аргумента x             */
w(x)                          /* вызов функции w(x)             */
}
```

В результате его запуска посредством

```
$ bc -l < fun.bc > fun.res
```

получим в файле **fun.res** следующий результат:

```
8.99754060051501279986883477968249877021342617279776
8.99798622967396189763732200591777643476069814268573
8.99835114074282109926149300438382196183304259657130
8.99864994539330454240802754502342940273654722596357
8.99889461325593972145970291288119859580964876063391
8.99909494866309035525683886363259042211509257103025
8.99925898169114057869846127420395939833901528597643
8.99939328880003771011522595411715451460296270768362
8.99950325567335002314478564750315315080877805424147
```

Сравнение его с таблицей из пункта **5.2** наглядно демонстрирует неверность полученных в ней данных. Если сравним время работы исполнимого файла ФОРТРАН-программы со временем работы утилиты **bc**, повторив запуск обоих под присмотром утилиты **time** (**time ./a.out** и **time bc -l** $<$ **fun.bc** $>$ **fun.res**), то обнаружим, что ФОРТРАН-программа работает раз в 15 быстрее. Правда, если положить scale=8, то **bc** будет работать только вдвое медленнее, но только результаты, как и в случае ФОРТРАНа, окажутся абсолютно неверными.

5.3.3 Пример использования системы maxima

Высокоточный расчёт можно также провести и в среде системы аналитических преобразований **maxima**, доступной под эгидой **GNU**-лицензии. Основное назначение **maxima** состоит именно в проведении аналитических преобразований. Поэтому посредством **maxima** (помимо высокоточного расчёта) можно найти и искомый предел $\lim\limits_{\mathbf{x\to 0}} \mathbf{w(x)}$, и получить разложения в ряды Маклорена (числителя и знаменателя нашей $\mathbf{w(x)}$ и её самой), а также аналитически решать многие другие задачи, не говоря уже о возможности **maxima**-подключения утилиты **gnuplot** для графического отображения результатов (альтернативными аналогами **maxima** служат коммерческие продукты **windows**-приложение **Математика** и пакет **Maple**).

Применим **maxima** для решения проблем, возникших в нашей задаче: нахождения предела $\lim\limits_{\mathbf{x\to 0}} \mathbf{w(x)}$; получения разложений в ряд Маклорена числителя и знаменателя $\mathbf{w(x)}$, а также её самой; и, наконец, расчёта значений $\mathbf{w(x)}$.

Составим **maxima**-скрипт:

```
                                        /*             Файл test30.mac */
writefile("test30.res")$                /* Установка файла вывода      */
num(x):=cos(x)-1+x^2/2-x^4/24+x^6/720$  /* Функция из числителя  w(x)  */
den(x):=sin(x)/x-1+x^2/6-x^4/120+x^6/5040$ /* Функция из знаменателя w(x) */
w(x):=num(x)/den(x)$                    /* Определяем функцию w(x)     */
lhospitallim:10$          /* Правило Лопиталя разрешаем применять 10 раз */
limit(w(x),x,0);          /* Находим lim w(x) при x-->0                  */
taylor(cos(x),x,0,8);     /* Получаем разложение cos(x) в ряд Маклорена  */
taylor(sin(x)/x,x,0,9);   /*      --"--"--       sin(x)     --"--"--     */
taylor(num(x),x,0,10);    /*      --"--"-- числителя  w(x)  --"--"--     */
taylor(den(x),x,0,10);    /*      --"--"-- знаменателя w(x) --"--"--     */
taylor(  w(x),x,0,8);     /*      --"--"--         w(x)     --"--"--     */
fpprec:50$                /* Задали число цифр мантиссы.                 */
t0:1.0b0$                 /* Конечные точки отрезка [t0,t1].             */
t1:2.0b0$
 n:10$                    /* Число шагов равномерного дробления [t0,t1]. */
h:bfloat((t1-t0)/n);      /* Шаг дробления отрезка [t0,t1].              */
for i:0 step 1 thru n do  /* Цикл по i от 0 до n с шагом 1               */
  (                       /*   открытие составного оператора тела цикла; */
    t:bfloat(t0+i*h),     /*     очередной узел дробления [t0,t1];       */
    x:bfloat(exp(-t)),    /*     аргумент функции w(x);                  */
    y:bfloat(w(x)),       /*     значение функции w(x);                  */
    print(y)              /*     вывод результата                        */
  );                      /*   закрытие составного оператора тела цикла  */
x:1b-20$
print(" x=",x,"fpprec=50")$
bfloat(w(x));             /* расчёт w(1b-20) при fpprec=50               */
fpprec:215;
print(" x=",x,"fpprec=215")$
bfloat(w(x));             /* расчёт w(1b-20) при fpprec=215              */
closefile();              /*   закрытие файла вывода                     */
```

Пояснения:

1. По умолчанию **maxima** настроена на режим диалога с пользователем. Нам удобнее возможность ввода всех нужных инструкций из файла и вывода результатов работы не только на экран, но и в файл на диске (для фиксации результатов), что обеспечивает **maxima**-функция **writefile**, аргументом которой служит имя файла, принимающего результат.

2. Наличие значка **$**, завершающего какую-то команду, означает, что результат её работы не будет выведен на экран (т.е. результат нам нужен, но неинтересен). Для вывода результата завершаем **maxima**-оператор значком **;**.

3. Далее нами определены функции **num(x)**, **den(x)** и **w(x)**. На самом деле нам нужна только **w(x)**. Остальные использованы просто для демонстрации.

4. При поиске предела выражения, сводящегося к неопределённости вида $\frac{\mathbf{0}}{\mathbf{0}}$, часто помогает последовательное применение Лопиталя. По умолчанию опционная **maxima**-переменная **lhospitallim** задаёт кратность его применения равную **4**, что маловато для нашего случая. Переопределим её значение на **10**.

5. Оператор присваивания в **maxima** обозначается двоеточием **:**.

6. Вызов **limit(w(x),x,0)** ищет искомый предел при $\mathbf{x} \to \mathbf{0}$.

7. Вызов **taylor(cos(x), x, 0, 8)** найдёт усечённое (вплоть до $\mathbf{x^8}$) разложение **cos(x)** в ряд Тейлора в окрестности точки **x=0**.

8. Опционная **maxima**-переменная **fpprec** задаёт число значащих цифр мантиссы числа с плавающей запятой (по умолчанию **fpprec=16**). Изменяем её значение на **50** (для сравнения с результатом работы утилиты **bc**).

9. **1.0b0**. Литера **b** (от bigfloat) указывает, что числовая константа имеет **fpprec**-значную мантиссу.

10. Функция **bfloat(выражение)** преобразует все числа и функции от них, входящие в выражение, в значения с **fpprec**-значной мантиссой.

11. Функция **print** выводит в строку значения своих аргументов.

12. В пяти предпоследних строках демонстрируется, что при значении аргумента $\mathbf{x} = \mathbf{10^{-20}}$ пятидесятизначной мантиссы недостаточно для правильного расчёта значения **w(x)**, в то время как пятисотзначная гарантирует правильность цифр из сорока восьми старших десятичных разрядов.

13. Активация **maxima**-скрипта из командной строки:

```
$ maxima -b test30.mac
```

5.3.4 Результат работы скрипта test30.mac

```
;; Dribble of #<IO TERMINAL-STREAM> started on 2011-10-30 15:58:34.
#<OUTPUT BUFFERED FILE-STREAM CHARACTER test30.res>
                              6       4    2
                             x     - x    x
(%i3)               num(x) := --- + ---- + -- - 1 + cos(x)
                             720    24    2
                               6       4    2
                              x     - x    x        sin(x)
(%i4)               den(x) := ---- + ---- + -- - 1 + ------
                             5040   120    6          x
                                       num(x)
(%i5)                           w(x) := ------
                                       den(x)
(%i6)                          lhospitallim : 10
(%i7)                          limit(w(x), x, 0)
(%o7)                                  9
(%i8)                        taylor(cos(x), x, 0, 8)
                             2    4    6      8
                            x    x    x      x
(%o8)/T/                 1 - -- + -- - --- + ----- + . . .
                            2    24   720   40320
                                    sin(x)
(%i9)                        taylor(------, x, 0, 9)
                                       x
                           2    4      6       8
                          x    x      x       x
(%o9)/T/                1 - -- + --- - ---- + ------ + . . .
                          6    120   5040   362880
(%i10)                     taylor(num(x), x, 0, 10)
                              8        10
                             x        x
(%o10)/T/                    ----- - ------- + . . .
                             40320   3628800
(%i11)                     taylor(den(x), x, 0, 10)
                              8        10
                             x        x
(%o11)/T/                    ------ - -------- + . . .
                             362880   39916800
(%i12)                       taylor(w(x), x, 0, 8)
                    2       4        6                8
                   x     8 x        x          38971 x
(%o12)/T/     9 - -- + ------ + -------- - -------------- + . . .
                   55   117975   90840750   53000127180000
```

1. Метки **%i3**, **%i4** и т.д. именуют номера **вводимых** (**i** – от input) команд. **%o7**, **%o8** и т.д. – имена выводимых результатов.

2. Метки **%i1** и **%i2** не попали в файл вывода, так как они именовали команды **batch("test30.mac")** и **writefile("test30.res")$** соответственно, которые отработали до появления файла **test30.res**.

3. Меток с **%o3** по **%o6** нет, так как мы завершали команды, соответствующие меткам с **%i3** по **%i6** значком **$**.

```
(%i13)                              fpprec : 50
(%i14)                              t0 : 1.0b0
(%i15)                              t1 : 2.0b0
(%i16)                                n : 10
                                           t1 - t0
(%i17)                           h : bfloat(-------)
                                             n
(%o17)                                1.0b-1
(%i18) for i from 0 thru n do (t : bfloat(i h + t0), x : bfloat(exp(- t)),
                                                  y : bfloat(w(x)), print(y))
8.997540600515012799868834779682498770213095830764 9b0
8.997986229673961897637322005917776434759624556408 8b0
8.998351140742821099261493004383821961831454875462b0
8.998649945393304542408027545023429402733141374670 9b0
8.998894613255939721459702912881198595801855301250 5b0
8.999094948663090355256838863632590422099930361136 7b0
8.999258981691140578698461274203959398312099190598 3b0
8.999393288800037710115225954117154514491480787002 4b0
8.999503255673350023144785647503153150577669413179 8b0
8.999593292629747914277159673742822891369838678885 4b0
8.999667011131957263300581905023488409798848777896 6b0
(%o18)                                 done
(%i19)                              x : 1.0b-20
(%i20)                       print( x=, x, fpprec=50)
 x= 1.0b-20 fpprec=50
(%i21)                             bfloat(w(x))
(%o21)     - 5.90417808188771421256213461023370651455554307082 57b-1
(%i22)                             fpprec : 215
(%o22)                                  215
(%i23)                       print( x=, x, fpprec=215)
 x= 1.0b-20 fpprec=215
(%i24)                             bfloat(w(x))
(%o24) 8.99999999999999999999999999999999999999999981818181...47b0\
(%i25)                              closefile()
;; Dribble of #<IO TERMINAL-STREAM> finished on 2011-10-30 15:58:39.
```

Заметим, что при $\mathbf{x \approx 0.1}$ и 50-значной мантиссе результаты **maxima**- и **bc**-расчётов совпадают лишь в пределах старших 37 разрядов. Разности в числителе и знаменателе $\mathbf{w(x)}$ по порядку величины равны $\approx \mathbf{10^{-13}}$, хотя сами уменьшаемые и вычитаемые порядка единицы. Это означает, что в их разностях на 13 десятичных цифр меньше (50 - 13 = 37).

5.3.5 Устранение субъективной причины неверности результата

Объективная причина получения неверного результата на типе **real(4)** заключается в недостаточной для используемой расчетной формулы разрядности ЭВМ. Субъективная причина состоит в том, что, зная про наличие объективной причины, мы, не приводя исходную формулу к виду, более подходящему для расчета на одинарной точности, пытаемся посредством операции вычитания найти разность, все значащие цифры которой получаются из разрядов уменьшаемого и вычитаемого, находящихся вне действующей разрядной сетки ЭВМ.

Для одиночного контрольного высокоточного расчёта **maxima** удобна и полезна. К сожалению, *одна дождинка еще не дождь* – и, если нет аппаратной поддержки ячеек сверхвысокой разрядности, то объективная причина не устраняется.

Субъективную причину можно устранить, если найти математическое преобразование, которое позволит аналитически исключить из исходной формулы вычитание почти равных слагаемых, оставив в качестве результата старшие члены разложения самой разности.

В нашем случае такое исключение проводится легко: достаточно просто зачеркнуть в разложениях $\cos \mathbf{x}$ и $\dfrac{\sin \mathbf{x}}{\mathbf{x}}$ вычитаемые в исходной формуле слагаемые:

$$\cos \mathbf{x} - \mathbf{1} + \frac{\mathbf{x^2}}{\mathbf{2}} - \frac{\mathbf{x^4}}{\mathbf{24}} + \frac{\mathbf{x^6}}{\mathbf{720}} = \sum_{\mathbf{k=4}}^{\infty} (\mathbf{-1})^{\mathbf{k}} \frac{\mathbf{x^{2k}}}{\mathbf{(2k)!}} = \mathbf{x^8} \sum_{\mathbf{m=0}}^{\infty} (\mathbf{-1^m}) \frac{\mathbf{x^{2m}}}{\mathbf{(2m+8)!}}$$

$$\frac{\sin \mathbf{x}}{\mathbf{x}} - \mathbf{1} + \frac{\mathbf{x^2}}{\mathbf{6}} - \frac{\mathbf{x^4}}{\mathbf{120}} = \sum_{\mathbf{k=4}}^{\infty} (\mathbf{-1})^{\mathbf{k}} \frac{\mathbf{x^{2k}}}{\mathbf{(2k+1)!}} = \mathbf{x^8} \sum_{\mathbf{k=0}}^{\infty} (\mathbf{-1^k}) \frac{\mathbf{x^{2k}}}{\mathbf{(2k+9)!}}$$

Так что искомая функция выражается отношением

$$\mathbf{w(x)} = \frac{\displaystyle\sum_{\mathbf{k=0}}^{\infty} (\mathbf{-1^k}) \frac{\mathbf{x^{2k}}}{\mathbf{(2k+8)!}}}{\displaystyle\sum_{\mathbf{k=0}}^{\infty} (\mathbf{-1^k}) \frac{\mathbf{x^{2k}}}{\mathbf{(2k+9)!}}}$$

Выводы:

1. **Наличие в формуле обращения к какой-нибудь функции, для которой в программном обеспечении существует соответствующая встроенная функция (например, sin(x)), ещё не означает, что можно не знать алгоритмов расчёта этой функции.**

2. **Формула может обладать опасными для расчёта на ЭВМ свойствами, так что использование встроенных функций не всегда выгодно.**

3. **Именно поэтому полезно знать, понимать и уметь применять различные способы их расчета.**

5.4 ФОРТРАН-77: решение задачи в режиме real(4)

Накопление каждой из сумм (числителя и знаменателя) можно организовать в пределах одного оператора цикла. Условие прекращения суммирования – как только очередное слагаемое не в состоянии изменить накопленную сумму.

Для расчета значения очередного слагаемого выгодна рекурентная формула, выражающая его через номер и значение предыдущего, которая исключает необходимость явного расчета факториалов и вызова функции возведения в степень:

$$\mathbf{a_{k+1}} = -\mathbf{a_k} \cdot \frac{\mathbf{x^2}}{\mathbf{(2k+9)(2k+10)}}$$

$$\mathbf{b_{k+1}} = -\mathbf{b_k} \cdot \frac{\mathbf{x^2}}{\mathbf{(2k+10)(2k+11)}}$$

Легко сообразить (см. для проверки в пункте **5.3.4** соответствующие разложения, полученные **maxima**-командой **taylor**), что начальное значение слагаемого для числителя равно $a = \dfrac{1}{40320.0}$, а для знаменателя — $b = \dfrac{1}{362880.0}$. Так что при $\mathbf{x} \leq \mathbf{1}$ для расчёта $\mathbf{w(x)}$ выгодно использовать функцию $\mathbf{w1(x)}$:

```
function w1(x)  !                              Файл w1.for
implicit none   !                 Исправлен первый недостаток
real w1, x, x2
real sa, sa1, sb
real a, b
integer k
x2=x*x       ! Запомнили x**2, чтобы не перевычислять его в цикле.
sa1=1.0      ! Задаем значения переменных, входящих в условие продолжения
sa=0         ! цикла do while так, чтобы его тело гарантированно начало
sb=0         ! работу: sa1 - предыдущее значение числителя; sa - текущее.
a=1/40320.0  ! Инициализируем текущее слагаемое  числителя.
b=a/9        ! --"--"--"--"--"--"--"--"--"--"-- знаменателя.
k=0          ! --"--"--"--"--  индекс (номер) текущего слагаемого.
do while (abs(sa).ne.sa1)   ! Пока не совпали предыдущая и текущая суммы:
  sa1=abs(sa)               !   переопределяем предыдущую на текущую;
  sa=sa+a                   !   уточняем текущую сумму числителя,
  a=-a*x2/(2*k+9)/(2*k+10)  !   вычисляем его очередное слагаемое;
  sb=sb+b                   !   уточняем текущую сумму знаменателя,
  b=-b*x2/(2*k+10)/(2*k+11) !   вычисляем его очередное слагаемое;
  k=k+1                     !   вычисляем номер очередного слагаемого
enddo
w1=sa/sb                    !   присваиваем результат ИМЕНИ ФУНКЦИИ.
end
```

Заметим, что **нет необходимости** запоминать значения текущих слагаемых числителя и знаменателя в элементах массива, несмотря на наличие в формулах индексированных переменных.

5.4.1 Новая тестирующая главная программа.

```
      program tsfs2p30a                          !    Файл  tsfs2p30a.for
      implicit none                              ! Исправлен первый недостаток.
      integer ninp, nres, n, i
      real w0, w1, t0, tn, ht, t, x, r0, r1, aer, rer
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=(t0+i*ht); x=exp(-t); r0=w0(x); r1=w1(x)
        aer=abs(r1-r0)                           ! Абсолютная погрешность
        rer=aer/r1                               ! Относительная погрешность
        write(nres,1001) i, t, r0, r1, aer, rer
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',12x,'w0',12x,'w1',11x,
     >                                    'aer',11x,'rer')
 1001 format(1x,i5,2x,2x,e14.6,e14.6,e14.6,e14.6,e14.6)
      end
```

Замечания:

1. Наряду с вызовом **w1(x)** вычисляем абсолютную и относительную погрешности **w0(x)**, полагая значения **w1(x)** точными (в пределах 6-7 цифр мантиссы).

2. В операторе **open** открытия файла **result** (принимающего результат), заданы не только ключевые параметры **unit=nres** и **file='result'**, но и параметр **status='replace'**. Последний означает, что при очередном запуске программы старый результат, находившийся в файле **result** будет полностью новым.

3. Абсолютную и относительную погрешности можно вычислить и начертить внутри **gnuplot**-скрипта, не поручая их расчёт главной программе, например, так:

```
plot [1:2] [0:10] 'result' u 2:3   title ' w0(x) '  w lp lw 3 ps 3,\
     ''       u 2:4   title ' w1(x) '  w l  lw 3,\
     ''       u 2:(abs(($3)-($4)))   title ' abserr w0 ' w l,\
     ''       u 2:(abs(($3)-($4))/($4))   title ' relerr w0 ' w l
```

4. Напоминаем, что значок доллара перед номером колонки на языке утилиты **gnuplot** означает выборку чисел, стоящих в данной колонке. Если его не написать, то утилита *решит*, что абсолютную величину надо получить для разности **3-4**.

5.4.2 Make-файл для модернизированной программы.

```
comp:=gfortran
       main : tsfs2p30a.o w0.o w1.o
[  TAB ]     $(comp) tsfs2p30a.o w0.o w1.o -o main
tsfs2p30a.o : tsfs2p30a.for
[  TAB ]     $(comp) -c tsfs2p30a.for
       w0.o : w0.for
[  TAB ]     $(comp) -c w0.for
       w1.o : w1.for
[  TAB ]     $(comp) -c w1.for
     clear  :
[  TAB ]     rm *.o
     result : input main
[  TAB ]     ./main
     restab : result main
[  TAB ]     cat result
     resplt : result main
[  TAB ]     gnuplot 'view.gnu'
```

1. **comp** – придуманная нами переменная для упрощения модификации **make**-файла на тот случай, если пожелаем заменить компилятор.

2. Запись **$(comp)** означает, что утилита **make** должна взять имя команды из переменной **comp**.

3. Добавление новой зависимости **w1.o** в рабочий проект привело к необходимости добавления новой цели **w1.o** и команды ее достижения. Такой подход полезен при начальном освоении технологии работы с утилитой **make**, обеспечивая в принципе лучшее понимание сути дела. Однако, утилита **make** имеет средство для автоматической генерации имен целей. Поэтому размер **make**-файла не обязан линейно зависеть от количества описываемых и выполняемых целей, оставаясь не больше приведенного выше (подробнее см., например, курс "Операционные системы").

4. Помним, что отключить действие правила умолчания нужно в каждой единице компиляции; в частности, и в функции **w0(x)**. Её исходный текст после исправления:

```
function w0(x)                          !           Файл w0.for
implicit none                           ! Исправлен первый недостаток
real w0, x, x2, a, b
x2=x*x
a=cos(x) - 1 + x2* 0.5*(1 - x2/12 * ( 1 - x2 / 30))
b=sin(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42))
w0=a/b
end
```

5.4.3 Результаты пропуска модернизированной программы.

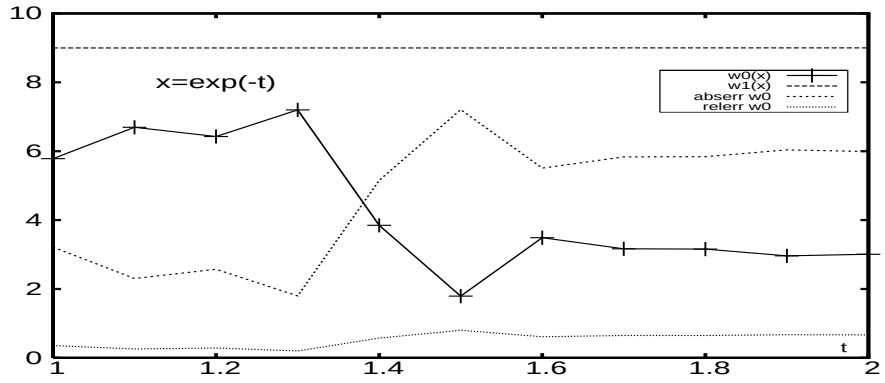
Убедимся, что если при $\mathbf{t} \in [-\mathbf{2}, -\mathbf{1}]$ результаты, полученные обеими функциями $\mathbf{w0}(\mathbf{x})$ и $\mathbf{w1}(\mathbf{x})$ практически совпадают

```
#   t0=  -1.000000         tn=  -2.000000         n=           5
#  i            t             w0             w1            aer           rer
    0     -0.100000E+01  0.886935E+01  0.886936E+01  0.476837E-05  0.537623E-06
    1     -0.120000E+01  0.880782E+01  0.880782E+01  0.190735E-05  0.216552E-06
    2     -0.140000E+01  0.871934E+01  0.871934E+01  0.190735E-05  0.218749E-06
    3     -0.160000E+01  0.859467E+01  0.859466E+01  0.953674E-06  0.110961E-06
    4     -0.180000E+01  0.842466E+01  0.842466E+01  0.190735E-05  0.226401E-06
    5     -0.200000E+01  0.820505E+01  0.820504E+01  0.953674E-06  0.116230E-06
```

то на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{1}, \mathbf{2}]$

```
#   t0=   1.000000         tn=   2.000000         n=          10
#  i            t             w0             w1            aer           rer
    0      0.100000E+01  0.578293E+01  0.899754E+01  0.321461E+01  0.357276E+00
    1      0.110000E+01  0.669445E+01  0.899799E+01  0.230353E+01  0.256006E+00
    2      0.120000E+01  0.642638E+01  0.899835E+01  0.257197E+01  0.285827E+00
    3      0.130000E+01  0.720023E+01  0.899865E+01  0.179842E+01  0.199854E+00
    4      0.140000E+01  0.384725E+01  0.899889E+01  0.515164E+01  0.572475E+00
    5      0.150000E+01  0.179171E+01  0.899909E+01  0.720739E+01  0.800902E+00
    6      0.160000E+01  0.348971E+01  0.899926E+01  0.550955E+01  0.612222E+00
    7      0.170000E+01  0.316620E+01  0.899939E+01  0.583319E+01  0.648176E+00
    8      0.180000E+01  0.315875E+01  0.899950E+01  0.584075E+01  0.649008E+00
    9      0.190000E+01  0.296075E+01  0.899959E+01  0.603885E+01  0.671013E+00
   10      0.200000E+01  0.300834E+01  0.899967E+01  0.599133E+01  0.665728E+00
```

соответствующие результаты существенно различны: функция $\mathbf{w1}(\mathbf{x})$ при $x \ll 1$ получает правильное значение в отличие от функции $\mathbf{w0}(\mathbf{x})$.



Замечание. Исследование работы функций **w0** и **w1** при отрицательных значениях **t** таких, что $\mathbf{x} \gg \mathbf{1}$ не требуется по условию задачи. Однако, анализ правильности соответствующих результатов не менее полезен для уяснения другой причины потери точности.

5.4.4 Примеры GNUPLOT-скриптов.

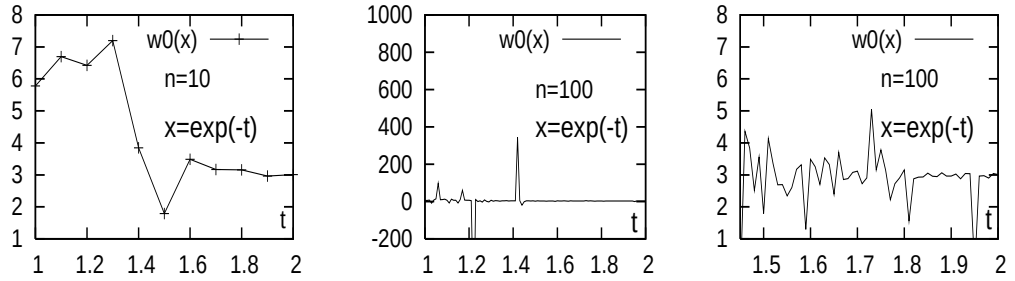
GNUPLOT-скрипт, посредством которого получен рисунок из предыдущего пункта состоит из двух частей: первая выводит изображение на экран, вторая – в файл **result.eps**.

```
set terminal x11                          # установка порта приписки
set output                                # вывод на экран
set border 15 lw 3
set key  1.9, 8.2  box
set label 't'         at  1.97,0.3 center font "Helvetica,18"
set label 'x=exp(-t)' at  1.2 ,8   center font "Helvetica,24"
set xtics font "Helvetica,24"
set ytics font "Helvetica,24"
plot [1:2] [0:10] 'result' u 2:3   title ' w0(x) '  w lp lw 3 ps 3,\
     ''       u 2:4   title ' w1(x) '  w l  lw 3,\
     ''       u 2:5   title ' abserr w0 ' w l lw 3,\
     ''       u 2:6   title ' relerr w0 ' w l lw 3
pause 5 "the picture 1."                  # задержка рисунка на 5 секунд
set terminal postscript eps enhanced      # смена порта приписки
set output "result.eps"                   # имя файла с рисунком
replot                                    # перевод рисунка через новый порт
reset                                     # восстановление настроек умолчания
```

Пояснения:

1. **set terminal x11** – установка типа терминала.
2. **set output** – хотим осуществить вывод рисунка на экран.
3. **set key 1.9, 8.2 box** – хотим разместить в окрестности точки с указанными координатами **легенду**, информирующую о выбранном утилитой стиле черчения графиков. Служебное слово **box** нацеливает утилиту на черечение рамки вокруг легенды.
4. **set label "x=exp(-t)" at 1.2, 8 center font "Helvetica,24"** – намечаем поместить в область рисунка так называемую **метку** (некий уточняющий текст): **x=exp(-t)**. После служебного слова **at** указываем координаты расположения метки, тип шрифта **center font**, его название и размер **Helvetica,24**.
5. **set label "t" at 1.97, 0.3 center font "Helvetica,18"** – хотим поместить в пределах рисунка еще одну метку – имя независимого аргумента **t**.
6. **set xtics font "Helvetica,18"** – задание шрифта оцифровки оси абсцисс
7. **set ytics font "Helvetica,18"** – и оси ординат
8. Команда **plot** выводит в пределах прямоугольной области $\mathbf{x} \in [\mathbf{1}, \mathbf{2}]$ и $\mathbf{y} \in [\mathbf{0}, \mathbf{10}]$ графики четырех функций, соединяя соседние точки каждого графика отрезком прямой и используя в качестве аргумента данные из второго столбца.

GNUPLOT-скрипт, получающий рисунок



из пункта **5.2** (см. стр. 172):

```
set terminal postscript eps enhanced       # смена порта приписки
set output 'result100.eps'
set key
set size  1.0, 0.333     # условный размер области с несколькими рисунками.
set origin 0.0, 0.00     # координаты точки отсчета области.
set multiplot            # режим вывода нескольких рисунков на одну страницу.
  set size 0.333, 0.333                          # Размер левого рисунка;
  set origin 0.0, 0.000                          # коорд. его лев. нижн. угла.
    set label 1 "n=10" at 1.5, 6.0
    set label 4 'x=exp(-t)' at  1.5, 4.5 font "Helvetica,16"
    set label 5 't' at  1.95, 1.6
    plot [1:2] 'result10' using 2:3 title "w0(x)" w lp
    unset label 1                                # ликвидация меток по тэгам.
    unset label 4
    unset label 5
  set size 0.333, 0.333                          # Размер среднего рисунка;
  set origin 0.333, 0.0                          # коорд. его лев. нижн. угла.
    set label 2 "n=100" at 1.5,600
    set label 4 '{/Helvetica=16\x=exp(-t)}' at  1.5, 400     # можно и так.
    set label 5 't' at  1.95, -110 center font "Helvetica,16"
    plot [1:2] [-200:1000] 'result100' using 2:3 title "w0(x)" w l
    unset label 2                                # ликвидация меток по тэгам.
    unset label 4
    unset label 5
  set size 0.333, 0.333                          # Размер правого рисунка;
  set origin 0.666, 0.0                          # коорд. его лев. нижн. угла.
    set label 3 "n=100" at 1.75,6
    set label 4 '{/Helvetica=16\x=exp(-t)}' at  1.75, 4.5
    set label 5 't' at  1.98,1.6 center font "Helvetica,16"
    plot [1.45:2] [1:8]'result100' using 2:3  title " w0(x)"w l
    unset label 3
    unset label 4
    unset label 5
    unset multiplot              # закрытие режима многорисуночного вывода
reset
```

Пояснения:

1. Если не предполагается анимации, то нет особой нужды выводить рисунок на экран через терминал **x11** – можно обойтись **postscript**-терминалом, используя для обзора утилиту **gv**.

2. **set size 1.0, 0.333** – установка размера области с тремя рисунками. По умолчанию **GNUPLOT** под любой рисунок отводит область единичного размера. В данном случае предполагаем разместить по горизонтали три рисунка. Поэтому размер области по ширине должен быть втрое большим чем по высоте:

3. **set origin 0.0, 0.0** – координаты левого нижнего угла области.

4. **set multiplot** – возможность вывода трех различных рисунков достигается установкой режима **multiplot**.

5. **set size 0.333, 0.333** – устанавливаем размер левого рисунка и

6. **set origin 0.0, 0.000** – координаты его левого нижнего угла.

7. **set label 1 "n=10" at 1.5,6.0** – указываем, где разместить метку "n=10" с тэгом **1** (единица – идентификатор метки; он нужен для указания ликвидации метки посредством оператора **unset**, если она портит следующие рисунки).

8. **set label 4 'x=exp(-t)' at 1.5, 4.5 font "Helvetica,16"** – перед словом **font** посредством служебных слов **left**, **right** или **center** можно указать размещение метки относительно заданной точки (по умолчанию метка размещается слева).

9. **plot [1:2] 'result10' using 2:3 title "w0(x)" w lp** – чертим первый рисунок в отведенной ему области.

10. **unset label 1**; **unset label 4**; **unset label 5** — ликвидируем первую, четвертую и пятую метки первого рисунка, чтобы они не портили следующих.

11. **set size 0.333, 0.333** – устанавливаем размер среднего рисунка и

12. **set origin 0.333, 0.000** – координаты его левого нижнего угла.

13. Далее устанавливаем метки второго рисунка, чертим его и снова ликвидируем метки, после чего пишем последовательность аналогичных инструкций для третьего.

14. **unset multiplot** – закрываем многорисуночный вывод (на некоторых терминалах рисунки появляются только после закрытия режима **multiplot**).

15. Обратим внимание, что данные, по которым строятся графики, могут находиться в разных файлах. Так первый рисунок чертится по данным из файла **result10**, а второй и третий – по данным из файла **result100** (файл **result**, получаемый при очередном пропуске программы, переименовывался в **resul10** и в **result100** вручную)

5.4.5 Что знаем и умеем после сдачи пунктов 1-7 контрольной?

1. При программировании помним, что формулы верные аналитически могут обладать свойствами, которые делают их опасными для расчета на ЭВМ.

2. Объективная причина возникновения подобных свойств — конечная разрядность ячеек, из-за которой данное вещественного типа представляется машиной лишь приближенно, как правило, с погрешностью отличной от нуля.

3. Значение погрешности представления данного типа **real*4** меньше значения данного примерно в $\mathbf{10^7}$ раз, а для данного типа **real*8** – в $\mathbf{10^{16}}$ раз. Кажется, что подобными погрешностями можно пренебречь.

4. Однако машинный процесс расчета в силу конечной разрядности ячейки приводит к распространению погрешностей представления исходных данных, а иногда и к их катастрофическому росту в результате чего погрешность результата может на много порядков превысить значение самого результата.

5. Для катастрофической потери точности вовсе не требуется гигантское число арифметических операций. Так за одну операцию вычитания чисел, различющихся лишь в одном-двух младших битах разрядной сетки, теряются почти все верные цифры разности. Например, результат работы ФОРТРАН-программы:

```
write(*,*) 1.2345678-1.2345677  !  равен  1.1920929E-07
end
```

6. Прежде чем программировать формулу для расчета проверяем: “Не встречается ли в ней вычитание почти равных чисел?”. При возможной потере точности конечного результата исходную формулу следует аналитически преобразовать так, чтобы исключить из нее опасные для вычисления на ЭВМ фрагменты.

7. Если, например, вычитаемое (почти равное уменьшаемому) представляет собой результат извлечения кваратного корня, то часто помогает умножение и деление формулы на соответствующее сопряженное выражение так, что после получения разности квадратов уменьшаемого и вычитаемого исключение равных величин можно точно осуществить аналитически, а не поручать ЭВМ.

8. Стандартные приемы раскрытия неопределенностей при вычислении пределов нередко получают формулу аналитически эквивалентную исходной, но свободную от влияния эффектов, связанных с потерей точности. Другими словами, эти приемы имеют не только академическое, но и ценное прикладное значение.

9. Всегда полезно посредством написания **make**-файла автоматизировать этапы генерации исполнимого файла: создание объектных файлов из исходных, компоновки исполнимого файла из объектных, а также вывода результатов в табличном или графическом виде. Мы научились писать соответствующие элементарные **make**-файлы.

10. В элементарных **make**-файлах **явно** указываются имена исходных и объектных файлов, от которых зависит компоновка исполнимого.

11. Узнали, что по синтаксису утилита **make** допускает использование понятия **переменной**. Например, в переменной можно запомнить имя компилятора. Тогда при оформлении нескольких целей получения объектных кодов вместо явного повтора имени конкретного компилятора можно автоматически выбирать это имя из имени переменной.

12. Указание о подобной выборке дается посредством значка **$**, предваряющего имя переменной, заключенное в круглые скобки. Например,

```
compiler:=gfortran
      main : tsfs2p30a.o w0.o w1.o
[ TAB ]      $(compiler) tsfs2p30a.o w0.o w1.o -o main
```

13. Синтаксис утилиты **make** предоставляет средства организовать запись **make**-файла так, чтобы любые имена файлов, входящих в проект, генерировались автоматически. Поэтому **make**-файл проекта, состоящего из достаточно большого количества исходных файлов, может состоять из небольшого числа правил (подробнее см. курс “Операционные системы” или в нашем курсе “Программирование” (второй семестр) главу **Кое-что об утилите make (часть вторая)**).

14. Утилита **gnuplot** позволяет с легкостью выводить в графическом виде результаты, представляемые числовыми таблицами.

15. Утилита **gnuplot** включает в свою библиотеку практически все встроенные функции традиционного ФОРТРАНа и все операции языка СИ, что позволяет ее использовать не только для черчения графиков, но и для ведения расчетов.

16. **gnuplot**-скрипт – это запись в отдельном файле (или файлах) на языке утилиты **gnuplot** команд оформления вывода рисунков. Подав утилите **gnuplot** в качестве аргумента имя **gnuplot**-скрипта, можно автоматизировать запуск и исполнение всех записанных в нём команд, создающих желаемый рисунок.

17. Умеем вывести графики нескольких функций в одну заданную область декартовой системы координат (команда **plot**).

18. Можем график каждой из них поместить в разных частях одной страницы (установка и использование режима **multiplot**).

19. Умеем размещать легенду, задавать размер шрифта, ставить метки, знаем для чего нужен тэг и можем выводить рисунок не только на экран, но и на диск в **eps**-файл.

5.5 ФОРТРАН: решение в режиме real(8)

(8-й пункт плана выполнения контрольной)

1. Исходные файлы.
2. Результаты тестирования функций w0 и w1 типа real(mp).
3. Что знаем после сдачи пункта 8 первой части контрольной?

5.5.1 Исходные файлы.

```
      program tsfd2p3077                        !    Файл  tsfd2p3077.for
      implicit none
      real*8 wd0, wd1, t0, tn, t, ht, x, r0, r1, aer, rer
      integer ninp, nres
      integer n, i
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, 1099) t0, tn, n
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=(t0+i*ht)
        x=dexp(-t)
        r0=wd0(x)
        r1=wd1(x)
        aer=dabs(r0-r1)
        rer=aer/r1
        write(nres,1001) i, t,  r0, r1, aer, rer
      enddo
      close(nres)
  100 format(d15.7)
  101 format(i15)
 1099 format(1x,' #   t0=',d23.15,2x,'tn=',d23.15,2x,'n='i3)
 1100 format(1x,' # ','i',4x,'t',14x,'w0',21x,'w1',11x,'aer',7x,'rer')
 1001 format(1x,i4,d11.3,d21.13,d21.13, d10.3, d10.3)
      end

      function wd0(x)                                 !    Файл wd0.for
      implicit none
      real*8 wd0, x, x2, a, b
      x2=x*x
      a=dcos(x) - 1 + x2* 0.5d0*(1 - x2/12d0 * ( 1 - x2 / 30d0))
      b=dsin(x)/x - 1d0+x2/6d0 * (1-x2/20d0*(1-x2/42d0))
      wd0=a/b
      end
```

```
function wd1(x)                                        !    Файл w1.for
implicit none
real*8 wd1, x, x2, sa1, sa, sb, a, b
integer k, k2
x2=x*x
sa1=1d0; sa=0d0; sb=0d0; a=1d0/40320d0; b=a/9d0
k=0
do while (abs(sa).ne.sa1)
  sa1=abs(sa); k2=2*k
  sa=sa+a; a=-a*x2/(k2+ 9)/(k2+10)
  sb=sb+b; b=-b*x2/(k2+10)/(k2+11)
  k=k+1
enddo
wd1=sa/sb
end
```

1. Встроенные функции, имена которых начинаются с буквы **d** (**dexp**, **dabs**, **dsin**, **dcos** и др.) работают с аргументом и возвращают результат типа **real*8** (соответственно функции **exp**, **abs**, **sin**, **cos** и др. нацелены на работу в режиме **real*4**). Упомянутые имена функций называют *специфическими* [9], [3].

2. В современном ФОРТРАНе наряду со **специфическими** именами имеется понятие и **родового** имени функции, которое совпадает с ее общепринятым математическим обозначением (например, **cos**), но позволяет в зависимости от типа аргумента автоматически вызывать либо **cos** (при аргументе типа **real**(**4**)), либо **dcos** (при аргументе типа **real(8)**), либо **ccos** (при аргументе **complex(4)**), либо **cdcos** (при аргументе **complex(8)**).

3. В ФОРТРАНе-77 при записи константы типа **real*8** наличие суффикса **d** является обязательным. Проанализируйте результат работы программы:

```
write(*,'(d25.16)') 1.25    ! В современном ФОРТРАНе наиболее
write(*,'(d25.16)') 1.3     ! практичная форма записи константы
write(*,'(d25.16)') 1.3d0   ! 1.3 имеет вид 1.3_mp, где mp - имя
write(*,'(d25.16)') 1.3_8   ! целочисленной константы, описанной
write(*,'(d25.16)') 1.3_mp  ! в модуле.
end
```

4. Буква **e** на языке оператора **format** — это ключ (дескриптор) перевода данного типа **real*4** в форме с плавающей запятой в десятичную (при выводе) или в двоичную (при вводе) систему счисления. Разделённые точкой числа **15.7** после литеры **e** означают число позиций, отводимое на размещение данного, и число знаков после запятой соответственно. Для перевода данного типа **real*8** в качестве дескриптора используем букву **d** (вместо **e**).

5.5.2 Результаты тестирования функций w0 и w1 типа real(8)

```
#   t0= -0.200000000000000D+01  tn= -0.100000000000000D+01  n= 10
# i    t              w0                     w1           aer       rer
  0 -0.200D+01 0.82050457490012D+01 0.82050457490012D+01 0.000D+00 0.000D+00
  1 -0.190D+01 0.83209086910438D+01 0.83209086910438D+01 0.000D+00 0.000D+00
  2 -0.180D+01 0.84246621827649D+01 0.84246621827649D+01 0.000D+00 0.000D+00
  3 -0.170D+01 0.85157973388969D+01 0.85157973388969D+01 0.000D+00 0.000D+00
  4 -0.160D+01 0.85946658125882D+01 0.85946658125882D+01 0.178D-14 0.207D-15
  5 -0.150D+01 0.86621361557249D+01 0.86621361557249D+01 0.178D-14 0.205D-15
  6 -0.140D+01 0.87193399234080D+01 0.87193399234080D+01 0.355D-14 0.407D-15
  7 -0.130D+01 0.87674998656048D+01 0.87674998656048D+01 0.355D-14 0.405D-15
  8 -0.120D+01 0.88078222909625D+01 0.88078222909625D+01 0.533D-14 0.605D-15
  9 -0.110D+01 0.88414353401492D+01 0.88414353401492D+01 0.178D-13 0.201D-14

#   t0=  0.100000000000000D+01  tn=  0.200000000000000D+01  n= 10
# i    t              w0                     w1           aer       rer
  0  0.100D+01 0.89975405991527D+01 0.89975406005150D+01 0.136D-08 0.151D-09
  1  0.110D+01 0.89979862235488D+01 0.89979862296740D+01 0.613D-08 0.681D-09
  2  0.120D+01 0.89983511522792D+01 0.89983511407428D+01 0.115D-07 0.128D-08
  3  0.130D+01 0.89986499224847D+01 0.89986499453933D+01 0.229D-07 0.255D-08
  4  0.140D+01 0.89988945776076D+01 0.89988946132559D+01 0.356D-07 0.396D-08
  5  0.150D+01 0.89990949749568D+01 0.89990949486631D+01 0.263D-07 0.292D-08
  6  0.160D+01 0.89992590387076D+01 0.89992589816911D+01 0.570D-07 0.634D-08
  7  0.170D+01 0.89993926465523D+01 0.89993932888000D+01 0.642D-06 0.714D-07
  8  0.180D+01 0.89995031981246D+01 0.89995032556733D+01 0.575D-07 0.639D-08
  9  0.190D+01 0.89995937281365D+01 0.89995932926297D+01 0.436D-06 0.484D-07

#   t0=  0.200000000000000D+01  tn=  0.200000000000000D+02  n=  9
# i    t              w0                     w1           aer       rer
  0  0.200D+01  0.8999670544540D+01  0.8999667011132D+01 0.353D-05 0.393D-06
  1  0.400D+01  0.7867540410660D+01  0.8999993900687D+01 0.113D+01 0.126D+00
  2  0.600D+01 -0.2039099200482D+01  0.8999999888287D+01 0.110D+02 0.123D+01
  3  0.800D+01  0.1115889384465D+01  0.8999999997954D+01 0.788D+01 0.876D+00
  4  0.100D+02 -0.1341669587390D+01  0.8999999999963D+01 0.103D+02 0.115D+01
  5  0.120D+02  0.1297797897008D+00  0.8999999999999D+01 0.887D+01 0.986D+00
  6  0.140D+02 -0.2553472942699D+01  0.9000000000000D+01 0.116D+02 0.128D+01
  7  0.160D+02 -0.5511271468273D+00  0.9000000000000D+01 0.955D+01 0.106D+01
  8  0.180D+02  0.3390110019669D+00  0.9000000000000D+01 0.866D+01 0.962D+00
  9  0.200D+02  0.3000000000000D+01  0.9000000000000D+01 0.600D+01 0.667D+00
```

Переход на тип **real(8)** сместил эффект катастрофической потери точности в сторону значений **t>3.8**, а на одинарной точности он проявлялся, начиная с **t<=1**. **Замечание:** Удвоенная точность включается и без изменения исходных файлов, если при вызове компилятора **gfortran** указана опция **-fdefault-real-8**, по которой описатель **real** и любая константа этого типа (например, 1.3) трактуются компилятором как описатель и константа типа **real(8)** соответственно. Достоинство опции — очевидно. Недостаток — на других компиляторах она может иметь другое имя.

5.5.3 Что знаем после сдачи пункта 8 первой части контрольной?

1. Всегда полезно отменять действие правила умолчания ФОРТРАНа. Делать это можно двояко: либо явно в **каждой** единице компиляции посредством оператора **implicit none**, либо правильно задав соответствующую опцию используемого компилятора.

2. Оператор ФОРТРАНа **data** задает начальные значения переменным на этапе компиляции, но не выполнения программы. Поэтому рабочие переменные **процедур**, инициализированные **data**, НЕ ИНИЦИАЛИЗИРУЮТСЯ им *во время их повторных вызовов*.

3. В СИ и С++ все локальные переменные, то есть описанные внутри функций, по умолчанию являются *автоматическими*. При очередном вызове функции явная инициализация такой переменной будет происходить во время выполнения функции, а не при компиляции, как в случае ФОРТРАН-оператора **data**. Поэтому, при ручном переводе текста С-функции (изменяющей значение инициализированной переменной) на ФОРТРАН необходимо соответствующую ФОРТРАН-инициализацию проводить оператором присваивания.

4. В современном ФОРТРАНе есть понятие **родового имени** процедуры, по которому можно вызвать любую из некоторого семейства процедур, отличающихся типом и/или количеством аргументов. У каждой из таких процедур есть и свое **специфическое имя**. Механизм обеспечения вызова по одному родовому имени разных процедур называется **перегрузкой** процедур.

5. Помним, что в старом ФОРТРАНе единственным способом записи вещественной числовой константы с удвоенной точностью является запись в форме с плавающей запятой и литерой **d** (или **D**), разделяющей мантиссу и порядок, например: **1.8d0**

6. Современный ФОРТРАН допускает и альтернативную форму записи константы с удвоенной точностью, например, **1.8_8** или **1.8e0_8**, где после подчеркивания указывается параметр разновидности (4 – для одинарной точности; 8 – для удвоенной; 16 – для четверной). Однако, удобнее всего форма **1.8_mp**, где **mp** — константа целого типа, описанная в модуле, например,

   ```
   module my_prec
   implicit none
   integer, parameter :: mp=8
   end module my_prec
   ```

   который посредством оператор **use my_prec** подсоединяется к единице компиляции, использующей имя **mp** и для описания переменныцх (например, **real(mp) x**), и для инициализации констант (например, **x=1.8_mp**). Достоинство — простота перевода программного продукта на иную допустимую разрядность типа **real**. Недостаток — подобный способ невозможен на ФОРТРАНе-77.

5.6 Решение контрольной на СИ.

(9-й пункт плана выполнения контрольной)

1. Чуть-чуть о вводе данных из файла и выводе в файл.
2. Одинарная точность.
3. Удвоенная точность.
4. Что знаем после сдачи пункта 9 первой части контрольной?

5.6.1 Чуть-чуть о вводе данных из файла и выводе в файл.

Как известно (см. пункт **1.3.6**), ввод исходных данных в С-программе можно осуществить через устройство стандартного ввода **stdin**, перенастроив его на ввод из из конкретного файла на диске, используя операцию перенаправления (<) операционной системы. Так, если исходные данные хранятся в файле **input**, а исполнимый файл нашей программы имеет имя **main**, то упомянутое перенаправление достигается командой **./main < input**.

Язык С, хотя и не имеет встроенных операторов ввода-вывода (как ФОРТРАН или С++), обладает зато широким набором соответствующих библиотечных функций. В частности, файл <**stdio.h**>, подключаемый оператором **include**, обеспечивает доступ к многочисленным и разнообразным функциям ввода-вывода. В пунктах **1.6.7** и **1.6.11** уже знакомились с функцией **scanf**, которая осуществляла форматизованный ввод из *потока* **stdin**, и **printf**, реализующей форматизованный вывод в *поток* **stdout**.

При решении расчетных задач исходные данные обычно хранятся в файле на диске, что удобно для модификации некоторых из них без изменения остальных. Файловый форматизованный ввод-вывод числовых данных в СИ обеспечивают функции **fscanf** и **fprintf**. Перед их вызовом необходимо, чтобы соответствующий файл был открыт. Открытие файла осуществляется функцией **fopen**, которой на вход подаются два аргумента:

1. имя файла, по которому операционнная система сможет его распознать;
2. способ доступа к файлу (чтение, запись; их режимы)

Код доступа к файлу	Смысловая нагрузка
r	Открытие потока (файла) для чтения.
w	Открытие пустого потока для записи.
a	Открытие потока для записи в конец файла (если файла нет, то он создается)

Ограничимся пока ими (об остальных можно узнать, например, в [14], [18]).

При удачном открытии файла функция **fopen** возвращает через свое имя значение указателя на предопределенную в С структуру типа **FILE**. Описание **FILE *fp;** означает, что переменная **fp** предназначена для хранения указателя на файл. Учитываем, однако, что значение указателя определится лишь при завершении работы функции **fopen**. Для уяснения ситуации рассмотрим программу:

```
#include <stdio.h>
int main()
{FILE *fpi, *fpr; int a; float b, c; double d;
  fpi=fopen("input","r");
  if (fpi==NULL) { printf("Неудача при открытии файла input !!!\n"); return 1;}
  fpr=fopen("result","w");
  if (fpr==NULL){ printf("Неудача при открытии файла result !!!\n"); return 2;}
   fscanf(fpi,"%i %f %e %le", &a, &b, &c, &d);
   fprintf(fpr,"a=%d  b=%f  c=%e   d=%25.15le\n",a, b, c, d);
   fclose(fpr); fclose(fpi);
   return 0;
}
```

Здесь

1. **fpi** - указатель на файл с именем **input** (предполагается, что в этот файл нужно поместить параметры, вводимые программой).

2. **fpr** - указатель на файл с именем **result** (предполагается, что в **result** программа будет выводить результаты).

3. В ФОРТРАНе вместо СИ-переменных **fpi** и **fpr** типа **FILE*** используются целочисленные переменные, хранящие номера устройств ввода-вывода.

4. Второй параметр в обращении к функции **fopen** задает способ доступа к файлу, согласно приведённой ранее таблицы.

5. При наличии файла **input** с исходными данными **1 3.4 3.5e1 7.1e2** в текущей директории после завершения работы программы получим файл **result**: **a=1 b=3.400000 c=3.500000e+01 d= 7.100000000000000e+02**

6. Если же файла с именем **input** в текущей директории нет (например, забыли его создать или назвали не именем **input**), то на экран высветится фраза: **Неудача при открытии файла input !!!** Аналогичное сообщение поступит при открытии файла **result** на запись, если диск переполнен или предпринята попытка записи в файл несуществующей директории.

7. Признаком того, что открытие файла прошло неудачно служит нулевое значение указателя, возвращаемое через имя **fopen**. Поэтому прежде чем начинать чтение или запись следует убедиться, что открытие потока (файла) действительно состоялось. В данной программе проверка выполнена сравнением значения указателя, выработанного функцией **fopen**, с нулевым указателем (константа **NULL**). Впрочем, условие можно записать и короче, например:

   if(!**fpi**){**printf**("**Неудача при открытии input**\n"); **return1**; }

8. Хороший стиль программирования предполагает явный вызов функции закрытия файла (**fclose**(**указатель_на_файл**)), когда работа с файлом завершена:

Не забываем явно закрывать файлы с результатом!

5.6.2 Одинарная точность.

С учетом изложенного в **5.6.1** СИ-решение задачи из **5.2** может иметь вид.

```
#include <stdio.h>                                    //  Файл  tsfs2p30.c
#include "w01.h"
int main()
{  float t0, tn, t, ht, x, r0, r1, aer, rer;
   int n, i;
  FILE *ninp, *nres;
  ninp=fopen("input","r");
  if (ninp==NULL) { printf("Неудача при открытии файла input !!!\n"); return 1;}
  nres=fopen("result","w");
  if (nres==NULL){ printf("Неудача при открытии файла result !!!\n"); return 2;}
   fscanf(ninp,"%e %e %d", &t0, &tn, &n);
   fprintf(nres," #  t0=%e  tn=%e  n=%i\n", t0, tn, n);  ht=(tn-t0)/n;
   fprintf(nres," # %2s %10s %15s %15s %10s %10s\n",
                            "i","t","r0","r1","aer","rer");
   for (i=1; i<=n;i++)
     { t=t0+ht*(i-1);   x=exp(-t); r0=w0(x);  r1=w1(x);
       aer=fabs(r0-r1);
       rer=aer/r1;
       fprintf(nres,"%5i %15.7e %15.7e %15.7e %10.3e %10.3e\n",
                              i, t, r0, r1, aer, rer);
     }
   fclose(ninp);
   fclose(nres);
   return 0;
}

float w0(float x)                                   //    Файл w0.c
{float x2, a, b;
      x2=x*x;
      a=cos(x) - 1 + x2* 0.5*(1 - x2/12 * ( 1 - x2 / 30));
      b=sin(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42));
      return(a/b);
}

float w1(float x)                                    //    Файл w1.c
{float x2, sa1, sa, sb, a, b;
 int k, k2;
 x2=x*x; sa1=1.0; sa=0; sb=0; a=1/40320.0; b=a/9; k=k2=0;
 while (fabs(sa)!=sa1)
   { sa1=fabs(sa);  sa=sa+a; a=-a*x2/(k2+ 9)/(k2+10);
                    sb=sb+b; b=-b*x2/(k2+10)/(k2+11);
     k=k+1; k2=2*k;
   }
 return (sa/sb);
}
```

Замечания:

1. **make**-файл к СИ-программе:

```
compiler:=gcc
 main       : tsfs2p30.o w0.o w1.o
[  TAB  ]    $(compiler) tsfs2p30.o w0.o w1.o -lm -o main
 tsfs2p30.o : tsfs2p30.c
[  TAB  ]    $(compiler) -c tsfs2p30.c
 w0.o       : w0.c
[  TAB  ]    $(compiler) -c w0.c
 w1.o       : w1.c
[  TAB  ]    $(compiler) -c w1.c
 clear      :
[  TAB  ]     rm *.o
 result     : input main
[  TAB  ]     ./main
 restab     : result main
[  TAB  ]     cat result
 resplt     : result main
[  TAB  ]     gnuplot 'view.gnu'
```

2. Результаты **r0** и **r1** совпадают в пределах семи старших цифр с результатами соответствующей ФОРТРАН-программы.

3. **Помним:** исходные ФОРТРАН-данные в файле **input** в форме с порядком можно записать и в виде **1.000000e+00**, и в виде **1.000000+00** (т.е. *без литеры "e" !*), что удобно. В СИ последний вариант (*без литеры "e"*) записи значений в файле с исходными данными **недопустим !!!**

4. Если в приведенной программе при описании прототипа функции **w0** указать тип аргумента отличный от типа аргумента, приведенного в описании функции (например, **double**), то когда описания функций находятся в отдельных файлах, С-компилятор не сообщит, как хотелось бы, о несовпадении типов формального и фактического аргументов (компилятор С++ сообщает).

5. Для решения С-проблемы из пункта 4 используем заголовочный файл **w01.h**:

```
float w0(float);       // Файл w01.h с описанием прототипов
float w1(float);       // (интерфейса) используемых функций.
```

   Посредством инструкции

```
#include "w01.h"
```

   его содержимое подключается к исходному файлу главной программы (перед её заголовком), обеспечивая гарантированный контроль соответствия формальных и фактических аргументов.

5.6.3 Удвоенная точность.

```
#include <stdio.h>                                    //  Файл  tsfd2p30.c
#include "w01d.h"
int main()
{  double t0, tn, t, ht, x, r0, r1;
   int n, i;
  FILE *ninp, *nres;
  ninp=fopen("input","r");
  if (ninp==NULL) {printf("Неудача при открытии файла input !!!\n");return 1;}
  nres=fopen("result","w");
  if (nres==NULL) {printf("Неудача при открытии файла result  !\n");return 2;}
   fscanf(ninp,"%le %le %d", &t0, &tn, &n);
   fprintf(nres," #  t0=%le  tn=%le  n=%i\n", t0, tn, n);    ht=(tn-t0)/n;
   fprintf(nres," # %2s %10s %17s %23s\n","i","t","r0","r1");
   for (i=1; i<=n;i++)
     { t=t0+ht*(i-1); x=exp(-t);
       r0=w0(x);
       r1=w1(x);
       fprintf(nres,"%5i %15.7e %23.16le %23.16le\n", i, t, r0, r1);
     }
   fclose(ninp); fclose(nres); return 0;
}
```

```
#include "w01d.h"
double w0(double x)                                    //    Файл w0.c
{double x2, a, b;
      x2=x*x; a=cos(x) - 1 + x2* 0.5*(1 - x2/12 * ( 1 - x2 / 30));
              b=sin(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42));
      return(a/b);
}
```

```
#include "w01d.h"
double w1(double x)                                     //    Файл w1.c
{double x2, sa1, sa, sb, a, b;
 int k, k2;
 x2=x*x; sa1=1.0; sa=0; sb=0; a=1/40320.0; b=a/9;
 k=k2=0;
 while (fabs(sa)!=sa1)
   { sa1=fabs(sa); sa=sa+a; a=-a*x2/(k2+ 9)/(k2+10);
                   sb=sb+b; b=-b*x2/(k2+10)/(k2+11);
     k=k+1; k2=2*k;
   }
 return (sa/sb);
}
```

```
double w0(double);                                    // Файл   w01d.h
double w1(double);
```

5.6.4 Что знаем после сдачи пункта 9 первой части контрольной?

1. Несмотря на наличие операции перенаправления потоков ввода/вывода, полезно уметь организовывать ввод/вывод и средствами языка программирования.

2. В языке СИ для форматного ввода числовых данных из файла служит функция **fscanf**, а для аналогичного вывода функция **fprintf**. В отличие от функций **scanf** и **printf** их первым аргументом служит переменная, хранящая указатель на соответствующий файл.

3. Тип структуры, на которую ссылается эта переменная, при описании задается служебным словом **FILE**.

4. Для инициализации ее содержимого, т.е. для помещения в нее указателя на реальный файл операционной системы (точнее для открытия потока и связывания его с конкретным файлом на диске), служит функция **fopen**, которая имеет два аргумента типа **указатель** на строку: первый указывает на строку с именем файла (возможно и с полным указанием пути к файлу), второй – на строку, определяющую режим использования.

5. Если в процессе открытия файла обнаружится ошибка (например, нет места диске или опечатка в строке, задающей имя файла), то функция **fopen** возвратит в качесте результата указатель **NULL**.

6. Поэтому, прежде чем читать из файла или писать в файл полезно убедиться, что открытие завершено корректно (что указатель возвращаемый **fopen** не равен **NULL**).

7. В остальном работа с **fscanf** и **fprintf** аналогична работе с **scanf** и **printf**.

8. Перед завершением работы программы открытый файл **необходимо** закрыть. Закрытие файла открытого **fopen** осуществляется функцией **fclose**, которая выгружает содержимое буфера вывода в файл и разрывает связь с потоком [18]. Если закрытие произошло успешно, то **fclose** возвращает **NULL**.

9. Если программа состоит из большого количества файлов, хранящих описания каких-то функций, в которых, возможно, есть обращения к функциям, описания которых находятся в других файлах, то полезно интерфейсную часть этих функций (описание прототипов) поместить в заголовочный файл, содержимое которого будет подсоединяться к нужным программным единицам посредством оператора

   ```
   #include "имя header-файла.h"
   ```

   Это обеспечит автоматический контроль типов аргументов функций.

6 Описания интерфейса в ФОРТРАНе-95.

Интерфейс – совокупность сведений необходимая программе для вызова процедуры (см. [7], а также **3.2.3**): имя процедуры, ее вид (функция или подпрограмма), имена и свойства её формальных параметров. Сообщая **интерфейс** программе, даём компилятору возможность отреагировать на несоответствие свойств формальных и фактических параметров, если таковое встретится (в языке СИ описанием интефейса служит описание прототипов функций). Знакомство с основными вариантами описания интерфейса в ФОРТРАНе-95, проведем на примере уже решенной задачи первой контрольной.

6.1 Неявная форма интерфейса.

Прежние ФОРТРАН-компиляторы использовали только неявную форму интерфейса. Она допустима и в ФОРТРАНе-95. Работая с ФОРТРАН-программами первой части контрольной, мы в этом уже убедились.

Допустим, что, работая над программой, ведущей расчет на одинарной точности, случайно сопоставили переменной **x** главной программы ошибочный тип **real*8**.

```
      program tsfs2p30951a                          ! Файл  tsfs2p30951a.for
      implicit none
      real w0, w1, t0, tn, t, ht, r0, r1, aer, rer
      real*8 x                                      ! <--=    наша опечатка
      integer ninp / 5 /, nres / 6 /, n, i
      open(unit=ninp, file='input');
      open(unit=nres, file='result', status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n;
      write(nres,1100)
      do i=0,n
        t=(t0+i*ht)
        x=exp(-t)
        write(nres,*) ' main: x=',x;
        r0=w0(x);   r1=w1(x);   aer=abs(r0-r1); rer=aer/r1;
        write(nres,1001) i, t, r0, r1, aer, rer
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0',14x,'w1',7x,'aer',7x,'rer')
 1001 format(1x,i5,2x,2x,e15.7,e15.7,e15.7,e11.3,e11.3)
      end
```

Компиляция пройдет успешно и будет создан исполнимый файл **main**. Однако после его запуска его (в зависимости от типа и версии компилятора) результат окажется

загадочным: в одних случаях возникает подозрение, что программа циклится; в других – результат, получаемый ЭВМ, оказывается *с ее точки зрения* НЕ ЧИСЛОМ (вместо числа печатается имя **NaN** – Not a Number).

Начнется поиск ошибок в формулах, а их там нет. Сейчас-то ошибку заметить просто. Однако в большой программе поиск подобной опечатки может занять много времени. Например, ограничивая число повторов уточняющего цикла и помещая отладочные печати (в главной программе — **write(6,*) ' main: x=',x** после **x=exp(-t)**, а в функции **w1(x)** — **write(6,*) ' w1: x=',x** перед **x2=x*x**) можем получить следующий результат (фрагмент):

```
 main: x=  0.367879450321198            <--=  Подала главная программа!
  w1:  x=   2.000000                    <--=  Воспринята функция w1 !!!
      0                        ...
 main: x=  0.332871079444885            <--=  Подала главная программа!
  w1:  x=   0.000000                    <--=  Воспринята функция w1 !!!
      1                        ...
 main: x=  0.301194190979004       <--=  Подала главная программа!
  w1:  x=   0.000000               <--=  Воспринята функция w1 !!!
      2                        ...
 main: x=  0.272531807422638       <--=  Подала главная программа!
  w1:  x=   2.000000               <--=  Воспринята функция w1 !!!
      3                        ...
 main: x=  0.246596977114677       <--=  Подала главная программа!
  w1:  x=  3.6893488E+19           <--=  Воспринята функция w1 !!!
      4      0.1400000E+01            NaN            NaN
```

Налицо факт: из-за несовпадения типов формального и фактического аргументов функция воспринимает совсем не то значение аргумента, которое подает ей главная программа. Дело в том, что в качестве аргумента функция **w1(x)**, считающая **x** четырехбайтовым, выбирает четырехбайтовую часть восьмибайтового данного, поданного главной программой. Поскольку на порядок числа четырехбайтовое и восьмибайтовое представления отводят разное количество бит, то и порядок, и мантисса принятого четырехбайтового данного будут отличны от порядка и мантиссы значения, поданного главной программой. Особо впечатляет $\mathbf{x = 3.68 \cdot 10^{+19}}$.

Ясно, что поиск подобных ошибок – весьма трудоемкий процесс.

Таким образом, хотя программа, написанная в стиле ФОРТРАНа-77, без каких-либо изменений проходит и на ФОРТРАНе-95, полезно знать, что последний позволяет оформить запись ее исходного кода в более наглядных и удобных для поиска ошибок, тестирования и модификации формах:

1. явное указание интерфейса;
2. модульный интерфейс;
3. интерфейс внутренних функций;
4. перегрузка интерфейса

Рассмотрим перечисленные способы путем соответствующей модификации решения задачи первой контрольной.

6.2 Варианты решений в стиле ФОРТРАНа-95.

1. Указание явного интерфейса функций.
2. Модульный интерфейс.
3. Интерфейс внутренних функций.
4. Решение с удвоенной точностью.
5. Пример решения с перегрузкой функций.
6. Передача через модуль только интерфейса.
7. О чем узнали из шестой главы?

6.2.1 Указание явного интерфейса функций.

Явный интерфейс внешних процедур задается посредством интерфейсного блока:

```
interface
                тело интерфейса
end interface
```

Тело интерфейса должно содержать описание интерфейсов используемых процедур. Обычно интерфейс процедуры состоит из ее заголовка, указания свойств ее формальных аргументов, типа значения возвращаемого через имя функции (*если процедура является функцией*) и оператора **end**, завершающего интерфейс данной процедуры.

```
      program tsfs2p30951                       !   Файл  tsfs2p30951.for
      implicit none
      interface
        function w0(x); real x, w0; end function w0
        real function w1(x); real x; end function w1
      end interface
      real t0, tn, t, ht, x, r0, r1, aer, rer
      integer ninp, nres, n, i
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n;
      write(nres,1100)
      do i=0,n;  t =(t0+i*ht); x=exp(-t); r0=w0(x);    r1=w1(x)
        aer=abs(r0-r1); rer=aer/r1
        write(nres,1001) i, t, r0, r1, aer, rer
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0',14x,'w1',7x,'aer',7x,'rer')
 1001 format(1x,i5,2x,2x,e15.7,e15.7,e15.7,e10.3, e10.3)
      end
```

Если в этой программе сопоставить переменной **x** тип **real(8)**, то еще на этапе компиляции получим:

```
 In file tsfs2p30951.for:19

        t=(t0+i*ht); x=exp(-t); r0=w0(x); r1=w1(x)
                                     1
Error: Type/rank mismatch in argument 'x' at (1)
 In file tsfs2p30951.for:19

        t=(t0+i*ht); x=exp(-t); r0=w0(x); r1=w1(x)
                                               1
Error: Type/rank mismatch in argument 'x' at (1)
make: *** [tsfs2p30951.o] Ошибка 1
```

Аналогично, если в тексте программы ошибочно снабдили функцию **w0** при вызове несколькими фактическими параметрами (например, **r0=w0(x,t)**), то на этапе компиляции получим сообщение

```
 In file tsfs2p30951.for:17
      do i=0,n;  t =(t0+i*ht); x=exp(-t); r0=w0(x,t);    r1=w1(x)
                                                      1
Error: More actual than formal arguments in procedure call at (1)
make: *** [tsfs2p30951.o] Ошибка 1
```

Таким образом, включение интерфейсной части в текст вызывающей программной единицы существенно упрощает работу программиста. Если количество используемых функций исчисляется десятками и интерфейсная часть, размещаемая в главной программе, оказывается раздражающе объемной, то ее можно поместить в отдельный файл, который подсоединить к главной программе, например, оператором **include**.

Замечания:
При описании интерфейса процедуры:

1. имена ее параметров могут отличаться от имен соответствующих формальных параметров, указанных при описании заголовка. Например, интерфейс функций **w0** и **w1** при желании, можно описать и так:

   ```
   interface
     function w0(x); real x, w0; end function w0
     function w1(z); real w1, z; end function w1
   end interface
   ```

   Вспомним, что при описании прототипов функций в СИ имена параметров можно вообще опустить – достаточно указать лишь тип параметра; к сожалению, подобное в ФОРТРАНе не допускается по синтаксису;

2. наряду с описанием параметров можно объявить и локальные переменные процедуры. Например,

```
interface                                                ! Когда это
  function w0(x); real x, w0;       end function w0      ! может быть
  function w1(z); real w1, z, a, b; end function w1      ! выгодно?
end interface
```

3. можно ту же информацию указать иной комбинацией операторов описания:

```
interface                                                ! Когда это
  function w0(x); real x, w0;      end function w0       ! может быть
  real function w1(z); real z; end function w1           ! выгодно?
end interface
```

4. Если тип функции в интерфейсном блоке не совпадает с ее типом в заголовке, то компилятор **gfortran** может не заметить этого:

```
interface                                                 ! В результате
  function w0(x); real x, w0;      end function w0        ! опечатки
  function w1(x); integer w1; real x; end function w1 ! integer w1
end interface                                             ! получим:
```

```
#   t0=   1.000000         tn=   2.000000         n=          10
#  i            t              w0              w1       aer       rer
   0      0.1000000E+01  0.5782932E+01  0.1091565E+10 0.109E+10 0.100E+01
   1      0.1100000E+01  0.6694451E+01  0.1091566E+10 0.109E+10 0.100E+01
   2      0.1200000E+01  0.6426376E+01  0.1091566E+10 0.109E+10 0.100E+01
   3      0.1300000E+01  0.7200233E+01  0.1091566E+10 0.109E+10 0.100E+01
   4      0.1400000E+01            NaN  0.1091566E+10       NaN       NaN
   5      0.1500000E+01            NaN  0.1091567E+10       NaN       NaN
   6      0.1600000E+01            NaN  0.1091567E+10       NaN       NaN
   7      0.1700000E+01            NaN  0.1091567E+10       NaN       NaN
   8      0.1800000E+01            NaN  0.1091567E+10       NaN       NaN
   9      0.1900000E+01            NaN  0.1091567E+10       NaN       NaN
  10      0.2000000E+01            NaN  0.1091567E+10       NaN       NaN
```

Иначе говоря, даже при указании интерфейса желательной реакции на неверное описание типа значения, возвращаемого функцией, нет.

6.2.2 Модульный интерфейс внешних функций.

Кроме явного интерфейса можно использовать модульный интерфейс, когда нужные процедуры описываются в особой единице компиляции, называемой **модулем**. Подсоединение **модуля** посредством оператора **use** к использующей его программе, предоставит ей в неявном виде контролирующие возможности явного интерфейса:

```
      program p30951b                ! Файл  главной программы: p30951b.for
      use myw01                      ! Оператор подсоединения модуля myw01
      implicit none
      real t0, tn, t, x, ht, r0, r1, aer, rer  ! Имена w0 и w1 здесь описывать
      integer ninp, nres, n, i                 ! не надо. Они описаны в модуле.
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input');
      open(unit=nres, file='result', status='replace')
      read(ninp,100)  t0, tn; read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n;  write(nres,1100)
      do i=0,n;  t=(t0+i*ht); x=exp(-t); r0=w0(x); r1=w1(x)
        aer=abs(r0-r1); rer=aer/r1
        write(nres,1001) i, t, r0, r1, aer, rer
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0',14x,'w1',7x,'aer',7x,'rer')
 1001 format(1x,i5,2x,2x,e15.7,e15.7,e15.7, e10.3, e10.3)
      end

      module myw01    !  Файл mymod.for с исходным текстом модуля myw01
      contains
      function w0(x)
        implicit none
        real w0, x2, a, b; real, intent(in) :: x
        x2=x*x; a=cos(x) - 1 + x2* 0.5*(1 - x2/12 * ( 1 - x2 / 30))
                b=sin(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42));
        w0=a/b
      end function w0
      function w1(x)
        implicit none
        real w1, x, x2, sa1, sa, sb, a, b; integer k
        x2=x*x; sa1=1.0; sa=0; sb=0; a=1/40320.0; b=a/9;   k=0
        do while (abs(sa).ne.sa1)
          sa1=abs(sa); sa=sa+a; a=-a*x2/(2*k+ 9)/(2*k+10)
                       sb=sb+b; b=-b*x2/(2*k+10)/(2*k+11); k=k+1
        enddo; w1=sa/sb
      end function w1
      end module myw01
```

Как только в главной программе используем описание **real(8) x**, так на этапе компиляции получим сообщение о несоответствии фактического и формального аргументов при вызовах функций **w0** и **w1**.

Пример **make**-файла, одним из правил которого является создание модуля:

```
comp:=gfortran
main : p30951b.o mymod.o
[ TAB ]   $(comp) p30951b.o mymod.o -o main
  p30951b.o : p30951b.for myw01.mod
[ TAB ]   $(comp) -c p30951b.for
     mymod.o myw01.mod : mymod.for
[ TAB ]   $(comp) -c mymod.for
 clear  :
[ TAB ]   rm *.o
 result : input main
[ TAB ]   ./main
 restab : result main
[ TAB ]   cat result
 resplt : result main
[ TAB ]   gnuplot 'view.gnu'
```

Заметим, что для построения исполнимого файла достаточно наличия соответствующих объектных **p30951b.o** и **mymod.o**. Однако, достижение цели **p30951b.o** зависит не только от наличия исходного файла **p30951b.for**, но и от наличия файла **myw01.mod**, который появляется вместе с объектным **mymod.o** при достижении цели **mymod.o**. Так как цель **p30951b.o** без наличия файла **myw01.mod** недостижима, то имя **myw01.mod** входит в список имён целей, которые достигаются командой компиляции исходного файла с текстом ФОРТРАН-**модуля**. Часто удобно, когда имя файла с модулем (без расширения) совпадает с именем модуля.

Результаты пропуска программы, использующей **модуль** в смысле ФОРТРАНа-90: (см. также пункт **5.4.3**):

```
 #   t0=   1.000000         tn=   2.000000         n=          10
 #  i            t              w0              w1       aer       rer
    0      0.1000000E+01  0.5782932E+01  0.8997540E+01 0.321E+01 0.357E+00
    1      0.1100000E+01  0.6694451E+01  0.8997986E+01 0.230E+01 0.256E+00
    2      0.1200000E+01  0.6426376E+01  0.8998350E+01 0.257E+01 0.286E+00
    3      0.1300000E+01  0.7200233E+01  0.8998650E+01 0.180E+01 0.200E+00
    4      0.1400000E+01  0.3847253E+01  0.8998894E+01 0.515E+01 0.572E+00
    5      0.1500000E+01  0.1791706E+01  0.8999095E+01 0.721E+01 0.801E+00
    6      0.1600000E+01  0.3489712E+01  0.8999260E+01 0.551E+01 0.612E+00
    7      0.1700000E+01  0.3166201E+01  0.8999393E+01 0.583E+01 0.648E+00
    8      0.1800000E+01  0.3158754E+01  0.8999504E+01 0.584E+01 0.649E+00
    9      0.1900000E+01  0.2960747E+01  0.8999595E+01 0.604E+01 0.671E+00
   10      0.2000000E+01  0.3008335E+01  0.8999667E+01 0.599E+01 0.666E+00
```

6.2.3 Интерфейс внутренних функций.

В ФОРТРАНе-77 все единицы компиляции – внешние. Их имена – **глобальны**. Внешнюю процедуру всегда можно вызвать из любой единицы компиляции, входящей в проект задачи (лишь бы нерекурсивно).

Иногда некоторые из процедур вызываются лишь внутри какой-то одной подпрограммы. ФОРТРАН-95 позволяет оформить их **внутренниими** по отношению к ней, так что вызов их из других единиц компиляции невозможен. Тем самым, вся программа в целом оказывается более структурированной (подчиненные алгоритмы не мешаются среди процедур более высокого уровня иерархии). Например,

```
      program p30951c
      implicit none
      real  t0, tn, t, x, ht, r0, r1, aer, rer;
      integer ninp /5/, nres /6/, n, i
      open(unit=ninp, file='input');
      open(unit=nres, file='result', status='replace')
      read(ninp,100)  t0, tn;  read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n; write(nres,1100)
      do i=0,n; t=(t0+i*ht); x=exp(-t); r0=w0(x); r1=w1(x)
                aer=(r0-r1); rer=aer/r1
                write(nres,1001) i, t, r0, r1, aer, rer
      enddo;    close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0',14x,'w1',8x,'aer',8x,'rer')
 1001 format(1x,i5,2x,2x,e15.7,e15.7,e15.7,e11.3,e11.3)
      contains                      ! Раздел описания внутренних функций
        function w0(x)              !                 главной программы:
        implicit none; real  w0, x, x2, a, b
          x2=x*x; a=cos(x) - 1 + x2* 0.5*(1 - x2/12 * ( 1 - x2 / 30))
                  b=sin(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42))
          w0=a/b
        end function w0
        function w1(x)
        implicit none; real w1, x, x2, sa1, sa, sb, a, b;  integer k
          x2=x*x; sa1=1.0; sa=0; sb=0; a=1/40320.0; b=a/9; k=0
          do while (abs(sa).ne.sa1)
            sa1=abs(sa); sa=sa+a; a=-a*x2/(2*k+ 9)/(2*k+10)
                         sb=sb+b; b=-b*x2/(2*k+10)/(2*k+11); k=k+1
          enddo; w1=sa/sb
        end function w1
      end
```

w0 и **w1** нельзя откомпилировать независимо от главной программы и вызывать из других единиц компиляции. Однако, если используем в главной программе ошибочное **real(8) x**, то как и при наличии явного интерфейса, получим сообщение об ошибке.

6.2.4 Решение с удвоенной точностью.

Переход на удвоенную точность сводится к замене в каждой программной единице описания **real** на **real(8)**, переопределения вещественных констант в форму **D** и, возможно, изменению формата вывода. Здесь приведено решение с явным указанием интерфейса:

```
      program tsfd2p30951                        !   Файл  tsfd2p30951.for
      implicit none
      interface
        function w0(x); real(8) x, w0; end function w0
        function w1(x); real(8) w1, x; end function w1
      end interface
      real(8) t0, tn, t, ht, x, r0, r1
      integer ninp / 5 /, nres / 6 /, n, i
      open(unit=ninp, file='input'); open(unit=nres, file='result')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' # t0=',t0,'  tn=',tn,' n=',n; ht=(tn-t0)/n;
      write(nres,1100)
      do i=0,n;  t=(t0+i*ht); x=exp(-t); r0=w0(x); r1=w1(x)
                 write(nres,1001) i, t, r0, r1
      enddo; close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',16x,'t',22x,'w0',22x,'w1')
 1001 format(1x,i5,2x,2x,d23.15,d23.15,d23.15)
      end

      function w0(x)                                 !    Файл w0.for
      implicit none
      real(8) x, x2, w0, a, b
      x2=x*x; a=cos(x) - 1 + x2* 0.5d0*(1 - x2/12 * ( 1 - x2 / 30))
              b=sin(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42))
      w0=a/b
      end


      function w1(x)                                 !    Файл w1.for
      implicit none
      real(8) w1, x, x2, sa1, sa, sb, a, b
      integer k
      x2=x*x; sa1=1d0; sa=0d0; sb=0d0; a=1/40320d0; b=a/9d0; k=0
      do while (abs(sa).ne.sa1)
        sa1=abs(sa); sa=sa+a; a=-a*x2/(2d0*k+ 9d0)/(2*k+10d0)
                     sb=sb+b; b=-b*x2/(2d0*k+10d0)/(2*k+11d0); k=k+1
      enddo
      w1=sa/sb
      end
```

6.2.5 Пример решения с перегрузкой функций.

Пусть в пределах одной программы нужен вызов каждой из функций (**w0(x)** и **w1(x)**) для работы как в режиме одинарной, так и удвоенной точности. Решение очевидно: описать четыре функции с разными именами (например, **w0(x)**, и **w1(x)** для одинарной; **dw0(x)** и **dw1(x)** для удвоенной), которыми и пользоваться.

Однако, ФОРТРАН-95 позволяет описать интерфейс **w0(x)** и **dw0(x)** так, чтобы по одному имени (например, **w0**) в зависимости от типа (и/или количества) аргументов вызывалась нужная. Подобный механизм называют **перегрузкой** функций.

```
      program tsfd2p30952                        !   Файл  tsfd2p30952.for
      implicit none
      interface w0
        function  w0(x); real    w0, x; end function w0
        function dw0(x); real(8) dw0, x; end function dw0
      end interface w0
      interface w1
        function  w1(x); real    w1, x; end function w1
        function dw1(x); real(8) dw1, x; end function dw1
      end interface w1
      real(8) t0, tn, t, ht, x, r08, r18
      real                  z, r04, r14
      integer ninp, nres, n, i
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn; read(ninp,101) n
      write(nres, *) ' # t0=',t0,'  tn=',tn,' n=',n
      ht=(tn-t0)/n;
      write(nres,1100)
      do i=0,n; t=(t0+i*ht); x=exp(-t); r08=w0(x); r18=w1(x);
                z=x;         r04=w0(z); r14=w1(x)
             write(nres,1001) i, t, r08, r18, r04, r14
      enddo; close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',9x,'t',11x,'r08',11x,'r18',11x,
     >                                  'r04',11x,'r14')
 1001 format(1x,i5,1x,e14.6,e14.6,e14.6, e14.6,e14.6)
      end
```

Здесь при описании интерфейса указано, что механизм перегрузки специфических функций **w0** и **dw0** должен включатся по родовому имени **w0**, а механизм перегрузки специфических функций **w1** и **dw1** включается по родовому имени **w1**.

```
function w0(x)                                    !    Файл w0.for
implicit none
real x, x2, w0, a, b                              ! одинарная точность
x2=x*x
a=cos(x) - 1 + x2* 0.5*(1 - x2/12 * ( 1 - x2 / 30))
b=sin(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42))
w0=a/b
end
function dw0(x)                                   ! удвоенная точность
implicit none
real(8) dw0, x, x2, a, b
x2=x*x
a=dcos(x) - 1d0 + x2* 0.5d0*(1d0 - x2/12d0 * ( 1d0 - x2 / 30d0))
b=dsin(x)/x - 1d0+x2/6d0 * (1d0-x2/20d0*(1d0-x2/42d0))
dw0=a/b
end
```

```
function w1(x)                                    !    Файл w1.for
implicit none
real w1, x, x2, sa1, sa, sb, a, b                 ! одинарная точность
integer k
x2=x*x; sa1=1.0; sa=0; sb=0; a=1/40320.0;   b=a/9; k=0
do while (abs(sa).ne.sa1)
  sa1=abs(sa); sa=sa+a; a=-a*x2/(2*k+ 9)/(2*k+10)
               sb=sb+b; b=-b*x2/(2*k+10)/(2*k+11); k=k+1
enddo
w1=sa/sb
end
function dw1(x)                                   ! удвоенная точность
implicit none
real(8) dw1, x, x2, sa1, sa, sb, a, b
integer k
x2=x*x; sa1=1d0; sa=0d0; sb=0d0; a=1/40320d0;   b=a/9d0; k=0
do while (abs(sa).ne.sa1)
  sa1=abs(sa); sa=sa+a; a=-a*x2/(2d0*k+ 9d0)/(2d0*k+10d0)
               sb=sb+b; b=-b*x2/(2d0*k+10d0)/(2d0*k+11d0); k=k+1
enddo
dw1=sa/sb
end
```

Здесь просто описаны четыре нужные функции. Любую из первых двух за счет описания имени интерфейса в главной программе можно вызывать используя механизм перегрузки по имени **w0**, а любую из второй пары по имени **w1**, хотя при желании можно использовать и их специфические имена. На следующей странице приведены результаты пропуска главной программы для различных начальных данных.

```
# t0=  -2.00000000000000        tn=  -1.00000000000000       n=          10
#  i         t           r08           r18           r04           r14
   0  -0.200000E+01  0.820505E+01  0.820505E+01  0.820505E+01  0.820504E+01
   1  -0.190000E+01  0.832091E+01  0.832091E+01  0.832091E+01  0.832091E+01
   2  -0.180000E+01  0.842466E+01  0.842466E+01  0.842466E+01  0.842466E+01
   3  -0.170000E+01  0.851580E+01  0.851580E+01  0.851580E+01  0.851580E+01
   4  -0.160000E+01  0.859467E+01  0.859467E+01  0.859467E+01  0.859466E+01
   5  -0.150000E+01  0.866214E+01  0.866214E+01  0.866214E+01  0.866214E+01
   6  -0.140000E+01  0.871934E+01  0.871934E+01  0.871934E+01  0.871934E+01
   7  -0.130000E+01  0.876750E+01  0.876750E+01  0.876750E+01  0.876750E+01
   8  -0.120000E+01  0.880782E+01  0.880782E+01  0.880782E+01  0.880782E+01
   9  -0.110000E+01  0.884144E+01  0.884144E+01  0.884144E+01  0.884144E+01
  10  -0.100000E+01  0.886936E+01  0.886936E+01  0.886935E+01  0.886936E+01

# t0=   1.00000000000000        tn=   2.00000000000000       n=          10
#  i         t           r08           r18           r04           r14
   0   0.100000E+01  0.899754E+01  0.899754E+01  0.578293E+01  0.899754E+01
   1   0.110000E+01  0.899799E+01  0.899799E+01  0.669445E+01  0.899799E+01
   2   0.120000E+01  0.899835E+01  0.899835E+01 -0.331333E+01  0.899835E+01
   3   0.130000E+01  0.899865E+01  0.899865E+01  0.720023E+01  0.899865E+01
   4   0.140000E+01  0.899889E+01  0.899889E+01  0.390580E+01  0.899889E+01
   5   0.150000E+01  0.899909E+01  0.899909E+01  0.179171E+01  0.899909E+01
   6   0.160000E+01  0.899926E+01  0.899926E+01  0.348971E+01  0.899926E+01
   7   0.170000E+01  0.899939E+01  0.899939E+01  0.287029E+01  0.899939E+01
   8   0.180000E+01  0.899950E+01  0.899950E+01  0.315875E+01  0.899950E+01
   9   0.190000E+01  0.899959E+01  0.899959E+01  0.296075E+01  0.899959E+01
  10   0.200000E+01  0.899967E+01  0.899967E+01  0.300834E+01  0.899967E+01

# t0=   4.00000000000000        tn=   20.0000000000000       n=           8
#  i         t           r08           r18           r04           r14
   1   0.400000E+01  0.786754E+01  0.899999E+01  0.299993E+01  0.899999E+01
   2   0.600000E+01 -0.203910E+01  0.900000E+01  0.300005E+01  0.900000E+01
   3   0.800000E+01  0.111589E+01  0.900000E+01  0.299996E+01  0.900000E+01
   4   0.100000E+02 -0.134167E+01  0.900000E+01  0.300250E+01  0.900000E+01
   5   0.120000E+02  0.129780E+00  0.900000E+01  0.299984E+01  0.900000E+01
   6   0.140000E+02 -0.255347E+01  0.900000E+01  0.300000E+01  0.900000E+01
   7   0.160000E+02 -0.551127E+00  0.900000E+01 -0.555556E+00  0.900000E+01
   8   0.180000E+02  0.339011E+00  0.900000E+01  0.338916E+00  0.900000E+01
   9   0.200000E+02  0.300000E+01  0.900000E+01  0.300000E+01  0.900000E+01
```

Как видно, в зависимости от типа аргумента родовое имя **w0** вызывает функцию, работающую соответственно в режиме либо **real(4)**, либо **real(8)** (аналогично работает и родовое имя **w1**). Если аргумент находится в безопасной для расчёта зоне, все функции дают практически одинаковый результат (верхняя таблица). При аргументе $\mathbf{t} \in [\mathbf{1}, \mathbf{2}]$ алгоритм **w0** дает приемлемый результат только на типе **real(8)**, т.е. когда работает **dw0**, вызванная через имя **w0**, посредством механизма перегрузки (средняя таблица). Наконец, при $\mathbf{t} \in [\mathbf{4}, \mathbf{20}]$ алгоритм **w0** не справляется с поставленной задачей даже в режиме удвоенной точности, хотя алгоритм **w1** в пределах шести значащих цифр мантиссы дает верный результат и на одинарной.

6.2.6 Передача через модуль явного интерфейса.

Когда описание интерфейса достаточно обширно или должно быть указано в нескольких единицах компиляции, то его непосредственное размещение в теле каждой из них неудобно. В таких случаях описание интерфейса выгодно разместить в отдельном файле, который подсоединять к нужной программной единице посредством оператора **include**. ФОРТРАН-95 предоставляет также возможность поместить описание интерфейса внутри **модуля** (без раздела **contains**). Например:

```
      program tsfd2p30953                        !   Файл  tsfd2p30953.for
      use myinter
      implicit none
      real*8 t0, tn, t, ht, x, r08, r18
      real                  z, r04, r14
      integer ninp, nres, n, i;  data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn; read(ninp,101) n
      write(nres, *) ' # t0=',t0,'  tn=',tn,' n=',n
      ht=(tn-t0)/n;  write(nres,1100)
      do i=0,n
        t=(t0+i*ht); x=exp(-t); r08=w0(x); r18=w1(x);
                     z=x;       r04=w0(z); r14=w1(x)
             write(nres,1001) i, t, r08, r18, r04, r14
      enddo;  close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',9x,'t',11x,'r08',11x,'r18',11x,
     >                                  'r04',11x,'r14')
 1001 format(1x,i5,1x,e14.6,e14.6,e14.6, e14.6,e14.6)
      end

      module myinter
      interface w0
        function  w0(x); real    w0, x; end function w0
        function dw0(x); real*8 dw0, x; end function dw0
      end interface w0
      interface w1
        function  w1(x); real    w1, x; end function w1
        function dw1(x); real*8 dw1, x; end function dw1
      end interface w1
      end module myinter
```

Тогда переход в древний режим неявного интерфейса гарантируется комментированием строки с оператором **use**, который подключает модуль с интерфейсом, а переход в контролирующий режим ФОРТРАНа-95 – ее раскомментированием. Безусловно, вместо оператора **use** можно воспользоваться оператором **include**, подключающим файл с интерфейсом, хотя при этом начисто лишаем себя практики эксплуатации модулей современного ФОРТРАНа.

6.3 Еще один пример использования перегрузки функций.

Рассмотрим еще один пример. В параграфе **3.1.3** была описана функция **fun(x)**,

```
      function fun(x)
      use my_prec
      implicit none
      real(mp) fun, x
      fun=2.0_mp*x-3.0_mp
      return
      end
```

которая для аргумента **x** типа **real*8** вычисляла значение **2*x-3** и возвращала его через имя **fun** в вызывающую программу. Пусть в рабочей программе требуется расчет той же функции и для аргументов типа **real**, **complex** и **complex*8**. Обращение к функции **fun** удовлетворит компилятор только в случае, если тип ее аргумента именно **real*8**. Действительно, попытка откомпилировать программу

```
      program tstinter
      implicit none
      interface
        function fun(x); real*8 x, fun; end function fun
      end interface
      real*4 x4; real*8 x8; complex*8 c4; complex*16 c8
      x4=1; write(*,1002) x4, fun(x4);
      x8=1; write(*,1003) x8, fun(x8)
      c4=(1.0,0.0);  write(*,1004) c4, fun(c4)
      c8=(1d0,0d0);  write(*,1005) c8, fun(c8)
 1002 format(1x,'real*4 : ', e15.7,'...', e15.7)
 1003 format(1x,'real*8 : ',d25.17,'...',d25.17)
 1004 format(1x,'complex*8 : ',e10.3,e10.3,'...',e15.7,e15.7)
 1005 format(1x,'complex*16: ',d10.3,d10.3,'...',d25.17,d25.17)
      end
```

приведет после вызова **gfortran -c tstinter.for** к

```
 In file tstinter.for:15
       i=1;          write(*,1001)  i, fun(i)
                                           1
Error: Type/rank mismatch in argument 'x' at (1)
 In file tstinter.for:16
      x4=1;          write(*,1002) x4, fun(x4)
                                           1
Error: Type/rank mismatch in argument 'x' at (1)
 In file tstinter.for:18
      c4=(1.0,0.0);  write(*,1004) c4, fun(c4)
                                           1
Error: Type/rank mismatch in argument 'x' at (1)
 In file tstinter.for:19
```

```
     c8=(1d0,0d0);  write(*,1005) c8, fun(c8)
                                    1
Error: Type/rank mismatch in argument 'x' at (1)
```

Опишем функции **fun4**, **fun8**, **cfun4** и **cfun8**, тип которых совпадает с типом их аргумента **x**, то есть:

```
     function fun4(x)   !                      Файл  fun4.for
     implicit none      ! Функция fun4(x) для аргумента x,
     real*4 fun4, x     ! заданного с одинарной точностью
     fun4=2.0*x-3.0     ! вычисляет значение функции 2*x-3.
     end

     function fun8(x)   !                      Файл  fun8.for
     implicit none      ! Функция fun8(x) для аргумента x,
     real*8 fun8, x     ! заданного с удвоенной точностью
     fun8=2d0*x-3d0     ! вычисляет значение функции 2d0*x-3d0.
     end

     function cfun4(x)  !                               Файл cfun4.for
     implicit none      ! Функция cfun4(x) для комплексного аргумента,
     complex*8 cfun4, x ! x, заданного с одинарной точностью, вычисляет
     cfun4=2.0*x-3.0    ! комплексное значение функции 2.0*x-3.0.
     end

     function cfun8(x)   !                               Файл cfun8.for
     implicit none       ! Функция cfun8(x) для комплексного аргумента,
     complex*16 cfun8, x ! x, заданного с одинарной точностью, вычисляет
     cfun8=2d0*x-3d0     ! комплексное значение функции 2.0*x-3.0.
     end
```

Тестирующая их программа может выглядеть так:

```
     program tstintrf     ! Тестирующая их программа может выглядеть
     implicit none        ! так:
     interface
       function fun4(x);  real*4     x, fun4;  end function fun4
       function fun8(x);  real*8     x, fun8;  end function fun8
       function cfun4(x); complex*8  x, cfun4; end function cfun4
       function cfun8(x); complex*16 x, cfun8; end function cfun8
     end interface
     real*4 x4; real*8 x8; complex*8 c4; complex*16 c8
     x4=1; write(*,1002) x4, fun4(x4);
     x8=1; write(*,1003) x8, fun8(x8)
     c4=(1.0,0.0);  write(*,1004) c4, cfun4(c4)
     c8=(1d0,0d0);  write(*,1005) c8, cfun8(c8)
1002 format(1x,'real*4 : ', e15.7,'...', e15.7)
1003 format(1x,'real*8 : ',d25.17,'...',d25.17)
1004 format(1x,'complex*8 : ',e10.3,e10.3,'...',e15.7, e15.7)
1005 format(1x,'complex*16: ',d10.3,d10.3,'...',d25.17,d25.17)
     end
```

Результаты работы программы:

```
real*4 :   0.1000000E+01... -0.1000000E+01
real*8 :   0.10000000000000000D+01... -0.10000000000000000D+01
complex*8 :  0.100E+01 0.000E+00... -0.1000000E+01  0.0000000E+00
complex*16:  0.100D+01 0.000D+00...
                -0.10000000000000000D+01  0.00000000000000000D+00
```

При аргументе **x=1** все результаты верны, но **неудобство** налицо: каждое оригинальное имя функции необходимо помнить. Несомненно, что гораздо удобнее, если обращение к любой из них происходит по одному имени, например, **fun**, а фактически работает под этим псевдонимом та, которая выбрана по типу аргумента (то есть **fun4**, если аргумент типа **real*4**; **fun8**, если аргумент типа **real*8** и т.д.). Определим псевдоним **fun** через **именование интерфейсного блока**. Именно,

```
      program tstintrb
      implicit none
      real(4) x4
      real(8) x8
      complex(4) c4
      complex(8) c8
      complex*8  c41
      complex*16 c81
      interface fun      !<--=  РОДОВОЕ ИМЯ для нижеследующих функций:
        function fun4(x);  real(4)    x, fun4;  end function fun4
        function fun8(x);  real(8)    x, fun8;  end function fun8
        function cfun4(x); complex(4)  x, cfun4; end function cfun4
        function cfun8(x); complex(8) x, cfun8; end function cfun8
      end interface
      x4=1;          write(*,1002) x4, fun(x4);
      x8=1;          write(*,1003) x8, fun(x8)
      c4=(1.0,0.0);  write(*,1004) c4, fun(c4)
      c8=(1d0,0d0);  write(*,1005) c8, fun(c8)
      c41=(1.0,0.0);  write(*,1006) c41, fun(c41)
      c81=(1d0,0d0);  write(*,1007) c81, fun(c81)
 1002 format(1x,'real(4) : ', e15.7,'...', e15.7)
 1003 format(1x,'real(8) : ',d25.17,'...',d25.17)
 1004 format(1x,'complex(4) : ',e10.3,e10.3,'...',e15.7,e15.7)
 1005 format(1x,'complex(8) : ',d10.3,d10.3,'...',d25.17,d25.17)
 1006 format(1x,'complex*8  : ',e10.3,e10.3,'...',e15.7,e15.7)
 1007 format(1x,'complex*16 : ',d10.3,d10.3,'...',d25.17,d25.17)
      end
```

Заметим, что описания типа **real*4** (устаревающее) и **real(4)** — эквивалентны, а описания типа **complex*8** (устаревающее) и **complex(8)** — нет.

complex*8 эквивалентно **complex(4)**.

complex*16 эквивалентно **complex(8)**.

6.4 О чем узнали из шестой главы? (кратко)

1. **Интерфейс** – информация об именах, типах и свойствах формальных параметров процедур, помещаемая в использующей их программной единице.

2. Современный ФОРТРАН позволяет явно указывать **интерфейс** используемых функций и подпрограмм, что усиливает контроль за правильностью обращения к ним. Описание интерфейса в СИ – это описание прототипов функций.

3. **Модуль** – единица компиляции ФОРТРАНа-95 (её не было в ФОРТРАНе-77).

4. **Модуль** может состоять из раздела описаний констант, переменных и **интерфейсов**, а также раздела реализации функций и подпрограмм.

5. В ФОРТРАНе-77 нет понятия **глобальной** переменной. Все переменные любой процедуры по отношению к ней **локальны** (не существуют вне процедуры). ФОРТРАН-95 посредством оператора **use** позволяет подключать к программе единицу компиляции вида **модуль**, обеспечивая доступ программы к константам, переменным и процедурам, описанным в **модуле**.

6. На самом деле в ФОРТРАНе-77 есть гибкий (и даже в чем-то удобный) механизм доступа из разных единиц компиляции к одним и тем же ячейкам оперативной памяти (так называемые *общие области* или *common*-блоки; мы их еще не проходили). Однако синтаксис их использования требует гораздо большей внимательности и осторожности, чем синтаксис подключения **модулей**.

7. **Модуль** в ФОРТРАНе-95 позволяет вообще обходиться без **common**-блоков.

8. **Перегрузка функций** – это механизм вызова различных функций по одному имени. Компилятор для выбора нужной функции из нескольких одноименных ориентируется на количество и/или тип ее аргументов при вызове.

9. Встроенные ФОРТРАН-функции **iabs**, **abs**, **dabs**, **cabs**, **cdabs** вычисляют значение абсолютной величины данных типа **integer**, **real**, **real*8**, **complex** и **complex*16** соответственно. Аналогичный набор специфических имен имеют многие встроенные функции ФОРТРАНа. При подаче функции в качестве аргумента данного несоответствующего типа ФОРТРАН-77 сообщает об ошибке.

10. Ясно, что при аргументе любого из указанных выше типов в качестве имени функции удобнее всего использовать одно общепринятое имя: **abs** – в случае абсолютной величины; **sin** – в случае расчета синуса и т.д. Начиная с ФОРТРАНа-90 такая возможность предоставлена. Подобное общепринятое имя в ФОРТРАНе-95 называется **родовым**. Снабжая **интерфейсный блок** *именем*, мы активируем механизм генерации родового имени для всех объявленных в нем функций. Можно перегружать не только процедуры, но и операторы (подробнее, см., например, [9]).

11. К сожалению, родовое имя пока нельзя подать в качестве фактического параметра другой процедуре.

6.5 Восьмое домашнее задание (вторая часть контрольной)

Решить задачу контрольной N 1 на ФОРТРАНе-95:

1. используя в главной программе явное описание интерфейса функций;
2. оформив используемые функции внутренними функциями главной программы;
3. поместив описание используемых функций в **модуль** ФОРТРАНа-95;
4. поместив в **модуль** ФОРТРАНа-95 только интерфейс используемых функций.

Решение каждой задачи разместить в отдельной директории со своим **make**-файлом и **gnuplot**-скриптом.

Письменно сформулировать:

1. какие синтаксические ошибки допускались в процессе отладки?
2. какие трудности возникали при написании **make**-файла (в частности, правила обеспечивающего построение объектного файла программы, подсоединяющей **модуль** ФОРТРАНа-95)?

7 Знакомство с некоторым инструментарием C++.

Язык C++ помимо возможностей языка C, предоставляет свои собственные базовые альтернативные возможности, отсутствующие в C. В этой главе коснемся слегка использования в своих программах **механизма перегрузки функций** и некоторых элементов **объектно-ориентированного инструментария ввода-вывода** (*флаги формата* и *манипуляторы ввода-вывода*).

В приводимых ниже примерах используются понятия класса и объекта. Пока мы не касались основ объектно-ориентированного программирования. Тем не менее вопросы ввода-вывода данных при программировании возникают всегда. Поэтому полезно иметь понятие о соответствующих возможностях объектно-ориентированного инструментария языка C++. Познакомимся с ними, как и в предыдущей главе, на примере решения контрольной задачи.

7.1 СИ-шное решение на C++.

```
#include <iostream>
#include <cmath>
using namespace std;
float w0(float);       //  Файл  tsfs2p30.cpp
float w1(float);
int main()
{ float t0, tn, t, ht, x, r0, r1, aer, rer;
  int n, i;
  FILE *ninp, *nres;
  ninp=fopen("input","r");
  if (ninp==NULL) { printf("Неудача открытия файла input  !\n"); return 1;}
  nres=fopen("result","w");
  if (nres==NULL) { printf("Неудача открытия файла result !\n"); return 2;}
  fscanf(ninp,"%e %e %d", &t0, &tn, &n);
  fprintf(nres," #  t0=%e  tn=%e  n=%i\n", t0, tn, n);
  ht=(tn-t0)/n;
  fprintf(nres," # %2s %10s %15s %15s%11s%11s\n",
                       "i","t","r0","r1","aer","rer");
  for (i=1; i<=n;i++)
    { t=t0+ht*(i-1); x=exp(-t);
      r0=w0(x);
      r1=w1(x);
      aer=abs(r0-r1);
      rer=aer/r1;
      fprintf(nres,"%5i %15.7e %15.7e %15.7e %10.3e %10.3e\n",
                        i, t, r0, r1, aer, rer);
    }
  fclose(ninp);
  fclose(nres);
  return 0;
}
```

В файлах с исходными текстами функций директива #**include** <**math.h**> заменена на #**include** <**cmath**>. Результат пропуска:

```
#  t0=1.000000e+00  tn=1.500000e+00  n=5
#  i          t              r0              r1         aer         rer
   1   1.0000000e+00   5.7829318e+00   8.9975405e+00  3.215e+00  3.573e-01
   2   1.1000000e+00   6.6944509e+00   8.9979858e+00  2.304e+00  2.560e-01
   3   1.2000000e+00   6.4263763e+00   8.9983501e+00  2.572e+00  2.858e-01
   4   1.3000000e+00   7.2002325e+00   8.9986496e+00  1.798e+00  1.999e-01
   5   1.4000000e+00   3.8472526e+00   8.9988937e+00  5.152e+00  5.725e-01
```

Заметим, что в исходной главной **C++**-программе абсолютная погрешность **aer** значения, найденного **w0**, вычисляется посредством использования встроенной функции **abs** (*aer=abs(r0-r1)*), в то время как в аналогичной **C**-программе (см. пункт **5.5.2**) использовалась функция **fabs**. Если в **C**-программе употребить функцию **abs**, то результат окажется следующим:

```
#  t0=1.000000e+00  tn=1.500000e+00  n=5
#  i          t              r0              r1         aer         rer
   1   1.0000000e+00   5.7829318e+00   8.9975405e+00  3.000e+00  3.334e-01
   2   1.1000000e+00   6.6944509e+00   8.9979858e+00  2.000e+00  2.223e-01
   3   1.2000000e+00   6.4263763e+00   8.9983501e+00  2.000e+00  2.223e-01
   4   1.3000000e+00   7.2002325e+00   8.9986496e+00  1.000e+00  1.111e-01
   5   1.4000000e+00   3.8472526e+00   8.9988937e+00  5.000e+00  5.556e-01
```

Сравнивая его с **C++**-результатом, видим, что числа в колонке **aer**, хотя и отпечатаны в форме с плавающей запятой, тем не менее очень похожи на соответствующие целочисленные значения. Дело в том, что в языке СИ одно имя (в данном случае **abs** всегда соответствует алгоритмам расчета абсолютной величины для аргументов **целого** типа. Если аргумент нецелый, то он по С-умолчанию преобразуется к целому так, что результат работы функции **abs** – всегда целое число.

В языке C++ работает **механизм перегрузки функций**, который в зависимости от типа аргумента (и/или их количества) выбирает нужный алгоритм расчета. Если бы аргумент был целый, то выбралась бы целочисленная функция со специфическим именем **abs**. Если же аргумент вещественного типа, то по имени **abs** автоматически выбирается функция со специфическим именем **fabs**. Механизм нужной нам перегрузки обусловлен подключением заголовка **include** <**cmath**>. Если вместо последнего использовать СИ-шный заголовочный файл **include** <**math.h**>, то результат на C++ был бы таков:

```
tsfs2p30.cpp: In function ‘int main()’:
tsfs2p30.cpp:25: error: call of overloaded ‘abs(float)’ is ambiguous
/usr/include/stdlib.h:778: note: candidates are: int abs(int)
/usr/lib/gcc/i386-redhat-linux/3.4.4/../../../../include/c++/3.4.4/cstdlib:
                     153: note: long long int __gnu_cxx::abs(long long int)
/usr/lib/gcc/i386-redhat-linux/3.4.4/../../../../include/c++/3.4.4/cstdlib:
                     123: note:                 long int std::abs(long int)
```

Другими словами, в СИ механизм перегрузки по имени **abs** распространяется исключительно на данные используемых целых типов, а для аргумента типа **float** неопределен.

7.2 Чуть-чуть о файловом вводе-выводе в C++

1. Язык C++ нацелен на решение задач *объектно-ориентированного программирования* (**ООП**), хотя при желании можно оставаться в рамках традиционной императивной направленности C.

2. И в C, и в C++ файл открывается посредством связывания его с потоком.

3. **Поток** ввода/вывода – это абстрактное устройство, которое принимает и выдает пользовательскую информацию.

4. **Поток** связан с физическим устройством (файлом или дисплеем) посредством системы ввода/вывода.

5. *Объектно-ориентированный инструментарий* – основное средство C++ для создания файла и связывания его с потоком.

6. *Классовый* подход для реализации работы с файлами требует включения в программу заголовка **fstream**, т.е:

   ```
   #include <fstream>}
   ```

   В **fstream** определены, в частности, классы **ifstream**, **ofstream** и **fstream**. Так что для создания и открытия файла ввода, например, с именем **input** и файла вывода с именем **result** в нужных местах программы достаточно поместить операторы:

   ```
   ifstream finp("input");
   ofstream fres("result")};
   ```

   Здесь **finp** – объект-поток класса (или типа) **ifstream**, а **fres** – объект-поток класса (или типа) **ofstream**. В C **finp** и **fres** моделировались указателями, принимающими значение от функции **fopen**. Для проверки факта открытия файлов можно, как и в C, использовать условия:

   ```
   if (!finp) { cout<< "Неудача при открытии файла input. "; return 1;}
   if (!fres) { cout<< "Неудача при открытии файла result."; return 1;}
   ```

7. Ввод из потока **finp** и вывод в поток **fres** осуществляется через их имена аналогично тому, как это делалось с потоками **cin** и **cout** (вспомним пункты **1.6.8**, **1.6.10** и **1.6.11**), то есть:

   ```
   finp >> t0 >> tn >> n;
   fres << "  #  t0=" << t0 <<"  tn=" << tn << " n=" << n << endl;
   ```

8. После завершения работы с объектом-потоком его необходимо закрыть посредством вызова соответствующего метода, т.е.

   ```
   finp.close();
   fres.close();
   ```

Таким образом, используя объектно-ориентированные модели потоков ввода и вывода языка C++, нашу программу можно записать так:

```
#include <iostream>                                  //  Файл  tsfs2p30a.cpp
#include <fstream>
#include <cmath>
using namespace std;
float w0(float); float w1(float);
int main()
{ float t0, tn, t, ht, x, r0, r1, aer, rer;  int n, i;
  ifstream ninp("input");
  if (!ninp) { printf("Неудача при открытии файла input !!!\n");  return 1;}
  ofstream nres("result");
  if (!nres) { printf("Неудача при открытии файла result !!!\n"); return 2;}
   ninp >> t0 >> tn >> n;
   nres <<" #  t0="<< t0 <<"  tn=" << "  n="<< n << endl; ht=(tn-t0)/n;
   nres <<" # i\t" << "t\t" << "r0\t" << "r1\t" << "aer\t" << "rer"<< endl;
   for (i=1; i<=n;i++)
     { t=t0+ht*(i-1); x=exp(-t); r0=w0(x); r1=w1(x);
       aer=abs(r0-r1); rer=aer/r1;
       nres<< i<< t << r0<< r1<<aer<<rer<<endl;    }
   ninp.close(); nres.close(); return 0;             }


#include <cmath>                                     //    Файл w0.c
float w0(float x)
{ float x2, a, b;
  x2=x*x; a=cos(x) - 1 + x2* 0.5*(1 - x2/12 * ( 1 - x2 / 30));
          b=sin(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42)); return(a/b); }

#include <cmath>
float w1(float x)                                     //    Файл w1.c
{ float x2, sa1, sa, sb, a, b; int k, k2;
 x2=x*x; sa1=1.0; sa=0; sb=0; a=1/40320.0; b=a/9; k=k2=0;
 while (fabs(sa)!=sa1)
   { sa1=fabs(sa); sa=sa+a; a=-a*x2/(k2+ 9)/(k2+10);
                   sb=sb+b; b=-b*x2/(k2+10)/(k2+11); k=k+1; k2=2*k;
   } return (sa/sb); }

 #  t0=1  tn=  n=10                              // Файл result
 # i t r0 r1 aer rer
115.782938.997543.214610.357276
21.16.694458.997992.303530.256006
31.26.426388.998352.571970.285827
41.37.200238.998651.798420.199854
51.43.847258.998895.151640.572475
61.51.791718.999097.207390.800902
71.63.489718.999265.509550.612222
81.73.16628.999395.833190.648176
91.83.158758.99955.840750.649008
101.92.960758.999596.038850.671013
```

7.3 Чуть-чуть о форматировании вывода в C++.

Причина столь неудачного в предыдущем пункте вывода результата — использование форматов, заданных в C++ по умолчанию. Их можно изменить на основе объектно-ориентированного инструментария C++ [16].

1. Каждый поток ввода/вывода C++ связан с набором **флагов формата**, которые предназначены для управления способом форматирования.

2. Их значения определены в классе **ios**. Класс **ios** — прародитель для класса **fstream**, который использован в нашей программе при описании объектов **ninp** и **nres**, так что свойство **наследования** обеспечит для них все возможности форматирования, предоставленные **ios**. Рассмотрим пример из [16] (глава **8.3**):

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ double t; int  i;
  cout.setf(ios::scientific);  cout.precision(13);
  cout.width( 6); cout<< "i";  cout.width(13); cout<<"t";
  cout.width(25); cout<<"sqrt(t)"; cout.width(20); cout<<"t^2\n"<<" ";
  for (i=2; i<=10;i++)
    { t=(double)i; cout.width( 5); cout << i << " ";
                   cout.width(21); cout << t <<" ";
                   cout.width(21); cout << sqrt(t) <<" ";
                   cout.width(21); cout << t*t <<"\n ";  }  return 0; }
```

3. Здесь **cout.setf(ios::scientific)** – вызов функции **setf** (установки **флага формата**, который внутри класса **ios** обозначен именем **scientific** – запись вещественного значения в форме с плавающей запятой). Доступ к флагу **scientific**, должен осуществляться через класс **ios** посредством оператора **::** расширения области видимости нашей программы до конкретного флага класса **ios** с целью изменить на него флаг формата, действующий пока по умолчанию).

4. **cout.precision(13)** – вызов функции **precision** (одного из методов класса **ios**) для модификации объекта **cout**, которая установит количество цифр числа, выводимых после запятой). По умолчанию при выводе значения с плавающей точкой выводиться **шесть** цифр, а мы хотим **тринадцать**.

5. **cout.width(6)** – вызов функции **width** для модификации объекта **cout**, которая установит минимальную ширину поля выводимого данного. По умолчанию в C++ при выводе любого значения ширина поля устанавливается равной минимальному количеству символов, требуемых для его размещения.

6. В **gnu**-компиляторе стандарта C++ после выполнения каждой операции вывода значение ширины поля возвращается к своему состоянию по умолчанию. Поэтому в примере перед каждой инструкцией вывода приходится заново переустанавливать ширину поля.

7.4 1-й вариант модификации программы из пункта 7.2

С учетом сказанного в предыдущем пункте модифицируем главную программу из пункта **7.2**, например, следующим образом:

```
#include <iostream>  // Программа демонстрирует некоторые возможности C++
#include <fstream>   // по объектно-ориентированному форматированию данных
#include <cmath>     // при выводе результата в файл.
using namespace std;
float w0(float); float w1(float);
int main()
{ float t0, tn, t, ht, x, r0, r1, aer, rer;
  int n, i;
  ifstream ninp("input");
  if (!ninp) { printf("Неудача при открытии файла input  !!!\n"); return 1;}
  ofstream nres("result");
  if (!nres) { printf("Неудача при открытии файла result !!!\n"); return 2;}
   ninp >> t0 >> tn >> n;
   nres.setf(ios::scientific);
   nres <<" #  t0="<< t0 <<"  tn=" << tn << "  n="<< n << endl; ht=(tn-t0)/n;
   nres.width( 5); nres<<"i" ;  nres.width(10); nres<<"t";
   nres.width(14); nres<<"r0";  nres.width(14); nres<<"r1";
   nres.width(14); nres<<"aer"; nres.width(14); nres<<"rer\n";
   for (i=1; i<=n;i++)
     { t=t0+ht*(i-1); x=exp(-t); r0=w0(x); r1=w1(x);
       aer=abs(r0-r1); rer=aer/r1;
       nres.width( 5); nres <<  i;   nres.width(14); nres << t;
       nres.width(14); nres << r0;   nres.width(14); nres << r1;
       nres.width(14); nres << aer;  nres.width(14); nres << rer<<endl;
     }
   ninp.close(); nres.close(); return 0;
}
```

Тогда содержимое файла **result** будет выглядеть так:

```
 #  t0=1.000000e+00  tn=2.000000e+00  n=10
    i         t            r0            r1           aer          rer
    1  1.000000e+00  5.782932e+00  8.997540e+00  3.214609e+00  3.572764e-01
    2  1.100000e+00  6.694451e+00  8.997986e+00  2.303535e+00  2.560056e-01
    3  1.200000e+00  6.426376e+00  8.998350e+00  2.571974e+00  2.858272e-01
    4  1.300000e+00  7.200233e+00  8.998650e+00  1.798417e+00  1.998541e-01
    5  1.400000e+00  3.847253e+00  8.998894e+00  5.151641e+00  5.724749e-01
    6  1.500000e+00  1.791706e+00  8.999095e+00  7.207389e+00  8.009015e-01
    7  1.600000e+00  3.489712e+00  8.999260e+00  5.509548e+00  6.122224e-01
    8  1.700000e+00  3.166201e+00  8.999393e+00  5.833192e+00  6.481761e-01
    9  1.800000e+00  3.158754e+00  8.999504e+00  5.840750e+00  6.490080e-01
   10  1.900000e+00  2.960747e+00  8.999595e+00  6.038848e+00  6.710133e-01
```

Вообще говоря, лишний раз убеждаемся, что более удобного средства для вывода табулированных функций нежели оператор **format** ФОРТРАНа нет. То, что в C++ пишется чуть ли не десятком операторов, ФОРТРАН позволяет записать одной короткой строкой. Причина ясна – средства, предоставляемые C++, разрабатывались совсем для иных целей (короче, не надо стрелять из *космической пушки* по мухам, хотя при *лазерном наведении* попасть всегда можно).

7.5 2-й вариант модификации программы из пункта 7.2

Наряду с флагами формата C++ предоставляет еще один способ форматирования информации [16] (глава **8.4**): *манипуляторы ввода/вывода*. Отсылая за деталями к упомянутой книге, приведем пример использования **манипуляторов** для корректировки нашей программы из пункта **5.5.2**.

```
#include <iostream>  // Программа демонстрирует применение некоторых
#include <iomanip>   // манипуляторов C++ из класса для форматирования
#include <fstream>   // данных, выводимых в файл.
#include <cmath>
using namespace std;
float w0(float); float w1(float);
int main()
{ float t0, tn, t, ht, x, r0, r1, aer, rer;
  int n, i;
  ifstream ninp("input");
  if (!ninp) { printf("Неудача при открытии файла input  !!!\n"); return 1;}
  ofstream nres("result");
  if (!nres) { printf("Неудача при открытии файла result !!!\n"); return 2;}
  ninp >> t0 >> tn >> n;
  nres << scientific;
  nres <<" #  t0="<< t0 <<"  tn=" << tn << "  n="<< n << endl; ht=(tn-t0)/n;
  nres << setw( 5) << "i" ; nres << setw(10)<<"t";
  nres << setw(14) <<"r0"; nres << setw(14)<< "r1";
  nres << setw(14) <<"aer"; nres << setw(14)<< "aer"<< endl;
 for (i=1; i<=n;i++)
   { t=t0+ht*(i-1); x=exp(-t); r0=w0(x); r1=w1(x); aer=abs(r0-r1); rer=aer/r1;
     nres << setw( 5)  << i; nres << setw(14)<< t;
     nres << setw(14)<<  r0; nres << setw(14)<< r1;
     nres << setw(14)<< aer; nres << setw(14)<< rer << endl;
    }
   ninp.close(); nres.close(); return 0;
}
```

Здесь вместо метода **width()** и флага формата **scientific** класса **ios** используются соответствующие манипуляторы вывода **setw(int)** и **scientific**.
Результат работы программы, очевидно, тот же:

```
 #  t0=1.000000e+00  tn=2.000000e+00  n=10
    i         t            r0            r1           aer           aer
    1  1.000000e+00  5.782932e+00  8.997540e+00  3.214609e+00  3.572764e-01
    2  1.100000e+00  6.694451e+00  8.997986e+00  2.303535e+00  2.560056e-01
    3  1.200000e+00  6.426376e+00  8.998350e+00  2.571974e+00  2.858272e-01
    4  1.300000e+00  7.200233e+00  8.998650e+00  1.798417e+00  1.998541e-01
    5  1.400000e+00  3.847253e+00  8.998894e+00  5.151641e+00  5.724749e-01
    6  1.500000e+00  1.791706e+00  8.999095e+00  7.207389e+00  8.009015e-01
    7  1.600000e+00  3.489712e+00  8.999260e+00  5.509548e+00  6.122224e-01
    8  1.700000e+00  3.166201e+00  8.999393e+00  5.833192e+00  6.481761e-01
    9  1.800000e+00  3.158754e+00  8.999504e+00  5.840750e+00  6.490080e-01
   10  1.900000e+00  2.960747e+00  8.999595e+00  6.038848e+00  6.710133e-01
```

7.6 Удвоенная точность.

Приведем пример использования манипуляторов для форматирования вывода результатов, полученных в режиме удвоенной точности:

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cmath>
using namespace std;
double w0(double); double w1(double);
int main()
{ double t0, tn, t, ht, x, r0, r1, aer, rer;
  int n, i;
  ifstream ninp("input");
  if (!ninp) { printf("Неудача при открытии файла input  !!!\n"); return 1;}
  ofstream nres("result");
  if (!nres) { printf("Неудача при открытии файла result !!!\n"); return 2;}
   ninp >> t0 >> tn >> n;
   nres << scientific;
   nres <<" #  t0="<< t0 <<"  tn=" << tn << "  n="<< n << endl; ht=(tn-t0)/n;
   nres << setw( 5) << "#  i" ; nres << setw(13)<<"t";
   nres<<setw(18)<<"r0"<<setw(18)<<"r1"<< setw(11)<<"aer"<<setw(11)<<"rer\n";
  for (i=1; i<=n;i++)
     { t=t0+ht*(i-1);
       x=exp(-t);
       r0=w0(x);
       r1=w1(x);
       aer=abs(r0-r1); rer=aer/r1;
       nres << setw( 5)<< i;
       nres<<setprecision(11)<<setw(18)<<t<<setw(18)<<r0<<setw(18)<<r1
           <<setprecision( 3)<<setw(10)<<aer<<setw(10)<<rer<<endl;
     }
   ninp.close();
   nres.close();
   return 0;
}
```

Результат совпадает с соответствующим ФОРТРАН-результатом, приведенным в пункте **5.5.2**.

```
 #  t0=1.000000e+00  tn=2.000000e+00  n=10
 #  i            t                 r0                 r1        aer       rer
    1 1.00000000000e+00 8.99754059915e+00 8.99754060052e+00 1.362e-09 1.514e-10
    2 1.10000000000e+00 8.99798622355e+00 8.99798622967e+00 6.125e-09 6.807e-10
    3 1.20000000000e+00 8.99835115228e+00 8.99835114074e+00 1.154e-08 1.282e-09
    4 1.30000000000e+00 8.99864992248e+00 8.99864994539e+00 2.291e-08 2.546e-09
    5 1.40000000000e+00 8.99889457761e+00 8.99889461326e+00 3.565e-08 3.961e-09
    6 1.50000000000e+00 8.99909497496e+00 8.99909494866e+00 2.629e-08 2.922e-09
    7 1.60000000000e+00 8.99925903871e+00 8.99925898169e+00 5.702e-08 6.336e-09
    8 1.70000000000e+00 8.99939264655e+00 8.99939328880e+00 6.422e-07 7.137e-08
    9 1.80000000000e+00 8.99950319812e+00 8.99950325567e+00 5.755e-08 6.395e-09
   10 1.90000000000e+00 8.99959372814e+00 8.99959329263e+00 4.355e-07 4.839e-08
```

7.7 О перегрузке функций в СИ++.

Перегрузка функций в C++ описывается проще чем в ФОРТРАНе-95: нет нужды в каких-то особых служебных словах вроде **interface** и **end interface**.

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cmath>
using namespace std;
float  w0( float); float  w1(float);
double w0(double); double w1(double);
int main()
{ double t0, tn, t, ht, x, r08, r18;
  float  z, r04, r14;
  int n, i;
  ifstream ninp("input");
  if (!ninp) { printf("Неудача при открытии файла input  !!!\n"); return 1;}
  ofstream nres("result");
  if (!nres) { printf("Неудача при открытии файла result !!!\n"); return 2;}
   ninp >> t0 >> tn >> n;
   nres << setprecision(6);
   nres << scientific;
   nres <<" #  t0="<< t0 <<"  tn=" << tn << "  n="<< n << endl; ht=(tn-t0)/n;
   nres << setw( 5) << "#  i" ; nres << setw(8)<<"t";
   nres << setw(15) <<"r08"; nres << setw(14)<< "r18";
   nres << setw(14) <<"r04"; nres << setw(15)<< "r14\n";
  for (i=1; i<=n;i++)
     { t=t0+ht*(i-1);        x= exp(-t);   r08=w0(x); r18=w1(x);
                             z=(float)x;   r04=w0(z); r14=w1(z);
       nres << setw( 5)<< i;
       nres << setw(14)<< t;
       nres << setw(14)<< r08;
       nres << setw(14)<< r18;
       nres << setw(14)<< r04;
       nres << setw(14)<< r14 <<endl;
     }
   ninp.close(); nres.close(); return 0;
}
```

Результат, очевидно, тот же:

```
 #  t0=1.000000e+00  tn=2.000000e+00  n=10
 #  i       t            r08           r18           r04           r14
    1  1.000000e+00  8.997541e+00  8.997541e+00  5.782932e+00  8.997540e+00
    2  1.100000e+00  8.997986e+00  8.997986e+00  6.694451e+00  8.997986e+00
    3  1.200000e+00  8.998351e+00  8.998351e+00 -3.313327e+00  8.998350e+00
    4  1.300000e+00  8.998650e+00  8.998650e+00  7.200233e+00  8.998650e+00
    5  1.400000e+00  8.998895e+00  8.998895e+00  3.905801e+00  8.998894e+00
    6  1.500000e+00  8.999095e+00  8.999095e+00  1.791706e+00  8.999095e+00
    7  1.600000e+00  8.999259e+00  8.999259e+00  3.489712e+00  8.999260e+00
    8  1.700000e+00  8.999393e+00  8.999393e+00  2.870293e+00  8.999393e+00
    9  1.800000e+00  8.999503e+00  8.999503e+00  3.158754e+00  8.999504e+00
   10  1.900000e+00  8.999594e+00  8.999593e+00  2.960747e+00  8.999595e+00
```

7.8 О чем узнали из главы 7? (кратко)

1. Для организации ввода-вывода **C++** (помимо средств языка **C**) предоставляет и объектно-ориентированную систему, основанную на иерархии классов.
2. В **C++** при запуске программы автоматически открывается несколько стандартных потоков, в частности, поток ввода **cin** и поток вывода **cout**. Первый связан с клавиатурой, остальные с экраном.
3. Каждый поток описывается объектом определенного класса, для которого в системе определены свои функции-члены (методы обработки) и свойства .
4. При подключении к программе посредством #**include** <**iostream**> класса **iostream** получаем возможность пользоваться потоками **cin** и **cout**.
5. Подключая к программе посредством #**include** <**fstream**> класс **fstream**, можем создавать новые объекты (потоки ввода и вывода).
6. Поскольку класс **fstream** является наследником класса **iostream**, а последний наследником класса **ios**, то можем использовать соответствующие методы и свойства родительских классов для форматирования созданных потоков.
7. Есть два способа форматирования: посредством **методов** объектов класса **ios** и посредством **манипуляторов**.
8. **scientific** – **флаг** класса **ios** для задания формы вывода данного с порядком.
9. При установке флага **scientific** имя **scientific** необходимо предварить явным указанием имени класса **ios**.
10. **width** – метод объекта для модификации ширины поля вывода данного.
11. **precision** – метод объекта для модификации количества знаков после запятой.
12. В **gnu**-компиляторе **C++** значение ширины поля после выполнения операции вывода возвращается к своему состоянию по умолчанию. Поэтому перед каждой инструкцией вывода ширину поля приходится переустанавливать.
13. **Манипулятор** – специальная функция для управления состоянием потока при выполнении операции ввода или вывода. Манипуляторы определены в пространстве имен **std**. Применение манипуляторов для форматирования менее громоздко по записи нежели явная установка флагов.
14. Инструкция #**include** <**iomanip**> подключает к программе манипуляторы с параметрами.
15. **scientific** – манипулятор, устанавливающий флаг **scientific** записи числа с плавающей запятой (при вызове манипулятора без параметра скобки не ставятся).
16. **setw** – манипулятор установки ширины поля.
17. **setprecision** – манипулятор установки количества цифр после запятой.

7.9 Девятое домашнее задание

(третья часть контрольной) Дать решение задачи из контрольной N 1 на **C++** в трех вариантах:

1. используя для форматирования вывода **флаги формата**;
2. используя для форматирования вывода **манипуляторы**;
3. используя механизм перегрузки функций обеспечить в пределах одной главной программы вызов по одному имени двух алгоритмов расчета каждой из вычисляемых функций (с одинарной и удвоенной точностью).

Решение каждой задачи разместить в отдельной директории со своим **make**-файлом и **gnuplot**-скриптом.

Письменно сформулировать:

"Какие ошибки обнаруживались в процессе отладки и пропуска задач?

8 Приложение I: Лабораторная работа N 1.

8.1 Расчет числа π.

Некто с целью проверки значения константы $\pi = \mathbf{3.14159265358979}$, приводимое в справочниках, составил программу расчета набора последовательных приближений длины полуокружности радиуса , приближая последнюю длиной полупериметра правильного вписанного $\mathbf{n}$-угольника, где $\mathbf{n = 6 \cdot 2^k}$ при $\mathbf{k = 0, 1, 2 \cdots, 14}$. Расчет полупериметра велся по формуле

$$\mathbf{p_k = n * b_k} \ ,$$

где для вычисления $\mathbf{b_k}$ (половины длины стороны очередного $\mathbf{n}$-угольника) использовалось ее выражение через $\mathbf{b_{k-1}}$ (половину длины стороны предыдущего):

$$\mathbf{b_k = \frac{\sqrt{r \cdot (r - \sqrt{r^2 - b_{k-1}^2})}}{2}}.$$

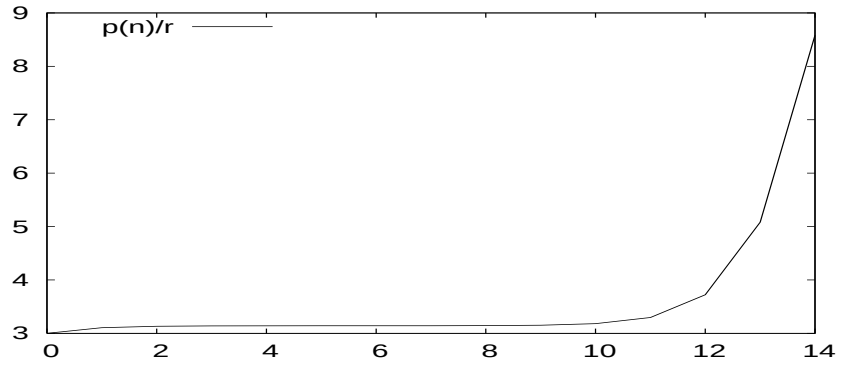
Тогда соответствующее приближенное значение π дается формулой $\pi \approx \mathbf{\frac{p_k}{r}}$
Была написана следующая программа:

```
      program tsfs2p01
      data ninp / 5 /
      data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      pi=4.0*atan(1.0)
      write(nres,*) ' #  pi=',pi
      read(ninp,101) r
      write(nres,*) ' #   r=',r
      write(nres,1000)
                    dn=6; b=r*0.5; p=3*r
      do i=0,14
        write(nres,1001) i,dn, b, p/r
        b=half(r,b)
        dn=dn*2
        p=dn*b
      enddo
      close(nres)
  101 format(e10.3)                                          ! Форматы ввода
 1000 format(1x,' #',2x,'i',12x,'n',10x,'b(n)',10x,'p(n)/r') ! и
 1001 format(1x,i5,2x,e15.7,e15.7,e15.7)                     !   вывода.
      end
```

которая в результате пропуска при $\mathbf{r = 3.4}$ дала такой результат:

```
#  pi=   3.141593
#   r=   3.400000
#  i            n           b(n)          p(n)/r
   0    0.6000000E+01  0.1700000E+01  0.3000000E+01
   1    0.1200000E+02  0.8799848E+00  0.3105829E+01
   2    0.2400000E+02  0.4437892E+00  0.3132629E+01
   3    0.4800000E+02  0.2223708E+00  0.3139353E+01
   4    0.9600000E+02  0.1112452E+00  0.3141042E+01
   5    0.1920000E+03  0.5563058E-01  0.3141491E+01
   6    0.3840000E+03  0.2781725E-01  0.3141712E+01
   7    0.7680000E+03  0.1391080E-01  0.3142204E+01
   8    0.1536000E+04  0.6959525E-02  0.3144068E+01
   9    0.3072000E+04  0.3487977E-02  0.3151490E+01
  10    0.6144000E+04  0.1760317E-02  0.3180997E+01
  11    0.1228800E+05  0.9120854E-03  0.3296384E+01
  12    0.2457600E+05  0.5149713E-03  0.3722333E+01
  13    0.4915200E+05  0.3514531E-03  0.5080771E+01
  14    0.9830400E+05  0.2968169E-03  0.8581850E+01
```

График зависимости отношения $\frac{\mathbf{p(n)}}{\mathbf{r}}$ от количества удвоений числа сторон:



1. Письменно сформулировать свое отношение к полученному результату.
2. Объяснить объективную и субъективную причины его появления.
3. Преобразовать расчетную формулу так, чтобы результат был верен.
4. Написать соответствующую новой схеме расчета функцию **half1(r,b)**.
5. Обеспечить наглядный вывод результатов обеих расчетных формул в одну таблицу.
6. Включить **make**-файл псевдоцель вывода на один рисунок графиков, демонстрирующих приближение результатов к числу π с ростом **k**.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.2 Табулирование функции $w(x) = \dfrac{1 + e^{-x} - 2e^{-x/2}}{x^2}$

Для двадцати равномерно распределённых по промежутку $[\mathbf{5 \cdot 10^{-4}}, \mathbf{1.5 \cdot 10^{-3}}]$ значений аргумента x вычислить таблицу значений функции:

$$\mathbf{w(x) = \frac{1 + e^{-x} - 2e^{-x/2}}{x^2}}$$

и построить ее график. Некто предложил для расчета программу:

```
      program tsfs2p02
      data ninp / 5 /
      data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      read(ninp,100) x0,xk
      read(ninp,101) n
      write(nres,1000) x0, xk, n
      write(nres, 1100)
      h=(xk-x0)/n
      do i=0,n
        x=x0+i*h
        r0=w0(x)
        write(nres,1101) i, x, r0
      enddo
      close(nres)
  100 format(e10.3)
  101 format(i10)
 1000 format(1x,'# x0=',e15.7,5x,'xk='e15.7,5x,'n=',i3)
 1100 format(1x,'# N ',7x,'x',12x,'w0(x)')
 1101 format(1x,i3,e15.7,e15.7)
      end
```

в которой табулирование велось непосредственно по приведенной выше формуле через вызов функции **w0**:

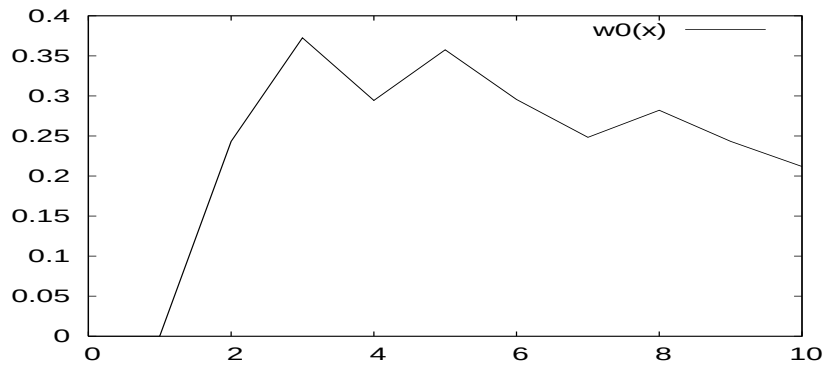
```
      function w0(x)
      w0= (1 + exp(-x) - 2 * exp(-x/2)) / x / x
      return
      end
```

Тестирование функции на промежутке **[1.0,2.0]** удовлетворило заказчика.

В результате пропуска программы по требуемому промежутку $[\mathbf{5 \cdot 10^{-4}}, \mathbf{1.5 \cdot 10^{-3}}]$ было получено:

```
# x0=  0.5000000E-03     xk=  0.1500000E-02     n= 10
# N        x            w0(x)
  0  0.5000000E-03  0.0000000E+00
  1  0.6000000E-03  0.0000000E+00
  2  0.7000000E-03  0.2432842E+00
  3  0.8000000E-03  0.3725290E+00
  4  0.9000000E-03  0.2943439E+00
  5  0.1000000E-02  0.3576278E+00
  6  0.1100000E-02  0.2955602E+00
  7  0.1200000E-02  0.2483527E+00
  8  0.1300000E-02  0.2821522E+00
  9  0.1400000E-02  0.2432843E+00
 10  0.1500000E-02  0.2119276E+00
```

График функции, вычисленной по алгоритму **w0(x)**:



1. Письменно сформулировать свое отношение к полученному результату (для упрощения работы используем возможности системы **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Преобразовать расчетную формулу так, чтобы результат был верен.
4. Написать соответствующую новой схеме расчета функцию **w1(x)**.
5. Обеспечить наглядный вывод результатов **w0(x)** и **w1(x)** в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок графиков **w0(x)** и **w1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.3 Отношение объемов.

Пользователь написал функцию **vsph(r)** расчета объема шара радиуса **r**:

```
      function vsph(r)                              !   Файл  vsph.for
      parameter (pi=3.14159265, c43=4.0/3.0)
      vsph=c43*pi*r*r*r
      end
```

и функцию **vcyl(r,h)** расчета объема кругового цилиндра высоты **h** с радиусом основания **r**:

```
      function vcyl(r,h)                            !   Файл vcyl.for
      parameter (pi=3.14159265)
      vcyl=pi*r*r*h
      end
```

и применил их для поиска отношения объема шарового слоя, заключенного между сферами с радиусами **r** и **r+r*q** (q<1 – доля радиуса, посредством которой характеризуется толщина слоя), к объему цилиндра с площадью основания равной площади поверхности сферы радиуса **r** и высотой **h=r*q**, используя программу:

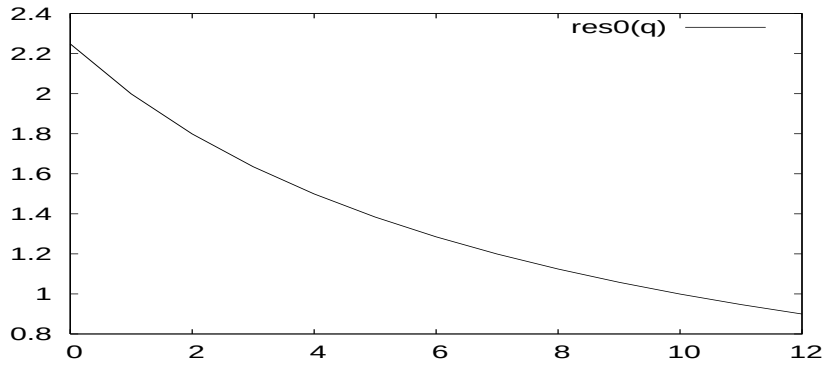
```
      program tsfs2p03
      data ninp / 5 /
      data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      pi=4.0*atan(1.0)
      write(nres,*) ' #  pi=',pi
      read(ninp,101) r, q0, q1
      write(nres,*) ' #   r=',r,'   q0=',q0, '   q1=',q1
      h=(q1-q0)/12
      write(nres,*) ' #  h=',h
      write(nres,1000)
      do i=0,12
        q=q0+i*h
        rq=r*q
        res0=(vsph(r+rq)-vsph(r))/vcyl(2*r, rq)
        write(nres,1001) i,q, res0
      enddo
      close(nres)
  101 format(e10.3)
 1000 format(1x,' #',2x,'i',12x,'q',10x,'res0')
 1001 format(1x,i5,2x,e15.7,e15.7,e15.7,e15.7)
      end
```

которая после ввода радиуса сферы **r** и значений граничных точек **q0** и **q1** промежутка задания параметра **q** вычисляла требуемое отношение для двенадцати равноотстоящих по **[q0,q1]** значений **q**. Точность работы программы при **r=3.000** и $\mathbf{q} \in [\mathbf{4e-2}, \mathbf{1e-1}]$ заказчика устроила.

Расчет при **r=3.0** и $\mathbf{q} \in [\mathbf{4e-8}, \mathbf{0.1e-7}]$ дал следующий результат

```
#  pi=   3.141593
#   r=   3.000000        q0=  4.0000000E-08    q1=  1.0000000E-07
#  h=  5.0000000E-09
#  i            q          res0
   0    0.4000000E-07  0.2248622E+01
   1    0.4500000E-07  0.1998775E+01
   2    0.5000000E-07  0.1798898E+01
   3    0.5500000E-07  0.1635361E+01
   4    0.6000000E-07  0.1499081E+01
   5    0.6500000E-07  0.1383767E+01
   6    0.7000000E-07  0.1284927E+01
   7    0.7500000E-07  0.1199265E+01
   8    0.8000000E-07  0.1124311E+01
   9    0.8500000E-07  0.1058175E+01
  10    0.9000000E-07  0.9993875E+00
  11    0.9500000E-07  0.9467882E+00
  12    0.1000000E-06  0.8994488E+00
```

График искомого отношения при **r=3** и $\mathbf{q} \in [\mathbf{4e-8}, \mathbf{1e-7}]$



1. Письменно сформулировать свое отношение к полученному результату (для его оценивания используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Преобразовать расчетную формулу так, чтобы результат был верен.
4. Написать соответствующую новой схеме расчета функцию **ratio1(q)**.
5. Обеспечить наглядный вывод результатов всех расчетов в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок графиков по обоим способам расчета.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.4 Косинус угла сферического треугольника.

Косинус угла сферического равностороннего треугольника со стороной, измеряемой в радианной мере дугой φ, выражается через нее формулой:

$$\cos(\mathbf{A}) = \frac{\cos(\varphi) - \cos^{\mathbf{2}}(\varphi)}{\sin^{\mathbf{2}}(\varphi)}$$

Некто решил проверить, что по мере уменьшения длины стороны φ величина $\cos(\mathbf{A})$ будет приближаться к **0,5**. Программа

```
      program tsfs2p04
      data ninp / 5 /, nres / 6 /
      real xx (100)
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      pi=4*atan(1.0)
      cpi3=cos(pi/3)
      read(ninp,100) n
      read(ninp,101) (xx(i),i=1,n)
      write(nres,1000) n
      write(nres, 1100)
      do i=1,n
        x=xx(i)
        resm0=cosa0(x)
        aer0=abs(resm0-cpi3)
        rer0=aer0/cpi3
        write(nres,1101) i, x, resm0, aer0, rer0
      enddo
      close(nres)
  100 format(i10)
  101 format(e10.3)
 1000 format(1x,'#  n=',i3)
 1100 format(1x,'#N ',5x,'x',13x,'cosa0',9x,'aer',7x,'rer')
 1101 format(i3,e14.7,e16.7,e10.2,e10.2)
      end
```

для **n** вводимых значений аргумента $\varphi \in [\mathbf{0.5}, \mathbf{1e-9}]$ вычислила по приведенной формуле (посредством функции **cosa0**)

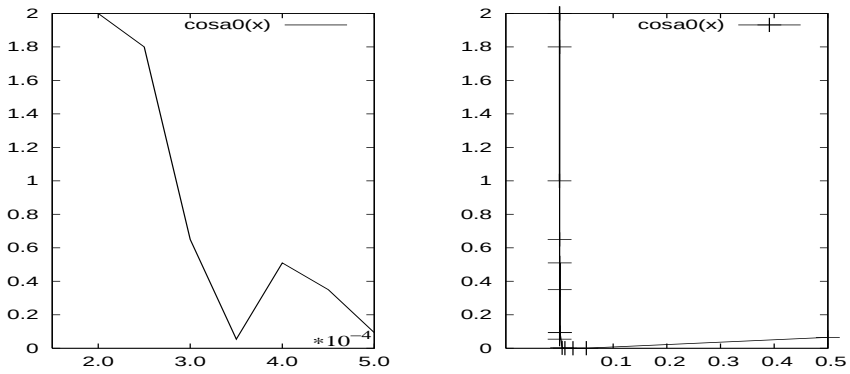
```
      function cosa0(x)
      c2=cos(x)**2
      s2=sin(x)**2
      cosa0= (cos(x)-c2)/s2
      end
```

таблицу значений искомого косинуса и их абсолютные и относительные погрешности относительно правильного значения.

В результате требуемого пропуска программа дала такой результат:

```
#  n= 15
#N      x              cosa0          aer       rer
 1 0.5000000E+00   0.4674003E+00  0.33E-01  0.65E-01
 2 0.5000000E-01   0.4996850E+00  0.31E-03  0.63E-03
 3 0.2500000E-01   0.4999623E+00  0.38E-04  0.75E-04
 4 0.1000000E-01   0.5001868E+00  0.19E-03  0.37E-03
 5 0.5000000E-02   0.5013633E+00  0.14E-02  0.27E-02
 6 0.1000000E-02   0.4536744E+00  0.46E-01  0.93E-01
 7 0.5000000E-03   0.4536743E+00  0.46E-01  0.93E-01
 8 0.4500000E-03   0.6773758E+00  0.18E+00  0.35E+00
 9 0.4000000E-03   0.2450581E+00  0.25E+00  0.51E+00
10 0.3500000E-03   0.4731372E+00  0.27E-01  0.54E-01
11 0.3000000E-03   0.8245475E+00  0.32E+00  0.65E+00
12 0.2500000E-03   0.1407348E+01  0.91E+00  0.18E+01
13 0.2000000E-03  -0.5000000E+00  0.10E+01  0.20E+01
14 0.1000000E-08  -0.4878910E+00  0.99E+00  0.20E+01
15 0.1000000E-09   0.0000000E+00  0.50E+00  0.10E+01
```

Графики относительной погрешности работы **cosa0(x)**:



1. Письменно сформулировать свое отношение к полученному результату (для упрощения его оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести расчетную формулу к виду удобному для расчёта (два–три способа).
4. Написать функции **cosa1(x)** и **cosa2(x)**, соответствующие новым формулам.
5. Обеспечить наглядный вывод относительных погрешностей расчета по всем формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок графиков относительных погрешностей каждого из способов расчета.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.5 Расчет отношения $r(x) = \dfrac{1.1 - \sqrt{1.21 - x^2}}{x \cdot (2.2 - \sqrt{4.84 - x})}$ при $\mathbf{x < 1}$

В некоторой задаче потребовался расчет отношения $\mathbf{r}(\mathbf{x})$. Были составлены две процедуры $\mathbf{r0}(\mathbf{x})$ и $\mathbf{r1}(\mathbf{x})$:

```
      function r0(x)
      r0= (1.1-sqrt(1.21-x*x))/(2.2-sqrt(4.84-x))/x
      end
      function r1(x)
      sn=sqrt(1-x*x/1.21)
      sd=sqrt(1-x/4.84)
      r1= 0.5*(1-sn)/(1-sd)/x
      end
```

Первая вела расчет непосредственно по формуле из заголовка. Во второй для уменьшения влияния погрешностей округления из под знака квадратного корня были вынесены константы **1,21** и **4,84** (соответственно в числителе и знаменателе) так, что оказалось возможным ранее приближенные значения **1,1**, **2,2** заменить на точную единицу. Тестирование обеих процедур проводилось программой

```
      program tsfs2p05
      data ninp / 5 /
      data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result', status='replace')
      read(ninp,100) x0,xk
      read(ninp,101) n
      write(nres,1000) x0, xk, n
      write(nres, 1100)
      h=(xk-x0)/n
      do i=0,n
        x=x0+i*h
         res0=r0(x)
         res1=r1(x)
        write(nres,1101) i, x, res0, res1
      enddo
      close(nres)
  100 format(e10.3)
  101 format(i10)
 1000 format(1x,'# x0=',e15.7,5x,'xk='e15.7,5x,'n=',i3)
 1100 format(1x,'# N ',7x,'x',12x,'r0(x)',12x,'r1')
 1101 format(1x,i3,e15.7,e15.7,e15.7)
      end
```

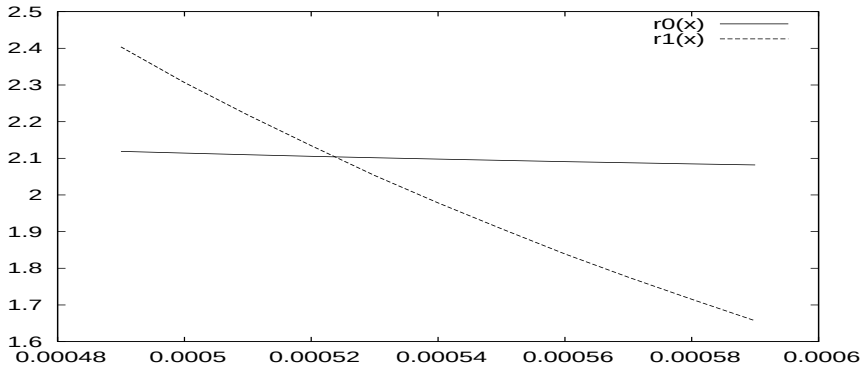
при $\mathbf{x} \in [\mathbf{0.49}, \mathbf{0.59}]$ и дало практически совпадающие результаты.

Однако, работа функций на требуемом рабочем диапазоне $[\mathbf{0.00049}, \mathbf{0.00059}]$ привела к следующему результату:

```
# x0=  0.4900000E-03     xk=  0.5900000E-03     n= 10
# N        x            r0(x)            r1
  0  0.4900000E-03  0.2118859E+01  0.2403788E+01
  1  0.5000000E-03  0.2114145E+01  0.2306805E+01
  2  0.5100000E-03  0.2109705E+01  0.2218082E+01
  3  0.5200000E-03  0.2105519E+01  0.2134381E+01
  4  0.5300000E-03  0.2101568E+01  0.2053093E+01
  5  0.5400000E-03  0.2097834E+01  0.1978474E+01
  6  0.5500000E-03  0.2094302E+01  0.1907851E+01
  7  0.5600000E-03  0.2090958E+01  0.1839047E+01
  8  0.5700000E-03  0.2087788E+01  0.1775694E+01
  9  0.5800000E-03  0.2084780E+01  0.1715560E+01
 10  0.5900000E-03  0.2081924E+01  0.1656809E+01
```

Как видно, совпадающих результатов нет. **Какой же из функций верить?**

Графики **r(x)**, полученные **r0(x)** и **r1(x)**:



1. Письменно сформулировать свое отношение к полученному результату (для упрощения его оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести расчетную формулу к виду удобному для расчёта (два способа).
4. Написать соответствующие новым схемам расчета функции **r2(x)** и **r3(x)**.
5. Обеспечить наглядный вывод результатов расчета по всем формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок всех графиков, соответствующих полученной таблице.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.6 Табулирование отношения $\dfrac{\tan x - x}{x - \sin x}$ при $x \ll 1$.

Нужна таблица значений при $x = exp(-t)$ и $t = 20.4(0.05)21.4$. Были составлены две подпрограммы-функции **d0**(**x**) и **d1**(**x**):

```
c                                                    файл d0.for
      function d0(x)
      d0= (tan(x)-x)/(x-sin(x))
      end

c                                                    файл d1.for
      function d1(x)
      d1= (tan(x)/x-1)/(1-sin(x)/x)
      end
```

Первая вела расчет непосредственно по формуле из заголовка. Вторая по чуть измененной, но аналитически тождественной. Именно, для уменьшения влияния погрешностей округления и в числителе, и в знаменателе было вынесен за скобки аргумент **x** так, что оказалось возможным одно из слагаемых заменить на точную единицу. Тестирование обеих функций проводилось программой

```
      program tsfs2p06
      data ninp / 5 /
      data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      read(ninp,100) t0,tk
      read(ninp,101) n
      write(nres,1000) t0, tk, n
      write(nres, 1100)
      h=(tk-t0)/n
      do i=1,n
         t=t0+h*(i-1)
         x=exp(-t)
         r0=d0(x)
         r1=d1(x)
         write(nres,1101) i, t, r0, r1
      enddo
      close(nres)
  100 format(e10.3)
  101 format(i10)
 1000 format(1x,'# t0=',e15.7,5x,'tk='e15.7,5x,'n=',i3)
 1100 format(1x,'# N ',7x,'t',9x,'d0(exp(-t))',4x,'d1(exp(-t))')
 1101 format(1x,i3,e15.7,e15.7,e15.7)
      end
```

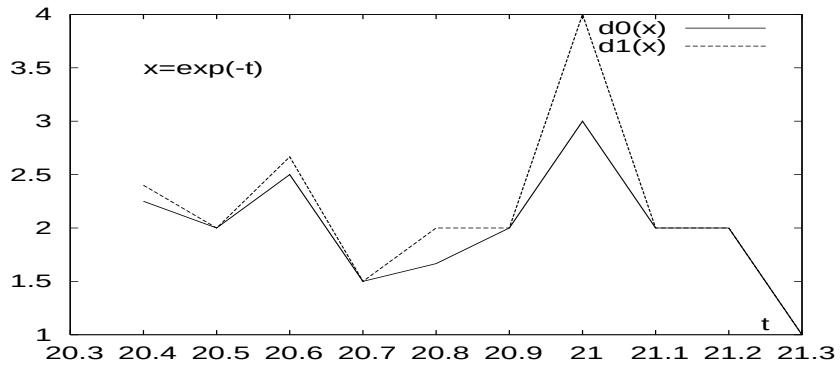
и на значениях аргумента $\mathbf{t} \in [\mathbf{1.0}, \mathbf{2.0}]$ дало практически совпадающие результаты.

Однако, тестирование функции на требуемом рабочем диапазоне $[\mathbf{20}.\mathbf{4}, \mathbf{21}.\mathbf{4}]$ привело к следующему:

```
# t0=  0.2040000E+02     tk=  0.2140000E+02     n= 10
# N        t          d0(exp(-t))    d1(exp(-t))
  1  0.2040000E+02  0.2250000E+01  0.2400000E+01
  2  0.2050000E+02  0.2000000E+01  0.2000000E+01
  3  0.2060000E+02  0.2500000E+01  0.2666667E+01
  4  0.2070000E+02  0.1500000E+01  0.1500000E+01
  5  0.2080000E+02  0.1666667E+01  0.2000000E+01
  6  0.2090000E+02  0.2000000E+01  0.2000000E+01
  7  0.2100000E+02  0.3000000E+01  0.4000000E+01
  8  0.2110000E+02  0.2000000E+01  0.2000000E+01
  9  0.2120000E+02  0.2000000E+01  0.2000000E+01
 10  0.2130000E+02  0.1000000E+01  0.1000000E+01
```

Некоторая странность результатов налицо. **Какой же из функций верить?**

Графики искомого отношения, полученные функциями d0(x) и d1(x)



1. Письменно сформулировать свое отношение к полученному результату (для упрощения его оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта (два–три способа).
4. Написать соответствующие новым схемам расчета функции **d2(x)** и **d3(x)**.
5. Обеспечить наглядный вывод результатов расчета по всем четырем формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок всех четырех графиков, соответствующих полученной таблице.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.7 Табулирование функции $(1-x)tg\frac{\pi x}{2}$ при $x \to 1$.

Нужна таблица значений при $x = 1 - k \cdot 10^{-6}$ и $k = 1(1)10$. Была составлена подпрограмма-функция $\mathbf{q0}(\mathbf{x})$:

```
c                                                    файл q0.for
      function q0(x)
      pi2=2*atan(1e0)
      q0= (1-x)*(tan(pi2*x))
      end
```

которая вела расчет непосредственно по формуле из заголовка. Тестирование функции проводилось программой

```
      program tsfs2p07
      data ninp / 5 /
      data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      write(nres,'('' # 2/pi='',e15.7)') 2/(4*atan(1e0))
      read(ninp,100) t0,tk
      read(ninp,101) n
      write(nres,1000) t0, tk, n
      write(nres, 1100)
      h=(tk-t0)/n
      do i=0,n
         t=(t0+h*i)*1e-6
         x=1-t
         r0=q0(x)
         write(nres,1101) i, t, x, r0
      enddo
      close(nres)
  100 format(e10.3)
  101 format(i10)
 1000 format(1x,'# t0=',e15.7,5x,'tk='e15.7,5x,'n=',i3)
 1100 format(1x,'# N ',7x,'t',14x,'x',13x,'r0')
 1101 format(1x,i3,e15.7,e15.7,e15.7)
      end
```

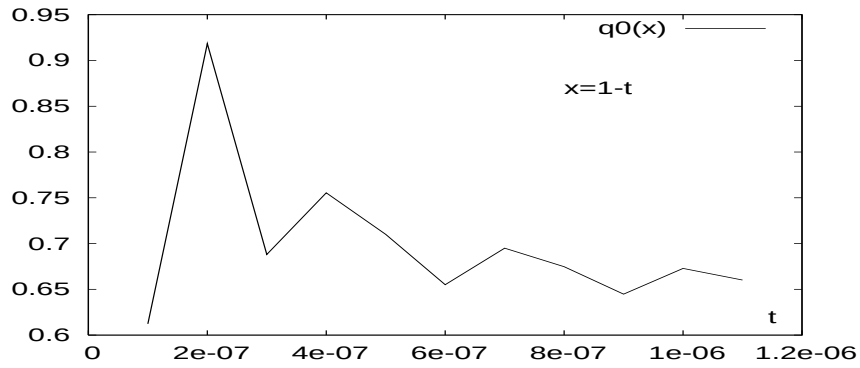
При $\mathbf{t} = \mathbf{0,1(0,05)0.6}$, что соответствует $\mathbf{x} = \mathbf{0,9(-0,05)0.4}$) тестирование дало результаты, устраивающие заказчика. В частности, при $\mathbf{t} = \mathbf{0,5}$ аргумент $\mathbf{x}$ тоже равен половине, как и $(\mathbf{1-x})$, а $\mathbf{tg\frac{\pi}{4} = 1}$. Так что семь цифр мантиссы результата на одинарной точности верны.

Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{10^{-6}}, \mathbf{10^{-5}}]$ привело к следующему:

```
# 2/pi=  0.6366197E+00
# t0=  0.1000000E+00     tk=  0.1100000E+01     n= 10
# N        t              x             r0
  0  0.1000000E-06  0.9999999E+00  0.6122490E+00
  1  0.2000000E-06  0.9999998E+00  0.9183736E+00
  2  0.3000000E-06  0.9999997E+00  0.6880755E+00
  3  0.4000000E-06  0.9999996E+00  0.7553975E+00
  4  0.5000000E-06  0.9999995E+00  0.7100605E+00
  5  0.6000000E-06  0.9999994E+00  0.6550228E+00
  6  0.7000000E-06  0.9999993E+00  0.6949816E+00
  7  0.8000000E-06  0.9999992E+00  0.6747413E+00
  8  0.9000000E-06  0.9999991E+00  0.6446998E+00
  9  0.1000000E-05  0.9999990E+00  0.6728238E+00
 10  0.1100000E-05  0.9999989E+00  0.6601472E+00
```

На первый взгляд результаты кажутся правдоподобными. Более внимательный их анализ заставит насторожиться: именно, странна их немонотонность. Насколько они верны?

График функции, табулированной на основе вызовов **q0(x)**:



1. Письменно сформулировать свое отношение к полученному результату (для упрощения его оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **q1**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок обоих графиков, соответствующих таблице.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.8 Табулирование функции $f(x) = \dfrac{p - \sqrt{p^2 - x^7}}{x^7}$

Для двадцати значений аргумента $\mathbf{x} \in [\mathbf{0.1165}, \mathbf{0.1365}]$ равномерно распределенных по указанному промежутку и вводимого параметра p вычислить таблицу значений функции:

$$f(x) = \frac{p - \sqrt{p^2 - x^7}}{x^7}$$

и построить ее график. Некто предложил для расчета программу:

```
      program tsfs2p08
      data ninp / 5 /
      data nres / 6 /
      data a, b, n / 0.1165, 0.1365, 15 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      read(ninp,101) p
      write(nres,*) ' #   p=',p
      write(nres,1000)
      h=(b-a)/n
      do i=1,n+1
        x=a+(i-1)*h
        r0=f0(p,x)
        r1=f1(p,x)
        write(nres,1001) i, x, r0, r1
      enddo
      close(nres)
c                                        Форматы ввода-вывода:
  101 format(e10.3)
 1000 format(1x,' #',2x,'i',9x,'x',12x,'f0(x)',10x,'f1(x)')
 1001 format(1x,i5,2x,e15.7,e15.7,e15.7)
      end
```

в которой табулирование велось двумя способами: через вызовы функций **f0** и **f1**:

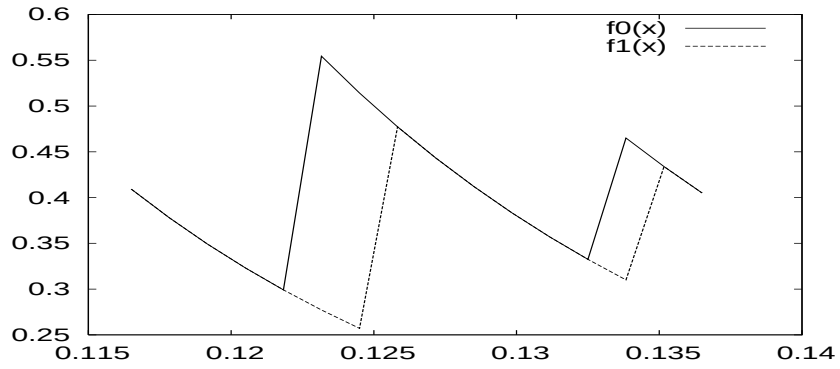
```
      function f0(p,x)                        !      Файл  f0.for
      x7=x**7
      p2=p*p
      f0=(p-sqrt(p2-x7))/x7
      end

      function f1(p,x)                        !     Файл   f1.for
      f1=(p-sqrt(p*p-x**7))/x**7
      end
```

В результате пропуска при $\mathbf{p = 1.3}$ программой получено:

```
#   p=  1.29999995
#  i         x            f0(x)          f1(x)
   1    0.1165000E+00  0.4092877E+00  0.4092877E+00
   2    0.1178333E+00  0.3779489E+00  0.3779488E+00
   3    0.1191667E+00  0.3493226E+00  0.3493225E+00
   4    0.1205000E+00  0.3231475E+00  0.3231475E+00
   5    0.1218333E+00  0.2991901E+00  0.2991901E+00
   6    0.1231667E+00  0.5544825E+00  0.2772412E+00
   7    0.1245000E+00  0.5142267E+00  0.2571134E+00
   8    0.1258333E+00  0.4772767E+00  0.4772767E+00
   9    0.1271667E+00  0.4433301E+00  0.4433302E+00
  10    0.1285000E+00  0.4121148E+00  0.4121148E+00
  11    0.1298333E+00  0.3833862E+00  0.3833862E+00
  12    0.1311667E+00  0.3569240E+00  0.3569240E+00
  13    0.1325000E+00  0.3325287E+00  0.3325287E+00
  14    0.1338333E+00  0.4650303E+00  0.3100202E+00
  15    0.1351667E+00  0.4338543E+00  0.4338543E+00
  16    0.1365000E+00  0.4050446E+00  0.4050445E+00
```

Графики функций **f0(x)** и **f1(x)** при **p=1.3**:



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки результата используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **f2(p,b)**.
5. Обеспечить наглядный вывод результатов всех расчетных формул в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок графиков всех трёх способов расчета.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.9 Табулирование отношения $\dfrac{1-x^{1/2}}{1-x^{1/3}}$ при $x \to 1$

Нужна таблица значений при $\mathbf{x} = \mathbf{exp}(-\mathbf{t^3})$ и $\mathbf{t}$=**0.0045(0.05)0.0095**.
Была составлена подпрограмма-функция **w0**(**x**):

```
c                                                    файл w0.for
      function w0(x)
      w0=(1-sqrt(x))/(1-exp(alog(x)/3))
      end
```

которая вела расчет непосредственно по формуле из заголовка.
Тестирование функции проводилось программой

```
      program tsfs2p09
      data ninp / 5 /
      data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      read(ninp,100) t0,tk
      read(ninp,101) n
      write(nres,1000) t0, tk, n
      write(nres, 1100)
      h=(tk-t0)/n
      do i=0,n
         t=t0+i*h
         x=exp(-t*t*t)
         r0=w0(x)
         write(nres,1101) i, t, x, r0
      enddo
      close(nres)
  100 format(e10.3)
  101 format(i10)
 1000 format(1x,'# t0=',e15.7,5x,'tk='e15.7,5x,'n=',i3)
 1100 format(1x,'# N ',7x,'t',14x,'x',13x,'r0')
 1101 format(1x,i3,e15.7,e15.7,e15.7)
      end
```

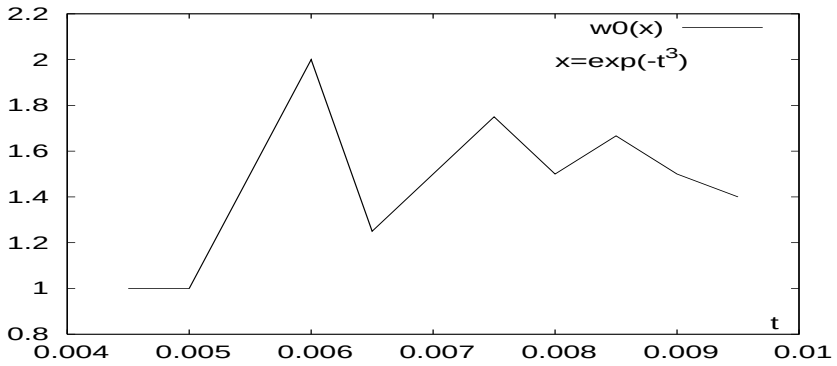
При **t**=**1(0,1)2** тестирование дало результаты, устраивающие заказчика.

Однако, тестирование функции при **t=0,0045(0,0005)0,0095** привело к таблице:

```
# t0=  0.4500000E-02     tk=  0.9500000E-02     n= 10
# N        t              x             r0
  0  0.4500000E-02  0.9999999E+00  0.1000000E+01
  1  0.5000000E-02  0.9999999E+00  0.1000000E+01
  2  0.5500000E-02  0.9999998E+00  0.1500000E+01
  3  0.6000000E-02  0.9999998E+00  0.2000000E+01
  4  0.6500000E-02  0.9999997E+00  0.1250000E+01
  5  0.7000000E-02  0.9999996E+00  0.1500000E+01
  6  0.7499999E-02  0.9999996E+00  0.1750000E+01
  7  0.7999999E-02  0.9999995E+00  0.1500000E+01
  8  0.8500000E-02  0.9999994E+00  0.1666667E+01
  9  0.9000000E-02  0.9999993E+00  0.1500000E+01
 10  0.9500000E-02  0.9999992E+00  0.1400000E+01
```

Насколько верны эти результаты?

График функции, табулированной на основе вызовов **w0(x)**:



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **w1**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок обоих графиков, соответствующих таблице.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.10 Табулирование отношения $\dfrac{\sin^2 x}{1+\cos^3 x}$ при $x \to \pi$

Нужна таблица значений при $\mathbf{x} = \pi + \mathbf{t} \cdot \mathbf{2} \cdot \mathbf{10^{-3}}$ и $\mathbf{t} = \mathbf{-1(0,1)1}$.
Была составлена подпрограмма-функция **rsi0**(**x**):

```
c                                                    файл rsi0.for
      function rsi0(x)
      rsi0=sin(x)**2/(1+cos(x)**3)
      end
```

которая вела расчет непосредственно по формуле из заголовка.
Тестирование функции проводилось программой

```
      program tsfs2p10
      data ninp / 5 /
      data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      pi=4*atan(1e0)
      read(ninp,100) t0,tk
      read(ninp,101) n
      write(nres,1000) t0, tk, n
      write(nres, 1100)
      h=(tk-t0)/n
      do i=0,n
         t=t0+i*h
         x=pi+2e-3*t
         r0=rsi0(x)
         write(nres,1101) i, t, x, r0
      enddo
      close(nres)
  100 format(e10.3)
  101 format(i10)
 1000 format(1x,'# t0=',e15.7,5x,'tk='e15.7,5x,'n=',i3)
 1100 format(1x,'# N ',7x,'t',14x,'x',13x,'r0')
 1101 format(1x,i3,e15.7,e15.7,e15.7)
      end
```

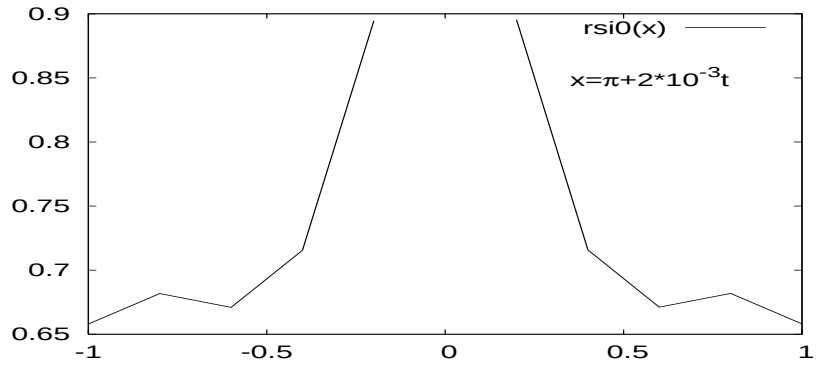
При $\mathbf{t} = \mathbf{-1000(200)1000}$ тестирование дало результаты, устраивающие заказчика.

Однако, тестирование функции при $\mathbf{t} = [-\mathbf{1}, (\mathbf{0.2})\mathbf{1}]$ привело к следующему:

```
# t0= -0.1000000E+01     tk=  0.1000000E+01     n= 10
# N        t              x             r0
  0 -0.1000000E+01  0.3139593E+01  0.6579345E+00
  1 -0.8000000E+00  0.3139993E+01  0.6816897E+00
  2 -0.6000000E+00  0.3140393E+01  0.6709471E+00
  3 -0.4000000E+00  0.3140793E+01  0.7154825E+00
  4 -0.2000000E+00  0.3141193E+01  0.8946908E+00
  5  0.1490116E-07  0.3141593E+01      +Infinity
  6  0.2000000E+00  0.3141993E+01  0.8954731E+00
  7  0.4000000E+00  0.3142393E+01  0.7157953E+00
  8  0.6000000E+00  0.3142793E+01  0.6711426E+00
  9  0.8000000E+00  0.3143193E+01  0.6818387E+00
 10  0.1000000E+01  0.3143593E+01  0.6580495E+00
```

Насколько верны эти результаты?

График функции, табулированной на основе вызовов **rsi0(x)**:



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **rsi1**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок обоих графиков, соответствующих таблице.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.11 Табулирование функции $\dfrac{\sin(x)}{\sqrt{1+tg(x)}-\sqrt{1-tg(x)}}$

В некоторой задаче потребовалась табулировать указанную функцию для $\mathbf{x} = \mathbf{e^{-t}}$ при $t \in [15.5, 16.5]$. Была составлена подпрограмма функции $\mathbf{q0}(\mathbf{t})$:

```
      function q0(x)                          !      Файл  q0.for
      t=tan(x)
      q0=sin(x)/(sqrt(1+t)-sqrt(1-t))
      end
```

которая вела расчет непосредственно по формуле из заголовка. Тестирование функции проводилось программой

```
      program tsfs2p11
      data ninp / 5 /
      data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      read(ninp,100) t0, tn
      read(ninp,101) n
      write(nres,*) ' #   t0=',t0,'  tn=',tn,'  n=',n
      write(nres,1000)
      ht=(tn-t0)/n
      do i=0,n
        t=t0+i*ht
        x=exp(-t)
        r0=q0(x)
        write(nres,1001) i, t, x, r0
      enddo
      close(nres)
c                                        Форматы ввода-вывода:
  100 format(e15.7)
  101 format(i15)
 1000 format(1x,' #',2x,'i',10x,'t',14x,'x',13x,'r0')
 1001 format(1x,i5,2x,e15.7,e15.7,e15.7)
      end
```

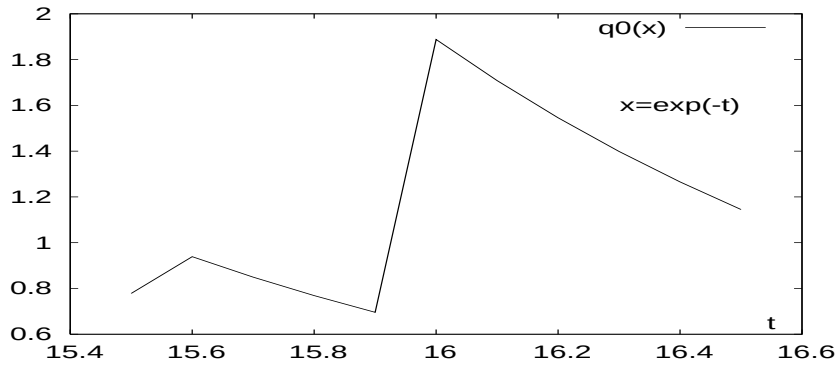
на значениях аргумента $\mathbf{x} \in [\pi/\mathbf{6}, \pi/\mathbf{4}]$ дало результаты, удовлетворяющие заказчика.

Однако, тестирование функции при $\mathbf{t} \in [\mathbf{15.5}, \mathbf{16.5}]$ привело к следующему:

```
#   t0=  15.5  tn=  16.5  n= 10
#  i          t              x             r0
   0    0.1550000E+02  0.1855391E-06  0.7782075E+00
   1    0.1560000E+02  0.1678827E-06  0.9388680E+00
   2    0.1570000E+02  0.1519066E-06  0.8495234E+00
   3    0.1580000E+02  0.1374507E-06  0.7686803E+00
   4    0.1590000E+02  0.1243706E-06  0.6955311E+00
   5    0.1600000E+02  0.1125352E-06  0.1888027E+01
   6    0.1610000E+02  0.1018260E-06  0.1708357E+01
   7    0.1620000E+02  0.9213594E-07  0.1545785E+01
   8    0.1630000E+02  0.8336817E-07  0.1398686E+01
   9    0.1640000E+02  0.7543461E-07  0.1265583E+01
  10    0.1650000E+02  0.6825604E-07  0.1145146E+01
```

Насколько они верны (или неверны) и почему?

График функции, табулированной на основе вызовов **f0(x)**:



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **f1**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок трех графиков, соответствующих таблице.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.12 Расчёт $F(n) = \ln(n!) + n - \sqrt{2\pi} - (n + \frac{1}{2}) \cdot \ln(n)$

Правильный результат положителен и при $\mathbf{n} >> \mathbf{1}$ не должен превосходить $\dfrac{\theta}{12n}$, где $0 < \theta < 1$. Для расчёта была написана функция:

```
      function c0(n)
      pi=4*atan(1.0)                        !      Файл  c0.for
      dn=n
      fln=0.0
      do i=n,2,-1
        di=i
        fln=fln+alog(di)
      enddo
      s23=(dn+0.5)*alog(dn)-dn+0.5*alog(2*pi)
      c0=fln-s23
      end
```

Тестирование ее посредством программы

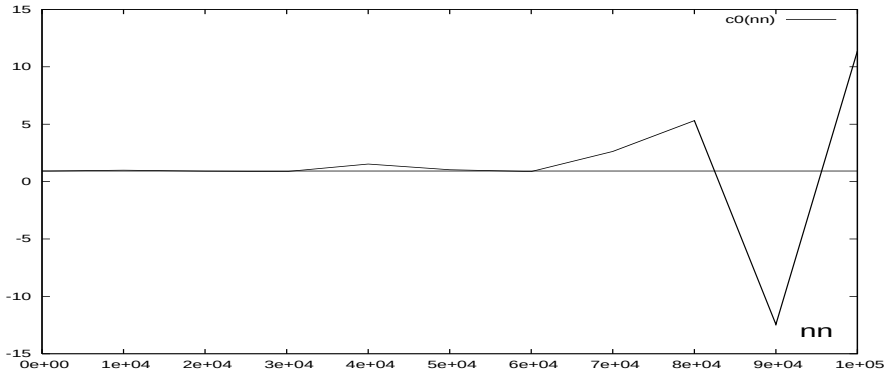
```
      program tsfs2p12
      integer nn(14)
      data ninp / 5 /, nres / 6 /
      pi=4*atan(1.0)
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100) n
      write(nres, *) ' #   n=',n,'      C=',0.5*alog(2*pi)
      read(ninp,100) (nn(k),k=1,n)
      write(nres,1100)
      do k=1,n
        c=c0(nn(k))
        write(nres,1001) k, nn(k), c
      enddo
      write(nres,*) n
      close(nres)
  100 format(i15)
 1100 format(1x,' #',2x,'k',6x,'nn(k)',11x,'C')
 1001 format(1x,i5,2x,i10,2x,e15.7)
      end
```

дало следующий результат

```
#   n= 13      C=  0.918938518
#  k       nn(k)           C
    1          10    0.8331299E-02
    2         100    0.7934570E-03
    3        1000    0.1953125E-02
    4       10000    0.7031250E-01
    5       20000   -0.1562500E-01
    6       30000   -0.3125000E-01
    7       40000    0.6250000E+00
    8       50000    0.1250000E+00
    9       60000   -0.6250000E-01
   10       70000    0.1687500E+01
   11       80000    0.4375000E+01
   12       90000   -0.1337500E+02
   13      100000    0.1050000E+02
13
```

Насколько они верны (или неверны) и почему?

1. Письменно сформулировать свое отношение к полученному результату.
2. Объяснить объективную и субъективную причины его появления.
3. Преобразовать расчетную формулу так, чтобы результат был верен.
4. Написать соответствующую новой схеме расчета функцию **c1(n)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **c0(n)** и **c1(n)**.
7. Модифицировать программу для получения результатов типа **real(mp)**.



8.13 Табулирование функции $\dfrac{2-\sqrt{4+e^{-x}}}{3-\sqrt{2(4+e^{-x})+1}}$

Функцию, указанную в заголовке, необходимо табулировать при $\mathbf{x = 12(0.1)13}$.

Была составлена подпрограмма-функция $\mathbf{v0(x)}$:

```
      function v0(x)                          !      Файл  v0.for
      w=4+exp(-x)
      v0=(2-sqrt(w))/(3-sqrt(2*w+1))
      end
```

которая вела расчет непосредственно по формуле из заголовка. Тестирование функции проводилось программой

```
      program tsfs2p13
      data ninp / 5 /, nres / 6 /
      pi=4*atan(1.0)
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100) x0,xn
      read(ninp,101) n
      write(nres, *) ' #   x0=',x0,'    xn=',xn,'    n=',n
      hx=(xn-x0)/n
      write(nres,1100)
      do i=0,n
        x=x0+i*hx
        r0=v0(x)
        write(nres,1001) i, x,r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',6x,'x',11x,'v0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

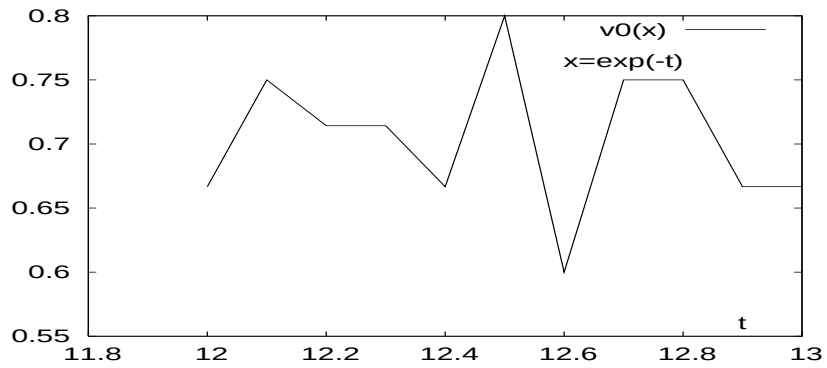
и на значениях аргумента из диапазона $\mathbf{t} = [\mathbf{-2}, \mathbf{-3}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы.

Однако, тестирование функции при $\mathbf{t} \in [\mathbf{12}, \mathbf{13}]$ привело к следующему:

```
#   x0=  12.    xn=  13.    n= 10
#  i       x           v0
   0      0.1200000E+02  0.6666667E+00
   1      0.1210000E+02  0.7500000E+00
   2      0.1220000E+02  0.7142857E+00
   3      0.1230000E+02  0.7142857E+00
   4      0.1240000E+02  0.6666667E+00
   5      0.1250000E+02  0.8000000E+00
   6      0.1260000E+02  0.6000000E+00
   7      0.1270000E+02  0.7500000E+00
   8      0.1280000E+02  0.7500000E+00
   9      0.1290000E+02  0.6666667E+00
  10      0.1300000E+02  0.6666667E+00
```

Насколько они верны (или неверны) и почему?

График функции **v0(x)** при **x=exp(-t)** (по оси абсцисс отложено t):



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **v1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **v0(x)** и **v1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.14 Проверка формулы $\dfrac{a^2-b^2}{a-b}=a+b$

Школьник решил проверить численно на компьютере правильность алгебраической формулы из заголовка и написал функцию:

```
c                                                    файл apb.for
      function apb0(a,b)
      a2=a*a
      b2=b*b
      apb0=(a2-b2)/(a-b)
      end
```

Тестирующая программа вводила **b**, вычисляя **a** посредством добавления к содержимому переменной **b** некоторого уменьшающегося приращения **x**, после чего вызывалась упомянутая функция **apb0** и находились абсолютная и относительная погрешности её работы (для расчёта точного значения использовалась правая часть формулы из заголовка).

```
      program tsfs2p14
      data ninp / 5 /
      data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result',status='replace')
      read(ninp,100) b
      write(nres,1000) b
      write(nres, 1100)
      x=b
      b2=b*b
      do while (a.ne.b)
        a=b+x
        r0=apb0(a,b)
        apb=a+b
        aer=abs(r0-apb)
        rer=abs(aer/apb)
        write(nres,1101) x, r0, aer, rer
        x=x/10
      enddo
      close(nres)
  100 format(e10.3)
 1000 format(1x,'# b=',e15.7,10x/' # a=b+x')
 1100 format(1x,'#',7x,'x',8x,'(a2-b2)/(a-b)',4x,'aer',6x,'rer')
 1101 format(1x,e15.7,e15.7,e10.2,e10.2)
      end
```

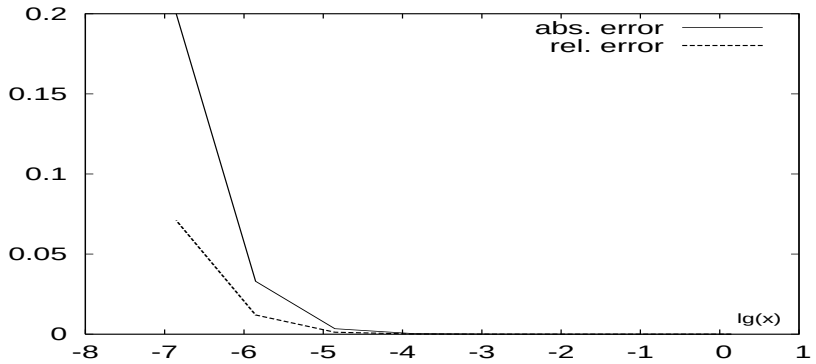
Начальное значения аргумента **x** полагалось равным **b**, каждое следующее вдесятеро меньше предыдущего. Уменьшение прекращалось, как только значение переменной **a** *"с точки зрения ЭВМ"* в точности оказывалось равной значению **b**.

Помогите школьнику понять получаемые результаты:

```
# b=  0.1400000E+01
# a=b+x
#       x         (a2-b2)/(a-b)    aer      rer
  0.1400000E+01  0.4200000E+01  0.00E+00  0.00E+00
  0.1400000E+00  0.2940000E+01  0.24E-06  0.81E-07
  0.1400000E-01  0.2814000E+01  0.00E+00  0.00E+00
  0.1400000E-02  0.2801430E+01  0.31E-04  0.11E-04
  0.1400000E-03  0.2799830E+01  0.31E-03  0.11E-03
  0.1400000E-04  0.2803419E+01  0.34E-02  0.12E-02
  0.1400000E-05  0.2833333E+01  0.33E-01  0.12E-01
  0.1400000E-06  0.3000000E+01  0.20E+00  0.71E-01
  0.1400000E-07             NaN       NaN       NaN
```

Насколько они верны (или неверны) и почему?

1. Письменно сформулировать свое отношение к полученному результату.
2. Объяснить объективную и субъективную причины его появления.
3. Преобразовать расчетную формулу так, чтобы результат был верен.
4. Написать соответствующую новой схеме расчета функцию **apb1**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода на один рисунок трех графиков, соответствующих таблице.
7. Модифицировать программу для получения результатов типа **real(mp)**.



8.15 Табулирование функции $(1+x)^{\frac{1}{x}}$

В некоторой задаче потребовалась табулировать указанную функцию для $\mathbf{x} = \mathbf{10^{-t}}$ при $t = 15(0.1)16]$. Были составлены две подпрограммы-функции $\mathbf{u0}(\mathbf{x})$ и $\mathbf{u1}(\mathbf{x})$:

```
      function u0(x)                              !      Файл  u0.for
      u0=(1+x)**(1.0/x)
      end

      function u1(x)                                ! файл u1.for
      x1=1+x
      u1=x1**(1.0/x)
      end
```

$\mathbf{u0}(\mathbf{x})$ вела расчет непосредственно по формуле из заголовка, а $\mathbf{u1}(\mathbf{x})$, в отличие от $\mathbf{u0}(\mathbf{t})$, значение основания предварительно записывала в рабочую переменную, возводя в степень содержимое последней. Тестирование функции проводилось программой

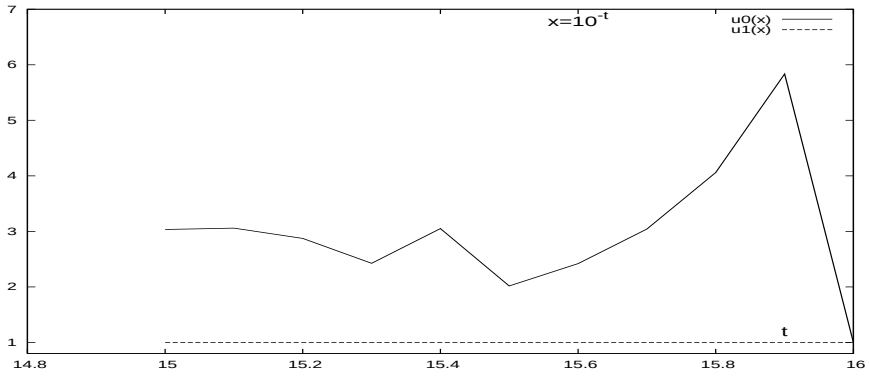
```
      program tsfs2p15
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result')
      read(ninp,100) t0, tn
      read(ninp,101) n
      write(nres,*)   ' #   t0=',t0,'  tn=',tn,'  n=',n
      write(nres,1000)
      ht=(tn-t0)/n
      do i=0,n
        t=t0+i*ht
        x=10**(-t)
        r0=u0(x)
        r1=u1(x)
        write(nres,1001) i, t, x, r0, r1
      enddo
  100 format(e15.7)
  101 format(i15)
 1000 format(1x,' #',2x,'i',10x,'t',13x,'x',14x,'r0',13x,'r1')
 1001 format(1x,i5,2x,e15.7,e15.7,e15.7,e15.7)
      end
```

на значениях аргумента $\mathbf{t} = [\mathbf{7}, \mathbf{11}]$ дало результаты, удовлетворяющие заказчика и совпадающие численно с точностью до семи значащих цифр на одинарной точности с аналитическим значением второго замечательного предела. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{15}, \mathbf{16}]$ привело к следующему:

```
#   t0=   15.00000       tn=   16.00000       n=          10
#  i          t             x             r0            r1
    0    0.1500000E+02  0.1000000E-14  0.1000000E+01  0.1000000E+01
    1    0.1510000E+02  0.7943275E-15  0.1000000E+01  0.1000000E+01
    2    0.1520000E+02  0.6309576E-15  0.1000000E+01  0.1000000E+01
    3    0.1530000E+02  0.5011870E-15  0.1000000E+01  0.1000000E+01
    4    0.1540000E+02  0.3981075E-15  0.1000000E+01  0.1000000E+01
    5    0.1550000E+02  0.3162278E-15  0.1000000E+01  0.1000000E+01
    6    0.1560000E+02  0.2511884E-15  0.1000000E+01  0.1000000E+01
    7    0.1570000E+02  0.1995263E-15  0.1000000E+01  0.1000000E+01
    8    0.1580000E+02  0.1584892E-15  0.1000000E+01  0.1000000E+01
    9    0.1590000E+02  0.1258927E-15  0.1000000E+01  0.1000000E+01
   10    0.1600000E+02  0.1000000E-15  0.1000000E+01  0.1000000E+01
```

Насколько они верны (или неверны) и почему?

1. Письменно сформулировать свое отношение к полученному результату.
2. Объяснить объективную и субъективную причины его появления.
3. Преобразовать расчетную формулу так, чтобы результат был верен.
4. Написать соответствующую новой схеме расчета функцию **u2(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило вывода рисунка с графиками **u0(x)**, **u1(x)**, **u2(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)**.



8.16 Табулирование функции $\dfrac{1-\cos x}{1-\cos\dfrac{x}{2}}$

Функцию, указанную в заголовке, необходимо табулировать для $\mathbf{x} = \mathbf{exp}(-\mathbf{t})$ при $\mathbf{t} = \mathbf{6.55}(\mathbf{0.1})\mathbf{7.55}]$.

Была составлена подпрограмма-функция $\mathbf{q0}(\mathbf{x})$:

```
function q0(x)                          !      Файл  q0.for
q0=(1-cos(x))/(1-cos(x/2))
end
```

которая вела расчет непосредственно по формуле из заголовка Тестирование функции проводилось программой

```
      program tsfs2p16                        !    Файл  tsfs2p16.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100) t0,tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=t0+i*ht
        x=exp(-t)
        r0=q0(x)
        write(nres,1001) i, t,r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',12x,'q0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

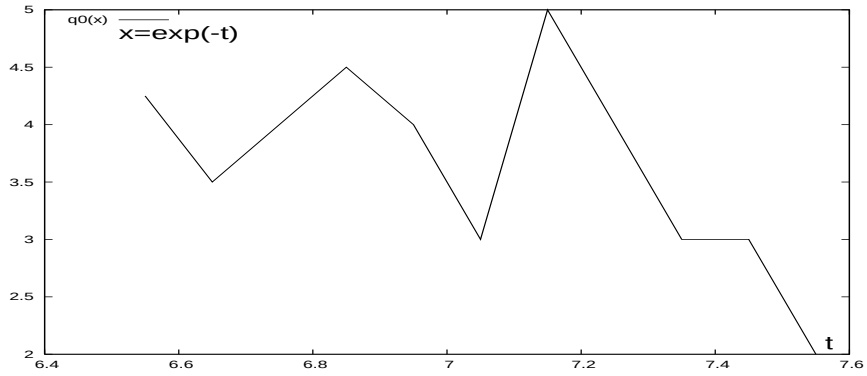
на значениях аргумента $\mathbf{t} = [\mathbf{0.2}, \mathbf{0.3}]$ дало результаты, верные в пределах семи значащих цифр на одинарной точности.

Однако, тестирование функции при $\mathbf{t} \in [\mathbf{6.55}, \mathbf{7.55}]$ привело к следующему:

```
#   t0=  6.55000019    tn=  7.55000019    n= 10
#  i            t            q0
   0      0.6550000E+01  0.4250000E+01
   1      0.6650000E+01  0.3500000E+01
   2      0.6750000E+01  0.4000000E+01
   3      0.6850000E+01  0.4500000E+01
   4      0.6950000E+01  0.4000000E+01
   5      0.7050000E+01  0.3000000E+01
   6      0.7150000E+01  0.5000000E+01
   7      0.7250000E+01  0.4000000E+01
   8      0.7350000E+01  0.3000000E+01
   9      0.7450000E+01  0.3000000E+01
  10      0.7550000E+01  0.2000000E+01
```

Насколько они верны (или неверны) и почему?

График по алгоритму **q0(x)** при **x=exp(-t)** (по оси абсцисс отложено t):



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **q1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **q0(x)** и **q1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.17 Табулирование функции $\dfrac{\cos 2x - \cos x}{\sin 2x - \sin x}$

Функцию, указанную в заголовке, необходимо табулировать для $\mathbf{x} = \mathbf{exp}(-\mathbf{t})$ при $\mathbf{t} = \mathbf{8}(\mathbf{0.1})\mathbf{9}]$.

Была составлена подпрограмма-функция $\mathbf{s0}(\mathbf{x})$:

```
      function s0(x)                          !      Файл  s0.for
      s0=(cos(2*x)-cos(x))/(sin(2*x)-sin(x))
      end
```

которая вела расчет непосредственно по формуле из заголовка. Тестирование функции проводилось программой

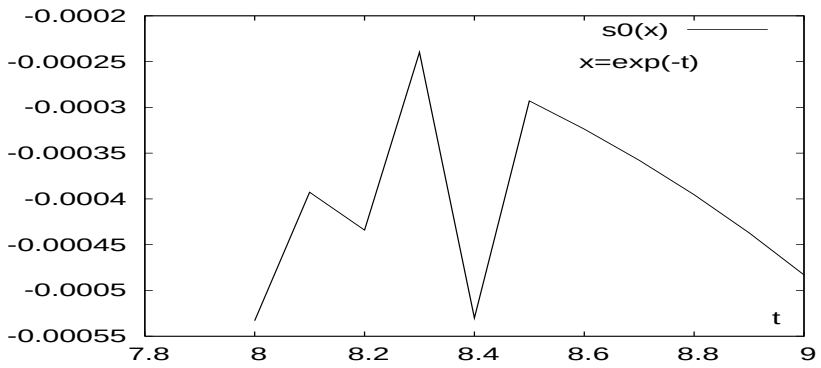
```
      program tsfs2p17                       !    Файл  tsfs2p16.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=t0+i*ht
        x=exp(-t)
        r0=s0(x)
        write(nres,1001) i, t,r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',12x,'q0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

и на значениях аргумента из диапазона $\mathbf{t} = [\mathbf{0.5}, \mathbf{0.6}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы.

Однако, тестирование функции при $\mathbf{t} \in [\mathbf{8}, \mathbf{9}]$ привело к следующему:

```
#   t0=  8.    tn=  9.    n= 10
#  i            t            q0
   0      0.8000000E+01 -0.5330369E-03
   1      0.8100000E+01 -0.3927314E-03
   2      0.8200000E+01 -0.4340350E-03
   3      0.8300000E+01 -0.2398415E-03
   4      0.8400000E+01 -0.5301315E-03
   5      0.8500000E+01 -0.2929430E-03
   6      0.8600000E+01 -0.3237523E-03
   7      0.8700000E+01 -0.3578014E-03
   8      0.8800000E+01 -0.3954318E-03
   9      0.8900000E+01 -0.4370195E-03
  10      0.9000000E+01 -0.4829814E-03
```

Насколько они верны (или неверны) и почему?
График функции по алгоритму **q0(x)** при **x=exp(-t)** (по оси абсцисс отложено t):



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **s1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **s0(x)** и **s1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.18 Табулирование функции $\dfrac{1.7-(1.7^3-x)^{\frac{1}{3}}}{x}$

Функцию, указанную в заголовке, необходимо табулировать для $\mathbf{x}=\mathbf{exp}(-\mathbf{t^2})$ при $\mathbf{t}=\mathbf{3.5}(\mathbf{0.02})\mathbf{3.8}]$.

Была составлена подпрограмма-функция **fun0**(**x**):

```
      function fun0(x)                          !      Файл  fun0.for
      fun0=(1.7-(1.7**3-x)**(1.0/3))/x
      end
```

которая вела расчет непосредственно по формуле из заголовка. Тестирование функции проводилось программой

```
      program tsfs2p18                        !    Файл  tsfs2p18.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=t0+i*ht
        x=exp(-t*t)
        r0=fun0(x)
        write(nres,1001) i, t, r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',11x,'fun0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

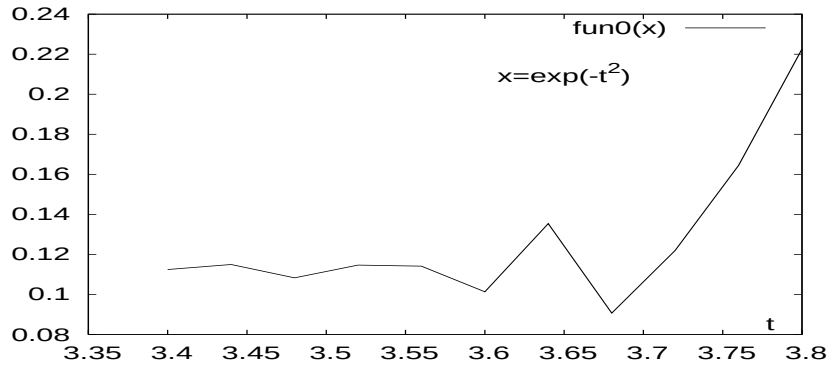
и на значениях аргумента из диапазона $\mathbf{t}=[\mathbf{0},\mathbf{0.6}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы.

Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{3.5}, \mathbf{3.8}]$ привело к следующему:

```
#   t0=  3.4000001    tn=  3.79999995    n= 10
#  i           t           fun0
   0      0.3400000E+01  0.1124597E+00
   1      0.3440000E+01  0.1149941E+00
   2      0.3480000E+01  0.1083328E+00
   3      0.3520000E+01  0.1146707E+00
   4      0.3560000E+01  0.1141578E+00
   5      0.3600000E+01  0.1013436E+00
   6      0.3640000E+01  0.1353844E+00
   7      0.3680000E+01  0.9071934E-01
   8      0.3720000E+01  0.1219694E+00
   9      0.3760000E+01  0.1645098E+00
  10      0.3800000E+01  0.2225985E+00
```

Насколько они верны (или неверны) и почему?

График по алгоритму **fun0(x)** при **x=exp(-t*t)** (по оси абсцисс отложено t):



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **fun1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **fun0(x)** и **fun1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

8.19 Табулирование функции $\dfrac{\sin x - \tan x}{4\sin^3 x/2}$

Функцию, указанную в заголовке, необходимо табулировать для $\mathbf{x} = \mathbf{exp}(-\mathbf{t^2})$ при $\mathbf{t} = \mathbf{2.7}(\mathbf{0.1})\mathbf{2.8}]$.

Была составлена подпрограмма-функция **sts0**(**x**):

```
      function sts0(x)                          !    Файл sts0.for
      sts0=(sin(x)-tan(x))/4/sin(x/2)**3
      end
```

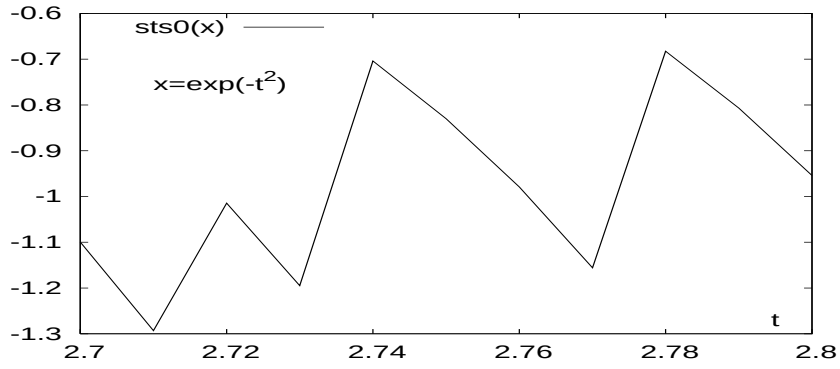
которая вела расчет непосредственно по формуле из заголовка. Тестирование функции проводилось программой

```
      program tsfs2p19                        !    Файл  tsfs2p19.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=t0+i*ht
        x=exp(-t*t)
        r0=sts0(x)
        write(nres,1001) i, t, r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',12x,'sts0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

и на значениях аргумента из диапазона $\mathbf{t} = [\mathbf{0}, \mathbf{1.0}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{2.7}, \mathbf{2.8}]$ привело к следующему:

```
#   t0=  2.70000005    tn=  2.79999995    n= 10
#  i            t             sts0
   0      0.2700000E+01 -0.1099391E+01
   1      0.2710000E+01 -0.1293117E+01
   2      0.2720000E+01 -0.1014596E+01
   3      0.2730000E+01 -0.1194814E+01
   4      0.2740000E+01 -0.7039437E+00
   5      0.2750000E+01 -0.8299773E+00
   6      0.2760000E+01 -0.9791631E+00
   7      0.2770000E+01 -0.1155858E+01
   8      0.2780000E+01 -0.6826285E+00
   9      0.2790000E+01 -0.8067796E+00
  10      0.2800000E+01 -0.9540827E+00
```

Насколько они верны (или неверны) и почему?
График по алгоритму **sts0(x)** при $\mathbf{x = exp(-t^2)}$ (по оси абсцисс отложено t):



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **sts1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **sts0(x)** и **sts1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

8.20 Табулирование функции $x - \sqrt{x^2 + 5x}$.

В некоторой задаче потребовалась табулировать указанную функцию для $\mathbf{x} = \mathbf{exp}(\mathbf{t})$ при $\mathbf{t} = \mathbf{15.8}(\mathbf{0.02})\mathbf{17.8}]$. Были составлены две подпрограммы-функции $\mathbf{w0}(\mathbf{x})$ и $\mathbf{w1}(\mathbf{x})$:

```
      function w0(x)                              !    Файл w0.for
      w0=x-sqrt(x**2+5*x)
      end
      function w1(x)                              !
      w1=x*(1-sqrt(1+5/x))
      end
```

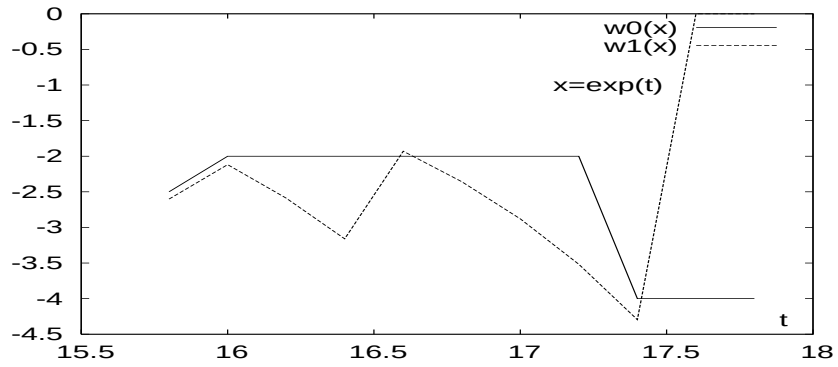
Первая вела расчет непосредственно по формуле из заголовка; вторая – по слегка модифицированной, полученной вынесением аргумента $\mathbf{x}$ из под знака корня и выделением аргумента в качестве явного сомножителя. Тестирование функций проводилось программой

```
      program tsfs2p20                        !    Файл  tsfs2p20.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=t0+i*ht
        x=exp(t)
        r0=w0(x)
        r1=w1(x)
        write(nres,1001) i, t, r0, r1
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0',13x,'w1')
 1001 format(1x,i5,2x,2x,e15.7,e15.7,e15.7)
      end
```

и при значениях аргумента $\mathbf{t}$ из диапазона $\mathbf{t} = [\mathbf{0}, \mathbf{1.0}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{15.8}, \mathbf{17.8}]$ привело к следующему:

```
#   t0=  15.8000002    tn=  17.7999992    n= 10
#  i            t              w0             w1
   0      0.1580000E+02 -0.2500000E+01 -0.2601862E+01
   1      0.1600000E+02 -0.2000000E+01 -0.2118614E+01
   2      0.1620000E+02 -0.2000000E+01 -0.2587683E+01
   3      0.1640000E+02 -0.2000000E+01 -0.3160599E+01
   4      0.1660000E+02 -0.2000000E+01 -0.1930184E+01
   5      0.1680000E+02 -0.2000000E+01 -0.2357529E+01
   6      0.1700000E+02 -0.2000000E+01 -0.2879495E+01
   7      0.1720000E+02 -0.2000000E+01 -0.3517019E+01
   8      0.1740000E+02 -0.4000000E+01 -0.4295700E+01
   9      0.1760000E+02 -0.4000000E+01  0.0000000E+00
  10      0.1780000E+02 -0.4000000E+01  0.0000000E+00
```

Насколько они верны (или неверны) и почему?
Графики функции, табулированной алгоритмами **w0(x)** и **w1(x)** при **x=exp(t)** (по оси абсцисс отложено t):



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **w2(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **w0(x)**, **w1(x)** и **w2(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

8.21 Табулирование функции $\sqrt{tg^2x + \frac{1.23}{cos(x)}} - tg(x)$.

В некоторой задаче потребовалась табулировать указанную функцию для $x = \frac{\pi}{2} - e^{-t}$ при $t = 12.3(0.4)16.3$. Была составлена подпрограмма-функция **w0**(**x**):

```
      function tg0(x)                              !    Файл tg0.for
      tg=tan(x)
      tg2=tg*tg
      tg0=sqrt(tg2+1.23/cos(x))-tg
      return
      end
```

которая вела расчет непосредственно по формуле из заголовка. Тестирование функций проводилось программой

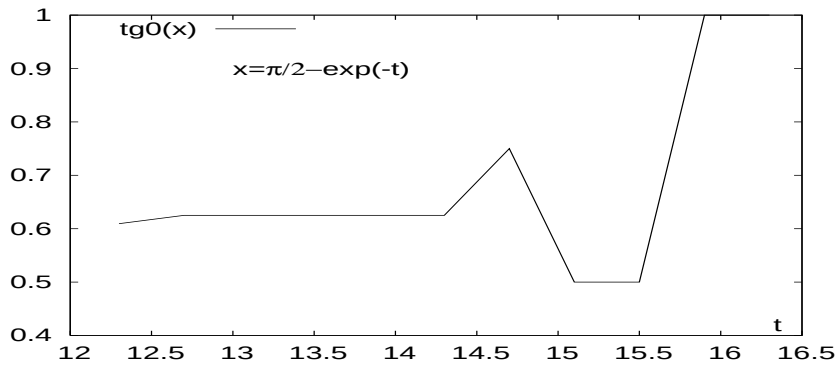
```
      program tsfs2p21                        !    Файл  tsfs2p21.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      pi2=2*atan(1.0)
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=t0+i*ht
        x=pi2-exp(-t)
        r0=tg0(x)
        write(nres,1001) i, t, r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'tg0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

и при значениях аргумента **t** из диапазона $\mathbf{t} = [\mathbf{0}, \mathbf{1.0}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{12.3}, \mathbf{16.3}]$ привело к следующему:

```
#   t0=  12.3000002    tn=  16.2999992    n= 10
#  i            t              tg0
   0      0.1230000E+02  0.6093750E+00
   1      0.1270000E+02  0.6250000E+00
   2      0.1310000E+02  0.6250000E+00
   3      0.1350000E+02  0.6250000E+00
   4      0.1390000E+02  0.6250000E+00
   5      0.1430000E+02  0.6250000E+00
   6      0.1470000E+02  0.7500000E+00
   7      0.1510000E+02  0.5000000E+00
   8      0.1550000E+02  0.5000000E+00
   9      0.1590000E+02  0.1000000E+01
  10      0.1630000E+02  0.1000000E+01
```

Насколько они верны (или неверны) и почему?

График функции, табулированной алгоритмом **tg0(x)** при $x = \frac{\pi}{2} - \exp -t$ (по оси абсцисс отложено t):



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **tg1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **tg0(x)** и **tg1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)**.

8.22 Табулирование функции $\frac{cos(2x)}{sin(x)-cos(x)}$.

Функцию, указанную в заголовке, необходимо табулировать для $\mathbf{x} = \frac{\pi}{4} - \mathbf{e}^{-\mathbf{t}}$ при $\mathbf{t} = \mathbf{15}(\mathbf{0.2})\mathbf{17}$.

Была составлена подпрограмма-функция $\mathbf{w0}(\mathbf{x})$:

```
function w0(x)                              !    Файл w0.for
w0=cos(2*x)/(sin(x)-cos(x))
end
```

которая вела расчет непосредственно по формуле из заголовка.

Тестирование функций проводилось программой:

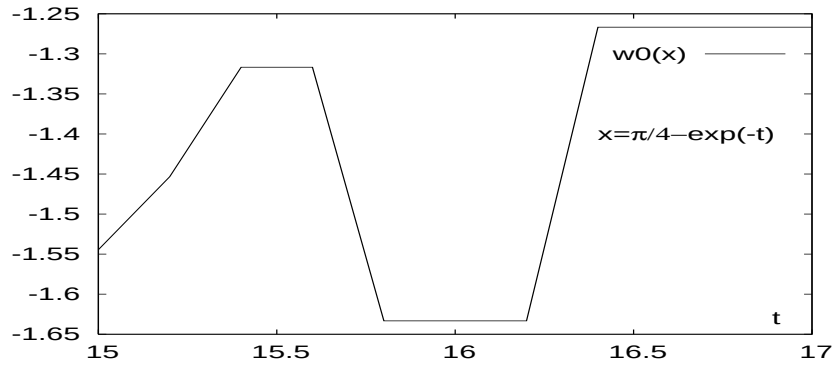
```
      program tsfs2p22                      !    Файл  tsfs2p22.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      pi4=atan(1.0)
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=t0+i*ht
        x=pi4-exp(-t)
        r0=w0(x)
        write(nres,1001) i, t, r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

и при значениях аргумента $\mathbf{t}$ из диапазона $\mathbf{t} = [\mathbf{0}, \mathbf{1.0}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{15}, \mathbf{17}]$ привело к следующему:

```
#   t0=  15.    tn=  17.    n= 10
#  i            t              w0
   0      0.1500000E+02 -0.1544441E+01
   1      0.1520000E+02 -0.1453329E+01
   2      0.1540000E+02 -0.1316661E+01
   3      0.1560000E+02 -0.1316661E+01
   4      0.1580000E+02 -0.1633322E+01
   5      0.1600000E+02 -0.1633322E+01
   6      0.1620000E+02 -0.1633322E+01
   7      0.1640000E+02 -0.1266645E+01
   8      0.1660000E+02 -0.1266645E+01
   9      0.1680000E+02 -0.1266645E+01
  10      0.1700000E+02 -0.1266645E+01
```

Насколько они верны (или неверны) и почему?

Графики функции, табулированной алгоритмом **w0(x)** при $\mathbf{x} = \frac{\pi}{4} - \exp -\mathbf{t}$ (по оси абсцисс отложено t):



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **w1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **w0(x)** и **w1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

8.23 Табулирование функции $\dfrac{\sin x}{\sqrt{2(1-\cos x)}}$.

Функция, указанная в заголовке, табулируется для $\mathbf{x} = \mathbf{e}^{-\mathbf{t}}$ при $\mathbf{t} = \mathbf{7.3}(\mathbf{0.1})\mathbf{8.3}$.

Была составлена подпрограмма-функция **w0**(**x**):

```
      function w0(x)                              !    Файл w0.for
      w0=sin(x)/sqrt(2*(1-cos(x)))
      end
```

которая вела расчет непосредственно по формуле из заголовка.

Тестирование функций проводилось программой

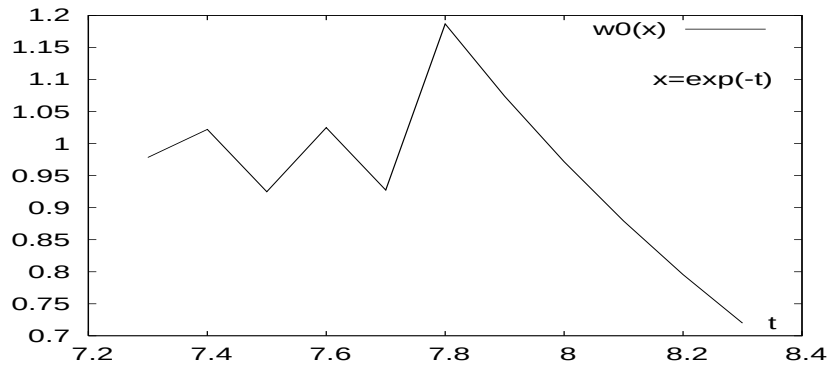
```
      program tsfs2p23                        !    Файл  tsfs2p23.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      pi4=atan(1.0)
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=t0+i*ht
        x=exp(-t)
        r0=w0(x)
        write(nres,1001) i, t, r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7,e15.7)
      end
```

и при значениях аргумента **t** из диапазона $\mathbf{t} = [\mathbf{0.5}, \mathbf{0.6}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{7.3}, \mathbf{8.3}]$ привело к следующему:

```
#   t0=  7.30000019    tn=  8.30000019    n= 10
#  i            t              w0
   0      0.7300000E+01  0.9782844E+00
   1      0.7400000E+01  0.1022127E+01
   2      0.7500000E+01  0.9248593E+00
   3      0.7600000E+01  0.1024924E+01
   4      0.7700000E+01  0.9273897E+00
   5      0.7800000E+01  0.1186719E+01
   6      0.7900000E+01  0.1073788E+01
   7      0.8000000E+01  0.9716036E+00
   8      0.8100000E+01  0.8791429E+00
   9      0.8200000E+01  0.7954819E+00
  10      0.8300000E+01  0.7197815E+00
```

Насколько они верны (или неверны) и почему?

Графики функции, табулированной алгоритмом **w0(x)** при $\mathbf{x = e^{-t}}$:



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **w1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **w0(x)** и **w1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

8.24 Табулирование функции $\sqrt{\dfrac{1-\cos x}{1+\cos x)}}$.

Функция, указанная в заголовке, табулируется для $\mathbf{x} = \mathbf{e}^{-\mathbf{t}}$ при $\mathbf{t} = \mathbf{7.3}(\mathbf{0.1})\mathbf{8.3}$.

Была составлена подпрограмма-функция $\mathbf{w0}(\mathbf{x})$:

```
function w0(x)                              !    Файл w0.for
w0=sqrt( (1-cos(x)) / (1+cos(x)) )
end
```

которая вела расчет непосредственно по формуле из заголовка.

Тестирование функций проводилось программой

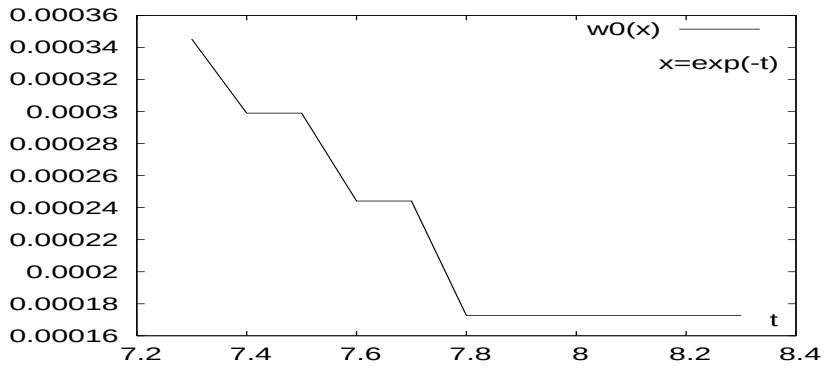
```
      program tsfs2p24                      !    Файл  tsfs2p24.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      pi4=atan(1.0)
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=(t0+i*ht)
        x=exp(-t)
        r0=w0(x)
        write(nres,1001) i, t, r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

и при значениях аргумента $\mathbf{t}$ из диапазона $\mathbf{t} = [\mathbf{0.5}, \mathbf{0.6}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{7.3}, \mathbf{8.3}]$ привело к следующему:

```
#   t0=  7.30000019    tn=  8.30000019    n= 10
#  i            t              w0
   0      0.7300000E+01  0.3452670E-03
   1      0.7400000E+01  0.2990100E-03
   2      0.7500000E+01  0.2990100E-03
   3      0.7600000E+01  0.2441406E-03
   4      0.7700000E+01  0.2441406E-03
   5      0.7800000E+01  0.1726335E-03
   6      0.7900000E+01  0.1726335E-03
   7      0.8000000E+01  0.1726335E-03
   8      0.8100000E+01  0.1726335E-03
   9      0.8200000E+01  0.1726335E-03
  10      0.8300000E+01  0.1726335E-03
```

Насколько они верны (или неверны) и почему?

Графики функции, табулированной алгоритмом **w0(x)** при $\mathbf{x} = \mathbf{e}^{-\mathbf{t}}$:



1. Письменно сформулировать свое отношение к полученному результату. (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **w1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **w0(x)** и **w1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

8.25 Табулирование функции $\dfrac{\sin 3x - 3\sin x}{4\sin^3 x}$.

В некоторой задаче потребовалась табулировать указанную функцию для $\mathbf{x} = \mathbf{e^{-t}}$ при $\mathbf{t} = \mathbf{9.3(0.1)10.3}$.

Была составлена подпрограмма-функции $\mathbf{w0(x)}$:

```
      function w0(x)
      sx=sin(x)                        !    Файл w0.for
      w0=(sin(3*x)-3*sx)/4/sx/sx/sx
      end
      function w1(x)
      w1=(sin(3*x)-3*sin(x))/4/sin(x)**3
      end
```

которая вела расчет непосредственно по формуле из заголовка.

Тестирование функций проводилось программой

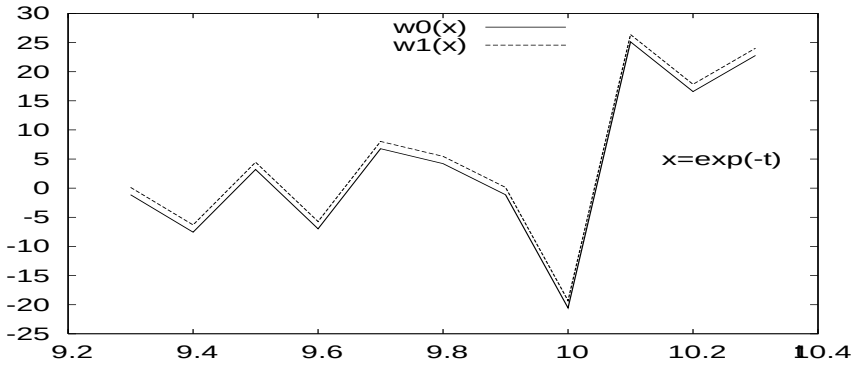
```
      program tsfs2p25                        !    Файл  tsfs2p25.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      pi=4*atan(1.0)
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=(t0+i*ht)
        x=exp(-t)
        r0=w0(x)
        r1=w1(x)
        write(nres,1001) i, t, x, r0, r1
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'x',14x,'w0',14x,'w1')
 1001 format(1x,i5,2x,2x,e15.7,e15.7,e15.7,e15.7)
      end
```

и при значениях аргумента $\mathbf{t}$ из диапазона $\mathbf{t} = [\mathbf{0.9}, \mathbf{1.0}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{9.3}, \mathbf{10.3}]$ привело к следующему:

```
#   t0=   9.300000         tn=   10.30000         n=          10
#  i            t              x              w0             w1
   0      0.9300000E+01  0.9142421E-04 -0.1125000E+01  0.1250000E+00
   1      0.9400001E+01  0.8272401E-04 -0.7551364E+01 -0.6301364E+01
   2      0.9500000E+01  0.7485183E-04  0.3212334E+01  0.4462335E+01
   3      0.9600000E+01  0.6772871E-04 -0.6979796E+01 -0.5729796E+01
   4      0.9700000E+01  0.6128351E-04  0.6778134E+01  0.8028134E+01
   5      0.9800000E+01  0.5545159E-04  0.4209063E+01  0.5459063E+01
   6      0.9900001E+01  0.5017465E-04 -0.1125000E+01  0.1250000E+00
   7      0.1000000E+02  0.4539993E-04 -0.2056358E+02 -0.1931358E+02
   8      0.1010000E+02  0.4107954E-04  0.2511437E+02  0.2636437E+02
   9      0.1020000E+02  0.3717032E-04  0.1658470E+02  0.1783470E+02
  10      0.1030000E+02  0.3363309E-04  0.2278061E+02  0.2403061E+02
```

Насколько они верны (или неверны) и почему?

Графики функций, табулированных алгоритмами **w0(x)** и **w1(x)** при $\mathbf{x = e^{-t}}$:



1. Письменно сформулировать свое отношение к полученному результату. (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **w1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **w0(x)** и **w1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

8.26 Табулирование функции $\dfrac{\cos 4x - 1}{8\cos^4 x - 8\cos^2 x}$.

В некоторой задаче потребовалась табулировать указанную функцию для $\mathbf{x} = \mathbf{e^{-t}}$ при $\mathbf{t} = \mathbf{9.3(0.1)10.3}$.

Была составлена подпрограмма-функции **w0**(**x**):

```
function w0(x)                              !    Файл w0.for
w0=(cos(4*x)-1)/(8*cos(x)**4-8*cos(x)**2)
end
```

которая вела расчет непосредственно по формуле из заголовка.

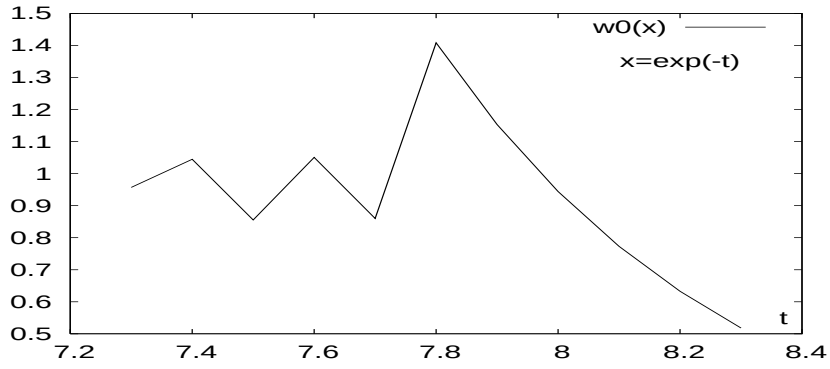
Тестирование функций проводилось программой

```
      program tsfs2p26                      !    Файл  tsfs2p26.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result',status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      pi4=atan(1.0)
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=(t0+i*ht)
        x=exp(-t)
        r0=w0(x)
        write(nres,1001) i, t, x, r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7,e15.7)
      end
```

и при значениях аргумента **t** из диапазона $\mathbf{t} = [\mathbf{0.9}, \mathbf{1.0}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{9.3}, \mathbf{10.3}]$ привело к следующему:

```
#   t0=   7.300000         tn=   8.300000         n=          10
#  i            t              w0
    0      0.7300000E+01  0.6755387E-03  0.9570406E+00
    1      0.7400000E+01  0.6112527E-03  0.1044745E+01
    2      0.7500000E+01  0.5530844E-03  0.8553649E+00
    3      0.7600000E+01  0.5004512E-03  0.1050470E+01
    4      0.7700000E+01  0.4528271E-03  0.8600519E+00
    5      0.7800000E+01  0.4097349E-03  0.1408302E+01
    6      0.7900000E+01  0.3707435E-03  0.1153021E+01
    7      0.8000000E+01  0.3354626E-03  0.9440135E+00
    8      0.8100000E+01  0.3035390E-03  0.7728922E+00
    9      0.8200000E+01  0.2746536E-03  0.6327915E+00
   10      0.8300000E+01  0.2485168E-03  0.5180855E+00
```

Насколько они верны (или неверны) и почему? Графики функции, табулированной алгоритмом **w0(x)** при $\mathbf{x = e^{-t}}$:



1. Письменно сформулировать свое отношение к полученному результату. (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **w1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **w0(x)** и **w1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

8.27 Табулирование функции $\dfrac{\sin^2(0.5+x)-\sin^2 0.5}{\cos^2(0.5+x)-\cos^2 0.5}$.

В некоторой задаче потребовалась табулировать указанную функцию для $\mathbf{x}=\mathbf{e^{-t}}$ при $\mathbf{t}=\mathbf{15.2(0.1)16.2}$.

Была составлена подпрограмма-функция $\mathbf{w0(x)}$:

```
function w0(x)                                  !    Файл w0.for
w0=(sin(0.5+x)**2-sin(0.5)**2)/(cos(0.5+x)**2-cos(0.5)**2)
end
```

которая вела расчет непосредственно по формуле из заголовка.

Тестирование функций проводилось программой

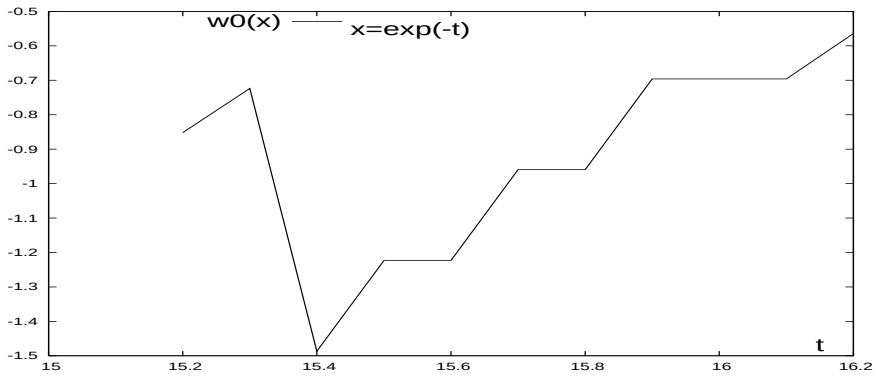
```
      program tsfs2p27                          !    Файл  tsfs2p27.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result', status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' #   t0=',t0,'    tn=',tn,'    n=',n
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=(t0+i*ht)
        x=exp(-t)
        r0=w0(x)
        write(nres,1001) i, t, r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

и при значениях аргумента $\mathbf{t}$ из диапазона $\mathbf{t}=[\mathbf{1.52},\mathbf{1.62}]$ дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t}\in[\mathbf{15.2},\mathbf{16.2}]$ привело к следующему:

```
#   t0=   15.20000         tn=   16.20000         n=          10
#  i            t              w0
   0      0.1520000E+02 -0.9652421E+00
   1      0.1530000E+02 -0.9652421E+00
   2      0.1540000E+02 -0.1087633E+01
   3      0.1550000E+02 -0.1087633E+01
   4      0.1560000E+02 -0.1087633E+01
   5      0.1570000E+02 -0.1087633E+01
   6      0.1580000E+02 -0.9102067E+00
   7      0.1590000E+02 -0.9102067E+00
   8      0.1600000E+02 -0.9102067E+00
   9      0.1610000E+02 -0.9102067E+00
  10      0.1620000E+02 -0.9102067E+00
```

Насколько они верны (или неверны) и почему?

Графики функции, табулированной алгоритмом **w0(x)** при $\mathbf{x = e^{-t}}$:



1. Письменно сформулировать свое отношение к полученному результату. (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчета.
4. Написать соответствующую новой схеме расчета функцию **w1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **w0(x)** и **w1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

8.28 ** Расчет постоянной Эйлера $\gamma = 0.57721566490$

Некто с целью проверить значение постоянной Эйлера, приводимое в справочниках (см., например, [30][стр.14]), составил подпрограмму-функцию **cs(x)**:

```
      function cs(n)                                  !    Файл cs.for
      s=0.0;           ! Функция cs(n) вычисляет приближение к постоянной
      do k=1,n         ! Эйлера через разность между n-ой частичной
        s=s+1.0/k      ! суммой гармонического ряда и ln(n).
      enddo
      cs=s-alog(float(n))
      end
```

которая вела расчет по формуле (см., например, [30][6.1.3])

$$\gamma(\mathbf{n}) = \sum_{\mathbf{k=1}}^{\mathbf{n}} \frac{\mathbf{1}}{\mathbf{k}} - \ln \mathbf{n}$$

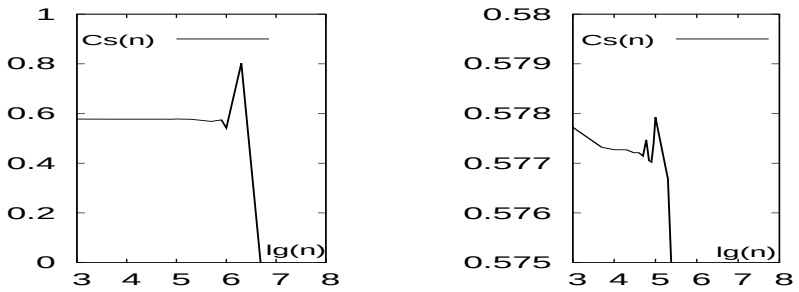
Пользователь полагал, что увеличивая **n** число слагаемых в ней, сможет получить сколько угодно верных значащих цифр постоянной. Тестирование функций проводилось программой

```
      program tsfs2p28                        !    Файл  tsfs2p28.for
      data ninp / 5 /, nres / 6 /
      dimension nn(20)
      data nn /    100,     1000,      5000,   10000,   20000,
     >      30000,   40000,    50000,     60000,   70000,   80000,
     >      90000,  100000,   200000,    500000,  800000, 1000000,
     >    2000000, 5000000, 10000000/
      data gamma /0.5772 15664 90153 28606 06512/
      open(unit=ninp, file='input')
      open(unit=nres, file='result')
      read(ninp,101)  n0, n1
      write(nres, *) ' #   n0=',n0,'    n1=',n1
      write(nres,1100)
      do i=n0,n1
        n=nn(i)
        r0=cs(n)
        write(nres,1001) i, n, r0
      enddo
      close(nres)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'n',14x,'r0')
 1001 format(1x,i5,2x,2x,i15,e15.7)
      end
```

которая вводила из файла номера элементов массива, хранящего набор испытываемых значений **n**. Результат оказался следующим:

```
#   n0=           2     n1=          20
#  i            n              r0
   2             1000  0.5777230E+00
   3             5000  0.5773211E+00
   4            10000  0.5772724E+00
   5            20000  0.5772696E+00
   6            30000  0.5772114E+00
   7            40000  0.5772114E+00
   8            50000  0.5771437E+00
   9            60000  0.5774698E+00
  10            70000  0.5770512E+00
  11            80000  0.5770226E+00
  12            90000  0.5773859E+00
  13           100000  0.5779257E+00
  14           200000  0.5766840E+00
  15           500000  0.5683289E+00
  16           800000  0.5742559E+00
  17          1000000  0.5418472E+00
  18          2000000  0.8023748E+00
  19          5000000 -0.2126598E-01
  20         10000000 -0.7144127E+00
```

Графики функции, табулированной алгоритмом **cs(n)**:



1. Письменно сформулировать свое отношение к полученному результату. (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **cs1(n)**.
5. Обеспечить наглядный вывод результатов в одну таблицу.
6. Включить в **make**-файл правило для вывода рисунка с графиками **cs(n)** и **cs1(n)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

8.29 Проверка формулы $\dfrac{a^3-b^3}{a-b} = a^2 + a \cdot b + b^2$

Школьник решил проверить на компьютере численную правильность аналитически верной алгебраической формулы из заголовка, оформив расчёт её левой части функцией **ab0(a,b)** и выведя абсолютную и относительную погрешность получаемого ей результата (в качестве точного значения брался результат правой части).

```
c                                                    файл ab.for
      function ab0(a,b)
      a3=a*a*a
      b3=b*b*b
      d=a-b
      ab0 =(a3-b3)/d
      end
```

Программа вычисляла **a** по формуле $\mathbf{a} = \mathbf{b} + \mathbf{x}$, где в качестве приращения **x** брались точки равномерного дробления промежутка $[\mathbf{x0}, \mathbf{x1}]$. Количество шагов дробления (**n**), абсциссы его концевых точек (**x0** и **x1**) и значение **b** должны вводиться из файла.

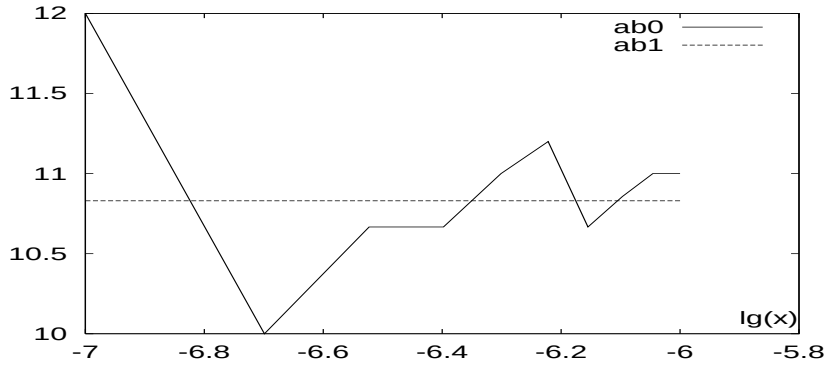
```
      program tsfs2p29
      data ninp / 5 /, data nres / 6 /
      open(unit=ninp,file='input')
      open(unit=nres,file='result')
      read(ninp,100) n
      read(ninp,101) x0, x1, b
      write(nres,1000) b
      h=(x1-x0)/n
      do i=1,n
        x=x0+(i-1)*h;  a=b+x
        r0=ab0(a,b)
        r1=a*a+a*b+b*b
aer=abs(r0-r1)
rer=abs(aer/r1)
        write(nres,1101) x, r0, r1, aer, rer
      enddo
      close(nres)
  100 format(i10)
  101 format(e10.3)
 1000 format(1x,'# b=',e15.7,5x,' a=b+x'/1x,'#',71('-')/
     > 1x,'#',8x,'x',9x,'(a3-b3)/(a-b)',3x,'a^2+ab+b^2)/x',
     >    6x,'aer',9x,'rer'/' #',71('-'))
 1101 format(1x,e16.7,e16.7,e16.7, e12.3, e12.3)
      end
```

Обнаружилось, что при $\mathbf{b} = \mathbf{1.7}$ и $\mathbf{x} \in [\mathbf{10^{-7}}, \mathbf{10^{-6}}]$ результаты, полученные по формулам левой и правой частей, которые аналитически эквивалентны друг другу, далеко не одинаковы. В то же время при $\mathbf{x} \in [\mathbf{2}, \mathbf{3}]$ результаты расчета правой левой частей совпадали: Помогите школьнику понять получаемое:

```
# b=  0.1900000E+01      a=b+x
#-------------------------------------------------------------------------
#        x          (a3-b3)/(a-b)   a^2+ab+b^2)/x      aer          rer
#-------------------------------------------------------------------------
   0.1000000E-06   0.1200000E+02   0.1083000E+02   0.117E+01   0.108E+00
   0.2000000E-06   0.1000000E+02   0.1083000E+02   0.830E+00   0.766E-01
   0.3000000E-06   0.1066667E+02   0.1083000E+02   0.163E+00   0.151E-01
   0.4000000E-06   0.1066667E+02   0.1083000E+02   0.163E+00   0.151E-01
   0.5000000E-06   0.1100000E+02   0.1083000E+02   0.170E+00   0.157E-01
   0.6000000E-06   0.1120000E+02   0.1083000E+02   0.370E+00   0.342E-01
   0.7000000E-06   0.1066667E+02   0.1083000E+02   0.163E+00   0.151E-01
   0.8000000E-06   0.1085714E+02   0.1083000E+02   0.271E-01   0.251E-02
   0.9000000E-06   0.1100000E+02   0.1083000E+02   0.170E+00   0.157E-01
   0.1000000E-05   0.1100000E+02   0.1083000E+02   0.170E+00   0.157E-01
```

Насколько они верны (или неверны) и почему?

1. Письменно сформулировать свое отношение к полученному результату.
2. Объяснить объективную и субъективную причины его появления.
3. Преобразовать расчетную формулу так, чтобы результат был верен.
4. Написать соответствующую преобразованной формуле функцию**ab1(a,b)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу вместе с погрешностями исходной формулы.
6. Включить в **make**-файл правило для вывода на один рисунок двух графиков, соответствующих таблице.
7. Модифицировать программу для получения результатов типа **real(mp)**.



8.30 Табулирование функции $\dfrac{\cos x - 1 + \dfrac{x^2}{2} - \dfrac{x^4}{24} + \dfrac{x^6}{720}}{\dfrac{\sin x}{x} - 1 + \dfrac{x^2}{6} - \dfrac{x^4}{120} + \dfrac{x^6}{5040}}$.

В некоторой задаче потребовалась табулировать указанную функцию для $\mathbf{x} = \mathbf{e}^{-\mathbf{t}}$ при $\mathbf{t} = \mathbf{15.2}(\mathbf{0.1})\mathbf{16.2}$.

Была составлена подпрограмма-функция $\mathbf{w0}(\mathbf{x})$:

```
function w0(x)                                    !    Файл w0.for
x2=x*x
a=cos(x) - 1 + x2* 0.5*(1 - x2/12 * ( 1 - x2 / 30))
b=sin(x)/x - 1+x2/6 * (1-x2/20*(1-x2/42))
w0=a/b
end
```

которая вела расчет непосредственно по формуле из заголовка.

Тестирование функций проводилось программой

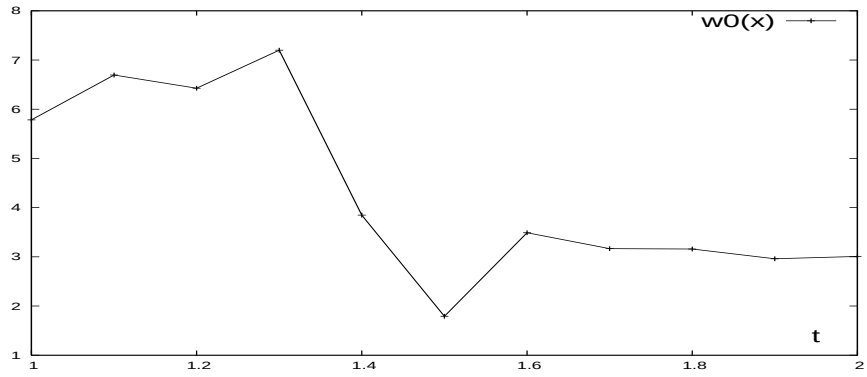
```
      program tsfs2p30                        !    Файл  tsfs2p30.for
      data ninp / 5 /, nres / 6 /
      open(unit=ninp, file='input')
      open(unit=nres, file='result', status='replace')
      read(ninp,100)  t0, tn
      read(ninp,101) n
      write(nres, *) ' # t0=',t0,'  tn=',tn,'  n=',n
      pi4=atan(1.0)
      ht=(tn-t0)/n
      write(nres,1100)
      do i=0,n
        t=(t0+i*ht)
        x=exp(-t)
        r0=w0(x)
        write(nres,1001) i, t, r0
      enddo
      close(nres)
  100 format(e15.7)
  101 format(i15)
 1100 format(1x,' #',2x,'i',12x,'t',14x,'w0')
 1001 format(1x,i5,2x,2x,e15.7,e15.7)
      end
```

и при значениях аргумента $\mathbf{t}$ из диапазона $\mathbf{t} = [-\mathbf{1.0}, -\mathbf{2.0}]$ (то есть при $\mathbf{x} = \mathbf{e}$ и $\mathbf{x} = \mathbf{e^2}$) дало на одинарной точности результаты верные в пределах семи значащих цифр мантиссы. Однако, тестирование функции на требуемом рабочем диапазоне $\mathbf{t} \in [\mathbf{15.2}, \mathbf{16.2}]$ привело к следующему:

```
# t0=   1.000000       tn=   2.000000       n=          10
#  i            t              w0
   0      0.1000000E+01  0.5782932E+01
   1      0.1100000E+01  0.6694451E+01
   2      0.1200000E+01  0.6426376E+01
   3      0.1300000E+01  0.7200233E+01
   4      0.1400000E+01  0.3847253E+01
   5      0.1500000E+01  0.1791706E+01
   6      0.1600000E+01  0.3489712E+01
   7      0.1700000E+01  0.3166201E+01
   8      0.1800000E+01  0.3158754E+01
   9      0.1900000E+01  0.2960747E+01
  10      0.2000000E+01  0.3008335E+01
```

Насколько они верны (или неверны) и почему?

Графики функции, табулированной алгоритмом **w0(x)** при $\mathbf{x} = \mathbf{e^{-t}}$ (по оси абсцисс отложено t):



1. Письменно сформулировать свое отношение к полученному результату (для упрощения процесса оценки используем систему **maxima**).
2. Объяснить объективную и субъективную причины его появления.
3. Привести исходную формулу к виду удобному для расчёта.
4. Написать соответствующую новой схеме расчета функцию **w1(x)**.
5. Обеспечить наглядный вывод результатов расчета по обеим формулам в одну таблицу.
6. Включить в **make**-файл правило вывода рисунка с графиками **w0(x)** и **w1(x)**.
7. Модифицировать программу для получения результатов типа **real(mp)** и проанализировать их.

Список литературы

[1] Браун С. Операционная система UNIX: Пер. с англ. – М.: Мир, 1986.–463 с.

[2] Керниган Б.В., Пайк Р. 1992. UNIX – универсальная среда программирования: Пер. с англ.; Предисл. М.И. Белякова. – М.: Финансы и статистика, – 304 с.

[3] Горелик А.М. 2006. Программирование на современном Фортране. – М.: Финансы и статистика, – 352 с.

[4] Стен Келли-Бутл 1995. Введение в UNIX: Пер. с англ. С. Орлова; Издательство “ЛОРИ”, 596 с.

[5] Кэвин Рейчард, Эрик Форстер Джонсон 1999. UNIX–справочник – СПб: Питер Ком, – 384 с.

[6] . 1995. Светозарова Г.И., Козловский А.В., Сигитов Т.В. Современные методы программирования в примерах и задачах.–М: Наука. Физматлит, –427 с.

[7] Бартеньев О.В. 1999. Фортран для студентов. – М.:"ДИАЛОГ – МИФИ – 400 с.

[8] Бартеньев О.В. 1999. Visual Fortran. Новые возможности. Издательство "ДИАЛОГ – МИФИ – ??? с.

[9] Бартеньев О.В. 2000. Современный ФОРТРАН. "3-е изд., доп. и перераб. – М.; ДИАЛОГ – МИФИ, – 448 с.

[10] Рыжиков Ю.И. 2004. Современный ФОРТРАН: Учебник. – СПб.: КОРОНА принт, –288 с.

[11] Немнюгин М.А., Стесик О.Л. 2004. Современный ФОРТРАН: Самоучитель. – СПб.: БХВ-Петербург, 496 с.

[12] Немнюгин М.А., Стесик О.Л. 2002. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ-Петербург, 400 с.

[13] Немнюгин М.А., Стесик О.Л. 2008. ФОРТРАН в задачах и примерах многопроцессорных вычислительных систем. – СПб.: БХВ-Петербург, 320 с.

[14] Романовская Л.М., Русс Т.В., Свитковский С.Г. Программирование в среде Си для ПЭВМ ЕС. – М.: Финансы и статистика, 1991, – 352 с.:ил.

[15] Касаткин А.И., Вальвачев А.Н. 1992. Профессиональное программирование на языке СИ: От Turbo C к Borland C++: Справ. пособие; Под общ. ред. А.И. Касаткина. – Мн.: Выш.шк., – 240 с.

[16] Шилдт Г. 2002. - Самоучитель C++: Пер. с англ. – 3-е изд. – СПб.: БХВ–Петербург, – 688 с.

[17] Гриффитс А. 2004. gcc. Настольная книга пользователей, программистов и системных администраторов. Пер. с англ./Артур Гриффитс. – К.: "ТИД"ДС", –624 с.

[18] Клюшин Д.А. 2004. Полный курс C++. Професстональная работа. – М.: Издательский дом "Вильямс" – 624 с. : ил.

[19] Кубенский А.А. 2004. Структуры и алгоритмы обработки данных: объектно ориентированный подход и реализация на C++. – СПб.: БХВ-Петербург, 464 с.

[20] Игнатов В. 2000. Эффективное использование GNU Make.

[21] Richard M.Stallman, Roand McGrath GNU Make Программа управления компиляцией. GNU make Версия 3.79 Апрель 2000. перевод (C) Владимира Игнатова $http://linux.yaroslavl.ru/docs/prog/gnu\_make\_3-79\_russian\_manual.html$

[22] Левин М. 2005. СИ++: Самоучитель / Максим Левин. – М.: ЗАО "Новый издательский дом", – 176с.

[23] Кристиан. 1985. Введение в ОС UNIX – М. ???? с.

[24] Бартеньев О.В. 1999. IMSL???Фортран для студентов. – М.:"ДИАЛОГ – МИФИ – 400 с.

[25] Люк ???2004. Полный курс C++. Профессиональная работа. – М.: Издательский дом "Вильямс", –672 с.: ил.

[26] Липский В. 1988. - Комбинаторика для программистов: Пер. с польск. – М.: Мир,– 213.

[27] Д.Кнут XXXX. - Искусство программирования для ЭВМ т. 1. Основные алгоритмы.

[28] Цветков А.С. 2005 Руководство по практической работе с каталогом Hipparcos: Учебно-метод. пособие. СПб., 2005. –104 с.

[29] Петрова В.А., Клюев В.В. 1996. - Редакторы и коммуникационные средства операционной системы UNIX. Методические указания. Санкт–Петербурский государственный университет.

[30] Справочник по специальным функциям с формулами, графиками и таблицами. Под редакцией М. Абрамовица и И. Стиган. 1979. Москва "Наука" Главная редакция физико-математической литературы. Перевод с английского под редакцией В.А. Диткина и Л.Н. Кармазиной. М., 832 стр. с илл.

[31] Numerical recipes in FORTRAN–77: The art of scientific computing (ISBN 0-521-43064-X) Copyright(C) 1986–1992 by Cambridge University Press.

[32] Numerical recipes in C: The art of scientific computing (ISBN 0-521-43108-5) Copyright(C) 1988–1992 by Cambridge University Press.

[33] Вирт Н., Алгоритмы + структуры данных = программы: Пер. с англ. – М.: Москва, Мир, 1985. – 406 с., ил.

[34] Дмитриева М.В. Кубенский А.А. Элементы современного программирования: Учеб. пособие/Под ред. С.С. Лаврова. – СПб.: Издательство С.–Петербургского университета, 1991. – 272 с.

[35] 1994. Лабораторный практикум по высшей математике: Учеб. пособие для втузов.–2-е изд., перераб. и доп.–М.: Высш. шк., –416 с.

[36] Крылов В.И., Шульгина Л.Т. Справочная книга по численному интегрированию, 1966. Издательство “Наука”, Гл. ред. физ.–мат. лит., – 372 с. –

[37] Крылов В.И. Приближенное вычисление интегралов, 1967. Издательство “Наука”, Гл. ред. физ.–мат. лит., – 500 с. –

[38] Пономаренко А.К. Алгол-процедуры (сборник), выпуск 12, с. 6