[15. Классы памяти.](#_Toc515770267)

[15.1. «Зоопарк» классов памяти.](#_Toc515770268)

[15.1.1. Автоматические переменные.](#_Toc515770269)

[15.1.2. Регистровые переменные.](#_Toc515770270)

[15.1.3. Статические переменные.](#_Toc515770271)

[15.1.4. Внешние переменные.](#_Toc515770272)

[15.1.5. Внешние статические переменные.](#_Toc515770273)

[15.2. Объявление переменных на внутреннем уровне.](#_Toc515770274)

[15.3. Объявление переменных на внешнем уровне.](#_Toc515770275)

[15.4. Переменные класса volatile](#_Toc515770276)

[15.5. Ключевое слово mutable](#_Toc515770277)

[15.6. Классы памяти и область действия.](#_Toc515770278)

[15.6.1. Правила области действия переменной.](#_Toc515770279)

[15.6.2. Операция уточнения области действия в C++.](#_Toc515770280)

[15.6.3. Выбор класса памяти.](#_Toc515770281)

[15.7. Пространства имен](#_Toc515770282)

[15.8. Функции и классы памяти.](#_Toc515770283)

[15.8.1. Объявления функций на внешнем уровне.](#_Toc515770284)

[15.9. Функция получения случайных чисел.](#_Toc515770285)

[15.10. Игра в кости.](#_Toc515770286)

[15.11. Функция получения целых чисел: getint().](#_Toc515770287)

[15.12. Сортировка.](#_Toc515770288)

1. Классы памяти.

Одно из достоинств языка Си состоит в том, что он позволяет управлять ключевыми механизмами программы. Классы памяти языка Си — пример такого управления; они дают возможность определить, с какими функциями связаны какие переменные и как долго переменная сохраняется в программе.

Ранее уже упоминалось о таком важном свойстве переменной, как время жизни. Существует четыре модификатора переменных, определяющих область видимости и время действия переменных. Все они приводятся в таблице 4.1.

**Таблица 4.1. Модификаторы переменных**

|  |  |  |  |
| --- | --- | --- | --- |
| **Модификатор** | **Применение** | **Область действия** | **Время жизни** |
| auto | локальное | Блок | временное |
| register | локальное | Блок | временное |
| extern | глобальное | Блок | временное |
| static | локальное | Блок | постоянное |
| глобальное | Файл |
| volatile | глобальное | Файл | постоянное |

Ниже будет рассмотрен более подробно каждый из приведенных классов памяти.

Программирование, точно так же как написание романа (или даже письма),— это не просто знание языковых правил — это нечто большее. Здесь мы рассмотрим несколько полезных функций. При этом попытаемся привести некоторые соображения, используемые при конструировании функций. В частности, сделаем упор на значение модульного подхода, разбивающего программы на выполнимые задачи. Сначала, однако, обсудим классы памяти.

В C/C++ имеется четыре наиболее распространенных спецификатора классов памяти:

auto

register

static

extern

Класс памяти предшествует объявлению переменной и указывает компилятору способ хранения этой переменной. Объекты, объявленные со спецификаторами auto или register, имеют локальное время жизни, а спецификаторы static или extern указывают на глобальное время жизни.

Эти четыре спецификатора класса памяти кроме класса памяти определяют видимость переменной или функции. Область видимости (иногда определяемая как область действия) относится к той части исходной программы, в которой к переменной или функции можно обращаться по имени. Объект с глобальным временем жизни существует на протяжении всего времени выполнения исходной программы.

Место, где располагается объявление переменной или функции в исходном файле, также влияет на класс памяти и видимость. Объявления, находящиеся вне всех описаний функций, считаются расположенными на внешнем уровне, а объявления, встречающиеся внутри описаний функций, относятся ко внутреннему уровню.

Точный смысл каждого спецификатора класса памяти зависит от двух факторов: от того, на каком уровне расположено объявление — на внешнем или на внутреннем; и от типа объекта — переменная это или функция.

## «Зоопарк» классов памяти.

### Автоматические переменные.

Модификатор auto используется при описании локальных переменных. Поскольку для локальных переменных данный модификатор используется по умолчанию, на практике его чаще всего опускают.

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

auto int MyVar = 2;

//то же что int MyVar = 2;

cout << MyVar;

getchar(); getchar();

return 0;

}



Модификатор auto применяется только к локальным переменным, которые видны только в блоке, в котором они объявлены. При выходе из блока такие переменные уничтожаются автоматически.

По умолчанию переменные, описанные внутри функции, являются автоматическими. Можно, однако, это подчеркнуть явно с помощью ключевого слова auto:

main()

{

auto int plox;

}

Так поступают, если хотят, например, показать, что определение переменной не нужно искать вне функции.

Автоматические переменные имеют локальную область действия. Только функция, в которой переменная определена, «знает» ее. (Конечно, можно использовать аргументы для связи значения и адреса переменной с другой функцией, однако это частичное и косвенное «знание».) Другие функции могут использовать переменные с тем же самым именем, но это должны быть независимые переменные, находящиеся в разных ячейках памяти.

Автоматическая переменная начинает существовать при вызове Функции, содержащей ее. Когда функция завершает свою работу и возвращает управление туда, откуда ее вызвали, автоматическая переменная исчезает. Ячейка памяти может снова использоваться для чего-нибудь другого.

Следует еще сказать об области действия автоматической переменной: область действия ограничена блоком ({}), в котором переменная описана. Мы всегда должны описывать наши переменные в начале тела функции (блока), так что областью действия их является вся функция. Однако в принципе можно было бы описать переменную внутри подблока. Тогда переменная будет известна только в этой части функции. Обычно при создании программы, программисты редко принимают во внимание упомянутое свойство. Но иногда торопливые программисты пользуются такой возможностью, особенно когда пытаются быстрее внести коррективы.

### Регистровые переменные.

Модификатор register предписывает компилятору попытаться разместить указанную переменную в регистрах процессора. Если такая попытка оканчивается неудачно, переменная ведет себя как локальная переменная типа auto. Размещение переменных в регистрах, оптимизирует программный код по скорости, так как процессор оперирует с переменными, находящимися в регистрах, гораздо быстрее, чем с памятью. Но в связи с тем, что число регистров процессора ограничено, количество таких переменных может быть очень небольшим.

#include <iostream.h>

int main ()

{

register int REG;

…

return 0;

}

Модификатор register применяют только к локальным переменным. Попытка употребления данного модификатора (так же как и модификатора auto) к глобальным переменным вызовет сообщение об ошибке. Переменная существует только в пределах блока, содержащего ее объявление.

Обычно переменные хранятся в памяти машины. К счастью, регистровые переменные запоминаются в регистрах центрального процессора, где доступ к ним и работа с ними выполняются гораздо быстрее, чем в памяти. В остальном регистровые переменные аналогичны автоматическим переменным. Они создаются следующим образом:

main()

{

register int quick;

…

}

Мы сказали «к счастью», потому что описание переменной как регистровой, является скорее просьбой, чем обычным делом. Компилятор должен сравнить ваши требования с количеством доступных регистров, поэтому вы можете и не получить то, что хотите. В этом случае переменная становится простой автоматической переменной.

#include <iostream> //cout

#include <conio.h> //\_getch();

using namespace std;

int int\_pwr(register int m, register int e);

int main()

{

int io;

io=int\_pwr(2,30);//возведение 2 в степень

cout << "2 is equal to 30 degrees " << io;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int int\_pwr(register int m, register int e)

{

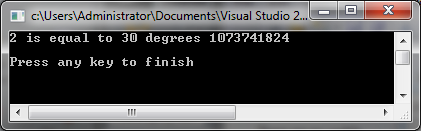
register int temp;

temp = 1;

for(; e; e--) temp = temp \* m;

return temp;

}



### Статические переменные.

Статические переменные во многом похожи на глобальные переменные. Для описания статических переменных используется модификатор **static**. Если такая переменная объявлена глобально, то она инициализируется при запуске программы, а ее область видимости совпадает с областью действия и простирается от точки объявления до конца файла. Если же статическая переменная объявлена внутри функции или блока, то она инициализируется при первом входе в соответствующую функцию или блок. Значение переменной сохраняется от одного вызова функции до другого. Таким образом, статические переменные можно использовать для хранения значений переменных на протяжении времени работы программы.

**Замечание.**

1) Статические переменные не могут быть объявлены в других файлах как внешние.

2) Если статическая переменная не проинициализирована явным образом, то, как и глобальная переменная, она инициализируется значением 0.

В качестве примера рассмотрим реализацию счетчика вызовов некоторой функции:

#include <iostream>

using namespace std;

int Count(void);

int main()

{

int result;

for(int i= 0; i< 30; i++)

result = Count();

cout << result;

getchar(); getchar();

return 0;

}

int Count (void)

{

static short counter = 0;

counter++;

//...

return counter;

}



Здесь главная функция в цикле (30 paз подряд) вызывает функцию Count (), которая содержит статическую переменную counter. Как видно из примера, начальная инициализация этой переменной нулем выполнится только один раз, при первом вхождении в тело функции. Поскольку значение переменной сохраняется между вызовами функции, на печать будет выведено число 30.

Название раздела не следует понимать буквально, т. е. считать, что такие переменные не могут изменяться. В действительности слово «статические» здесь означает, что переменные остаются в работе. Они имеют такую же область действия, как автоматические переменные, но они не исчезают, когда содержащая их функция закончит свою работу. Компилятор хранит их значения от одного вызова функции до другого. Следующий пример иллюстрирует это и показывает, как описать статическую переменную.

#include<stdio.h>

trystat()

{

int fade = 1;

static int stay = 1;

printf(" fade = %d and stay = %d\n", fade++, stay++);

}

main()

{

int count;

for (count = 1; count <= 3; count++)

{

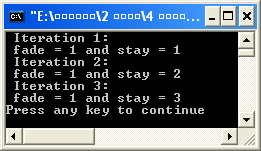
printf("Iteration %d:\n" , count);

trystat();

}

}

Результаты работы программы



Заметим, что функция trystat() увеличивает каждую переменную после печати ее значения.

Статическая переменная stay «помнит», что ее значение было увеличено на 1, в то время как для переменной fade начальное значение устанавливается каждый раз заново. Это указывает на разницу в инициализации: fade инициализируется каждый раз, когда вызывается trystat(), в то время как stay инициализируется только один раз при компиляции функции trystat().

### Внешние переменные.

Эта программа состоит из нескольких модулей, некоторые переменные могут использоваться для передачи значений из одного файла в другой. При этом некоторая переменная объявляется глобальной в одном модуле, а в других файлах, в которых она должна быть видима, производится ее объявление с использованием модификатора **extern**. Если объявление внешней переменной производится в блоке, она является локальной.

В отличие от предыдущих, этот модификатор сообщает, что первоначальное объявление переменной производится в каком-то другом файле. Рассмотрим пример использования внешней переменной:

// файл myheader.h

void ChangeFlag(void);

// файл myfunction.cpp

extern bool Flag;

void ChangeFlag(void)

{

Flag = !Flag;

}

#include "myheader.h"

#include <iostream>

using namespace std;

extern bool Flag;

int main()

{

setlocale(LC\_ALL, "Rus");

ChangeFlag();

if(Flag)

cout << "Сейчас TRUE\n";

else

cout << "Сейчас FALSE\n";

getchar(); getchar();

return 0;

}



Сначала в файле myheader.h объявляется функция ChangeFlag(). Далее в файле myfunction.cpp следует объявление глобальной логической переменной Flag и определяется реализация тела функции ChangeFlag() и наконец в главном модуле подключается заголовочный файл myheader.h и переменная Flag описывается как внешняя (extern). Поскольку описание функции ChangeFlag() включается в главный модуль директивой #include "myheader.h", данная функция доступна в теле функции main().

Переменная, описанная вне функции, является внешней. Внешнюю переменную можно также описать в функции, которая использует ее, при помощи ключевого слова extern. Описания могут выглядеть примерно так:

int errupt; /\* Три переменные, описанные вне функции \*/

char coal;

double up;

main()

{

extern int errupt; /\* объявлено, что З переменные \*/

extern char coal; /\* являются внешними \*/

extern double up;

Группу extern-описаний можно совсем опустить, если исходные определения переменных появляются в том же файле и перед функцией, которая их использует. Включение ключевого слова extern позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом файле. (Оба файла, конечно, должны быть скомпилированы, связаны или собраны в одно и то же время.)

Если слово extern не включено в описание внутри функции, то под этим именем создается новая автоматическая переменная. Вы можете пометить вторую переменную как «автоматическую» с помощью слова auto и тем самым показать, что это ваше намерение, а не оплошность.

Три примера демонстрируют четыре возможных комбинации описаний:

/\* Пример1 \*/

int hocus;

main()

{

extern int hocus; /\* hocus описана внешней \*/

...

}

magic()

{

extern int hocus;

...

}

Здесь есть одна внешняя переменная hocus, и она известна обеим функциям main() и magic().

/\* Пример2 \*/

int hocus;

main()

{

extern int hocus; /\* hocus описана внешней \*/

…

}

magic()

{

/\* hocus не описана совсем \*/

...

}

Снова есть одна внешняя переменная hocus, известная обеим функциям. На этот раз она известна функции magic() по умолчанию.

/\* ПримерЗ \*/

int hocus;

main()

{

int hocus; /\* hocus описана и является автоматической по умолчанию \*/

...

}

magic()

{

auto int hocus; /\* hocus описана автоматической \*/

...

}

В этом примере созданы три разные переменные с одинаковым именем. Переменная hocus в функции main() является автоматической по умолчанию и локальной для main(), в функции magic() она явно описана автоматической и известна только для magic(). Внешняя переменная hocus не известна ни main(), ни magic(), но обычно известна любой другой функции в файле, которая не имеет своей собственной локальной переменной hocus.

Эти примеры иллюстрируют область действия внешних переменных. Они существуют, пока работает программа, и так как эти переменные доступны любой функции, они не исчезнут, если какая-нибудь одна функция закончит свою работу.

### Внешние статические переменные.

Вы можете также описать статические переменные вне любой функции. Это создаст «внешнюю статическую» переменную. Разница между внешней переменной и внешней статической переменной заключается в области их действия. Обычная внешняя переменная может использоваться функциями в любом файле, в то время как внешняя статическая переменная может использоваться только функциями того же самого файла, причем после определения переменной. Вы описываете внешнюю статическую переменную, располагая ее определение вне любой функции.

static randx = 1;

rand()

{

…

…

}

## Объявление переменных на внутреннем уровне.

Любой из четырех спецификаторов класса памяти можно использовать для объявления переменных на внутреннем уровне. (По умолчанию используется **auto**.) Спецификатор **auto** объявляет переменную с локальным временем жизни. Она видна только в том блоке, в котором описана, и может иметь инициирующее значение.

Спецификатор класса памяти register указывает компилятору на необходимость размещения переменной в регистре (если это возможно). При использовании этого спецификатора увеличивается скорость доступа и уменьшается размер кода программы. Видимость переменной такая же, как при использовании спецификатора **auto**. Если компилятор встречает объявление **register**, но отсутствуют доступные регистры, переменной назначается класс памяти **auto** и она запоминается в оперативной памяти.

В ANSI С нельзя использовать адрес объекта типа **register**. Это ограничение, однако, не распространяется на C++. Если используется операция определения адреса (&) вместе с регистровой переменной в C++, компилятор помещает ее в оперативной памяти, так как он должен сохранить переменную в таком месте, которое имеет действительный адрес.

Переменная, объявленная на внутреннем уровне со спецификатором класса памяти **static**, имеет глобальное время жизни, но видима только в том блоке, в котором она объявлена. В отличие от переменных **auto**, переменные **static** сохраняют свои значения при выходе из блока. Можно инициализировать переменную **static** при помощи констант. По умолчанию начальное значение — ноль.

Переменная, объявленная со спецификатором класса памяти **extern**, является ссылкой на переменную с таким же именем, описанную на внешнем уровне в любом исходном файле программы. Внутреннее объявление **extern** используется для того, чтобы сделать видимым внутри блока описание переменной, сделанное на внешнем уровне. Это демонстрируется в следующей программе:

#include<stdio.h>

#include<iostream.h>

int ivalue1=1;

void function\_a(void)

{

// запоминается адрес глобальной переменной ivalue1;

static int \*pivalue1= &ivalue1;

// создается новая локальная переменная ivalue1; при этом глобальная

// переменная ivalue1 становится недоступной

int ivalue1 = 32;

// новая локальная переменная ivalue2

// видима только внутри функции function\_a

static int ivalue2 = 2;

ivalue2 += 2;

// печатаются числа 32, 4 и 1:

cout << " "<<ivalue1<<" " << ivalue2<< " "<< \*pivalue1;

}

main ()

{

// обращается к переменной ivalue1, описанной выше

extern int ivalue1;

//по умолчанию — начальное значение = 0, переменная ivalue2

// видима только в main()

static int ivalue2;

// запоминается в регистре (по возможности),

// начальное значение = 0

register int rvalue = 0;

//по умолчанию - класс памяти auto, переменная int\_value3

//получает начальное значение 0

int int\_value3 = 0;

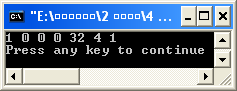
// печатаются числа 1, 0, 0, 0:

cout << ivalue1 <<" "<< rvalue <<" "<<ivalue2 <<" "<< int\_value3;

function\_a() ;

}

Результаты работы программы:



Поскольку ivalue1 переопределяется в function\_a(), доступ к глобальной переменной ivalue1 запрещен. Однако, при помощи указателя pivalue1 получен адрес глобальной переменной ivalue1 для печати хранящегося в ней значения.

## Объявление переменных на внешнем уровне.

В объявлении переменных на внешнем уровне можно использовать только класс памяти static или extern (auto или register использовать нельзя). Это могут быть либо описания переменных, либо ссылки на переменные, описанные в другом месте. Внешнее объявление переменной, инициализирующее переменную (неявно или явно), является объявлением-описанием:

static int ivalue1; // по умолчанию подразумевается 0

static int ivalue1 = 10 // явная инициализация

int ivalue2 =20; // явная инициализация

Если переменная описывается на внешнем уровне, она видима в оставшейся части того исходного файла, в котором она объявлена, и не видима в этом файле до точки ее описания. Кроме того, она не видима в других исходных файлах программы, если ее видимость не обеспечена ссылочным объявлением.

Внутри исходного файла можно только один раз описать переменную на внешнем уровне. Если использовать спецификатор класса памяти static, в другом исходном файле можно описать еще одну переменную с таким же именем и спецификатором static. Поскольку каждое статическое описание видимо только в соответствующем исходном файле, конфликтов не будет. При помощи спецификатора extern объявляется ссылка на переменную, которая описана в другом месте. Объявление external можно использовать для обеспечения видимости описания из другого файла или для того, чтобы расширить видимость переменной в том же самом файле. Переменная видима на протяжении оставшейся части того исходного файла, в котором находится объявленная ссылка.

Для того чтобы внешняя ссылка была действительной, связанная с ней переменная должна быть объявлена только один раз на внешнем уровне. Описание может располагаться в любом исходном файле программы. Следующая программа на C++ иллюстрирует использование ключевого слова extern:

//Исходный файл А

#include <iostream> //cout

#include <conio.h> //\_getch();

using namespace std;

void function\_a(void);

void function\_b(void);

int ivalue = 10; // действительное описание переменной

// ivalue

int main()

{

ivalue++; // используется объявление extern, приведенное выше

cout << ivalue << "\n"; // печатается 11

function\_a();

\_getch();

return(0);

}

extern int ivalue; // переменная ivalue определяется видимой

void function\_a(void)

{

ivalue++; // обращение к ivalue

cout << ivalue << "\n"; // печатается 12

function\_b();

}

//

//Исходный файл В

//

#include <iostream> //cout

using namespace std;

extern int ivalue; // ссылка на переменную ivalue,

// объявленную в Исходном файле А

void function\_b(void)

{

ivalue++;

cout << ivalue; // печатается 13

}

## Переменные класса volatile

В тех случаях, когда необходимо предусмотреть возможность модификации переменной периферийным устройством или другой программой, используют модификатор volatile. В связи с этим компилятор не пытается оптимизировать программу путем размещения переменной в регистрах.

Пример объявления таких переменных приведен ниже:

volatile short sTest;

volatile const int vciTest;

Как видно из примера, переменная vciTest с модификатором volatile в то же время может быть объявлена как константа. В этом случае ее значение не сможет меняться в разрабатываемой программе, но может модифицироваться в зависимости от внешних факторов.

Указывая **volatile** при объявлении переменной, программист просит компилятор не оптимизировать эту переменную.

Компилятор скорее всего оптимизирует код вроде такого, если переменная cancel не менялась в теле цикла.

bool **cancel =** false;

while( !cancel ) {

**;**

}

Если **cancel** не меняется, то ее и проверять каждый раз незачем, компилятор и не будет ее проверять.

Зато если вы укажете перед переменной **volatile**, то оптимизиации не будет. Предполагается, что переменная **cancel** могла измениться каким-то волшебным образом.

volatilebool **cancel =** false;

while( !cancel ) {

**;**

}

Каким волшебным образом? Из другого потока? Но обеспечивать доступ к переменным из разных потоков с помощью **volatile** не есть хорошо. Потому что переменную надо проверять (лочить) при считывании и записи. Чтобы не получилось, например, такой ситуации: один поток пишет в переменную, другой в этот же момент считывает и считает наполовину старое значение, наполовину новое, короче не пойми что получится.

Чисто теоретически указания **volatile** при работе с потоками достаточно, если тип данных, с которым идет работа, может быть записан на данной архитектуре атомарно, в один прием. Соответственно, надо точно знать к каким именно типам это относится. Казалось бы уж что-что, так bool должен писаться в один прием. Но на некоторых Windows'ах это вовсе даже и не так. И атомарность присутствует только при работе с char...

Несмотря ни на что, **volatile** таки используется для доступа к переменной из разных потоков.

Итак, **volatile** в языке Си - это квалификатор переменной, говорящий компилятору, что значение переменной может быть изменено в любой момент и что часть кода, которая производит над этой переменной какие-то действия (чтение или запись), не должна быть оптимизирована.

Есть три основных типа ошибок, касающихся квалификатора **volatile**:

* **неиспользование volatile там, где нужно**

обычно совершается программистами, которые не знают про существование **volatile**, или видели, но не понимают, что это такое;

* **использование volatile там, где нужно, но не так, как нужно**

присуща программистам, знающим, насколько важен **volatile** при программировании параллельных процессов или при доступе к периферийным регистрам, но не учитывающие некоторые его нюансы;

* **использование volatile там, где не нужно (бывает и такое)**

такое делают те, кто однажды обжегся на первых двух ошибках. Это не ошибка и она не приведет к неправильному поведению программы, но создаст свои неприятности.

**Замечание**.

Полезным для подробностей будет сайт:

1. **http://www.quizful.net/post/cpp-keywords-usage**
2. [**http://we.easyelectronics.ru/Soft/skolzkaya-dorozhka-dlya-poklonnikov-volatile.html**](http://we.easyelectronics.ru/Soft/skolzkaya-dorozhka-dlya-poklonnikov-volatile.html)

**Пример несоответствующего кода**

В этом примере несовместимого кода доступ к **volatile** объекту осуществляется с помощью ссылки, не зависящей от **volatile**, что приводит к неопределенному поведению:

#include<iostream>

usingnamespace **std;**

void **func(**void) {

staticvolatileint **\*\*ipp;**

staticint **\*ip;**

staticvolatileint **i = 0;**

**printf(**"i = %d.\n", i);

**ipp = &ip;** /\* Может выдавать warning диагностику \*/

**ipp = (**int\*\*) &ip; /\* Нарушение ограничений; Может выдавать warning диагностику \*/

**\*ipp = &i;** /\* обоснованный \*/

if **(\*ip != 0) {** /\* обоснованный \*/

/\* ... \*/

**}**

}

//void func(void) {

// static volatile int \*\*ipp;

// static volatile int \*ip;

// static volatile int i = 0;

//

// printf("i = %d.\n", i);

//

// ipp = &ip;

// \*ipp = &i;

// if (\*ip != 0) {

// /\* ... \*/

// }

//}

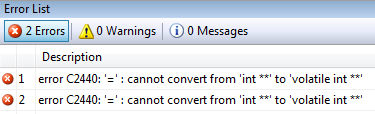
int **main () {**

**func();**

**system(**"PAUSE");

return **0;**

}



Назначение ipp = &ip небезопасно, потому что оно позволяет действительному следующему коду ссылаться на значение изменчивого объекта i через невостребованную ссылку ip. В этом примере компилятор может оптимизировать весь блок if, потому что \*ip! = 0 должен быть ложным, если объект, на который указывает ip, не является изменчивым.

**Удовлетворительное решение**

В этом совместимом решении ip объявляется **volatile**:

#include<iostream>

usingnamespace **std;**

//void func(void) {

// static volatile int \*\*ipp;

// static int \*ip;

// static volatile int i = 0;

//

// printf("i = %d.\n", i);

//

// ipp = &ip; /\* Может выдавать warning диагностику \*/

// ipp = (int\*\*) &ip; /\* Нарушение ограничений; Может выдавать warning диагностику\*/

// \*ipp = &i; /\* обоснованный \*/

// if (\*ip != 0) { /\* обоснованный \*/

// /\* ... \*/

// }

//}

void **func(**void) {

staticvolatileint **\*\*ipp;**

staticvolatileint **\*ip;**

staticvolatileint **i = 0;**

**printf(**"i = %d.\n", i);

**ipp = &ip;**

**\*ipp = &i;**

if **(\*ip != 0) {**

/\* ... \*/

**}**

}

int **main () {**

**func();**

**system(**"PAUSE");

return **0;**

}



## Ключевое слово mutable

Иногда есть необходимость изменить некий объект внутри класса, гарантируя неприкосновенность остальных элементов. Неприкосновенность можно гарантировать при помощи **const**, однако **const** запрещает изменение всего.

class **Exm**

{

int **a;**

int **b;**

public:

int **getA()** const

**{**

return **a;** // все правильно

**}**

int **setA(**int **i)** const

**{**

**a = i;**// ошибка доступа

**}**

}

Помочь в данном случае может определение переменной **а** с ключевым словом **mutable**. Внесём исправление в приведённый чуть выше пример:

class **Exm**

{

mutableint **a;** // добавили в объявление ключевое слово mutable

// позволяющие игнорировать модификатор const

// по отношению к данной переменной

int **b;**

public:

int **getA()** const//

**{**

return **a;** // все правильно

**}**

int **setA(**int **i)** const

**{**

**a = i;**// теперь всё правильно. Мы можем изменять переменную а

**b = i;** // Ошибка! Переменная b по прежнему не доступна для изменения.

**}**

}

**Пример.**

Предположим, вы идете в гостиницу, и вы отдаете приказ официанту принести немного блюд. После подачи заказа вы вдруг решите изменить порядок питания. Предположим, что отель предоставляет возможность изменить заказанную пищу и снова принять заказ новой еды в течение 10 минут после подачи 1-го порядка. Через 10 минут заказ не может быть отменен, и старый заказ не может быть заменен новым заказом. Смотри код.

#include<iostream>

#include<string.h>

using **std::cout;**

using **std::endl;**

class **Customer**

{

char **name[25];**

mutablechar **placedorder[50];**

int **tableno;**

mutableint **bill;**

public:

**Customer(**char\* s, char\* m, int **a,** int **p)**

**{**

**strcpy(name, s);**

**strcpy(placedorder, m);**

**tableno = a;**

**bill = p;**

**}**

void **changePlacedOrder(**char\* p) const

**{**

**strcpy(placedorder, p);**

**}**

void **changeBill(**int **s)** const

**{**

**bill = s;**

**}**

void **display()** const

**{**

**cout <<** "Customer name is: " **<< name << endl;**

**cout <<** "Food ordered by customer is: " **<< placedorder << endl;**

**cout <<** "table no is: " **<< tableno << endl;**

**cout <<** "Total payable amount: " **<< bill << endl;**

**}**

};

int **main()**

{

const **Customer c1(**"Pravasi Meet", "Ice Cream", 3, 100);

**c1.display();**

**c1.changePlacedOrder(**"GulabJammuns");

**c1.changeBill(150);**

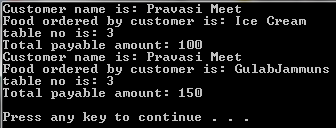
**c1.display();**

**cout <<** "\n";

**system(**"PAUSE");

return **0;**

}



Внимательно наблюдайте вывод вышеприведенной программы. *Значения элементов locationorder и элементов счета изменяются из функции const, потому что они объявлены как изменяемые.*

Ключевое слово mutable в основном используется, чтобы разрешить конкретный член данных объекта const быть модифицированным. Когда мы объявляем функцию as const, этот указатель, переданный функции, становится const. Добавление измененного в переменную позволяет указателю const изменять члены.

*mutable особенно полезен, если большинство членов должны быть постоянными, но некоторые должны быть обновляемыми. Члены данных, объявленные как изменяемые, могут быть изменены, даже если они являются частью объекта, объявленного как const.* **Вы не можете использовать изменяемый спецификатор с именами, объявленными как static или const, или ссылкой.**

Как упражнение предсказывает выход следующих двух программ

// PROGRAM 1

#include<iostream>

using **std::cout;**

class **Test {**

public:

int **x;**

mutableint **y;**

**Test() { x = 4; y = 10; }**

};

int **main()**

{

const **Test t1;**

**t1.y = 20;**

**cout << t1.y;**

**cout <<** "\n";

**system(**"PAUSE");

return **0;**

}



// PROGRAM 2

#include<iostream>

using **std::cout;**

class **Test {**

public:

int **x;**

mutableint **y;**

**Test() { x = 4; y = 10; }**

};

int **main()**

{

const **Test t1;**

**t1.x = 8;**

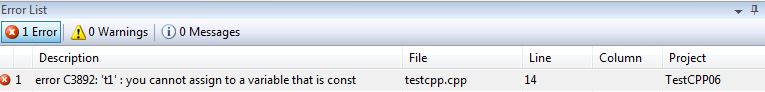
**cout << t1.x;**

**cout <<** "\n";

**system(**"PAUSE");

return **0;**

}



mutable показывает, что член класса является изменяемым, и его можно изменять в функциях, у которых указан модификатор const, а также у константных объектов.

Следующий пример полностью рабочий :

#include<iostream>

using **std::cout;**

struct **foo**

{

**foo () : a(0) {}** //инициализация a=0

void **change ()** const **{ a++; }**

mutableint **a;**

};

int **main ()**

{

const **foo f;**

**f.change();**

**f.a = 100;**

**cout << f.a;**

**cout <<** "\n";

**system(**"PAUSE");

return **0;**

}



Это ключевое слово может использоваться в классах, где, например, есть кэш, который может потребоваться обновлять в каждой функции вне зависимости от требования константности объекта. Еще пример - при ленивой инициализации данных класса(сигнализирующий флаг будет изменяемым, mutable).

## Классы памяти и область действия.

Мы уже упоминали раньше, что локальные переменные известны только функциям, содержащим их. В языке Си предполагается также, что о глобальных переменных «знают» сразу несколько функций. Предположим, например, что и main(), и critic() имеют доступ к переменной units. Это будет иметь место, если отнести units к «внешнему» классу памяти, как показано ниже:

#include<stdio.h>

#include<iostream.h>

int units; /\* внешняя переменная \*/

critic()

{

extern int units;

printf (" You haven't good luck. Try again\n");

//Тебе не везет, попробуй еще

scanf (" %d", &units);

}

main()

{

extern int units;

printf (" How much pounds of butter is on a barrel?\n");

//Сколько фунтов масла в бочонке?

scanf (" %d", &units);

while (units != 56)

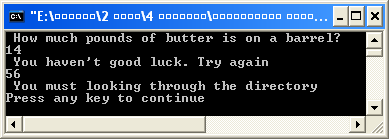
critic();

printf(" You must looking through the directory \n");

//Вы должны посмотреть в справочнике

}

Результаты работы программы



Мы сделали это. Обратите внимание, что второе значение units было прочитано функцией critic(), однако main() также «узнала» это новое значение, когда оно вышло из цикла while.

Мы сделали переменную units внешней, описав ее вне любого определения функции. Далее, внутри функций, использующих эту переменную, мы объявляем ее внешней при помощи ключевого слова extern, предшествующего спецификации типа переменной. Слово extern предлагает компьютеру искать определение этой переменной вне функции. Если бы мы опустили ключевое слово extern в функции critic(), то компилятор создал бы в функции critic новую переменную и тоже назвал бы ее units. Тогда другая переменная units() [которая находится в main()] никогда не получила бы нового значения.

Каждая переменная, как мы знаем, имеет тип. Кроме того, каждая переменная принадлежит к некоторому классу памяти. Есть четыре ключевых слова, используемые для описания классов памяти: extern (для внешнего), auto (для автоматического, static и register. До сих пор мы не обращали внимание на классы памяти, так как переменные, описанные внутри функции, считались относящимися к классу auto, если они не описывались иначе (по умолчанию они относились к классу auto).

Определение класса памяти переменной зависит от того, где переменная описана и какое ключевое слово (если оно есть) используется.

Класс памяти позволяет установить два факта. Во-первых, определить, какие функции имеют доступ к переменной. (Пределы, до которых переменная доступна, характеризуют ее «область действия».) Во-вторых, определить, как долго переменная находится в памяти. Теперь перейдем к свойствам каждого типа.

### Правила области действия переменной.

Для рассмотрения предлагается четыре правила видимости переменной, называемых также правилами области действия. Четыре области действия для переменной следующие: блок, функция, файл и программа. Переменная, объявленная внутри блока или функции известна только в этом блоке и функции. Правила области действия для переменных, используемых с функциями, в С и C++ аналогичны. Переменная, объявленная внешней по отношению к функции, известна в том файле, в котором она описана, начиная с момента объявления и до конца файла. Переменная, объявленная на внешнем уровне в одном исходном файле и объявленная как внешняя в других файлах, имеет в качестве области действия всю программу.

15_01

### Операция уточнения области действия в C++.

Область действия переменной, если она используется в функции, относится к диапазону, где эта переменная доступна. Правила области действия для переменных, используемых с функциями, в С и C++ аналогичны. Переменные могут иметь следующие области действия: локальную, файловую или внутри класса.

Локальную переменную можно использовать в описании функции. Тогда ее область действия ограничена этой функцией. Эта переменная считается доступной или видимой только внутри функции и имеет локальную область действия.

Переменные с областью действия внутри файла объявляются вне отдельных функций или классов. Эти переменный доступны или видимы во всем файле, где они объявлены и являются глобальными.

Можно задать переменную с областью действия внутри файла, а затем включить ее в описание функции, сделав область ее действия локальной. В этом случае локальная область действия имеет приоритет над файловой. В C++ имеется новое программное средство, называемое операцией уточнения области действия (::). Когда используется эта операция, выполняется обращение не к локальной переменной, а к переменной с областью действия внутри всего файла. Тогда переменная становится глобальной. Синтаксис обращения к глобальной переменной следующий:

::yourvariable

Пример (рассматривался в теме функции):

#include <iostream> //cout

#include <conio.h> //\_getch();

using namespace std;

int iproduct(int iw,int ix);

int in=10;

int main()

{

int il=3;

int im=7;

int io;

io=iproduct(il,im);

cout << "The product of " << il <<" \* " << im

<< " \* " << in << " is: " << io;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iproduct(int iw,int ix)

{

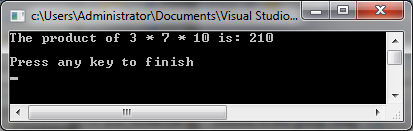
int iy;

int in=2;

iy=iw\*ix\*(::in); //iy=iw\*ix\*(in);

return(iy);

}



Операцию уточнения области действия не обязательно заключать в круглые скобки — они использованы в этом примере только для выделения операции. Теперь при вычислениях используется глобальная переменная с файловой областью действия. При выводе результата на экран вы увидите значение 3 \* 7 \* 10 = 210.

Операция уточнения области действия очень важна в C++. Правила области действия для переменных обуславливают появление своеобразных программных ошибок. Дополнительные примеры применения этой операции приводятся в последующем изложении материала.

### Выбор класса памяти.

Ответ на вопрос почти всегда один — «автоматический». В конце концов, почему этот класс памяти выбран по умолчанию? Мы знаем, что на первый взгляд использование внешних переменных очень соблазнительно. Опишите все ваши переменные как внешние, и у вас никогда не будет забот при использовании аргументов и указателей для связи между функциями в прямом и обратном направлениях. К сожалению, у вас возникнет проблема с функцией С, коварно изменяющей переменные в функции А, а это совсем не входит в ваши интересы. Неоспоримый совокупный опыт использования машин, накопленный в течение многих лет, свидетельствует о том, что такая проблема значительно перевешивает кажущуюся привлекательность широкого использования внешних переменных.

Одно из золотых правил защитного программирования заключается в соблюдении принципа «необходимо знать только то, что нужно». Организуйте работу каждой функции автономно, насколько это возможно, и используйте глобальные переменные только тогда, когда это действительно необходимо.

Иногда полезны и другие классы памяти. Но прежде чем их использовать, спросите себя, необходимо ли это.

**Резюме: Классы памяти**

**Ключевые слова** auto, extern, static, register

**Общие замечания**

Класс памяти определяет область действия переменной и продолжительность ее существования в памяти. Класс памяти устанавливается при описании переменной с соответствующим ключевым словом. Переменные, определенные вне функции, являются внешними и имеют глобальную область действия. Переменные, определенные внутри функции, являются автоматическими и локальными, если только не используются другие ключевые слова. Внешние переменные, определенные раньше функции, доступны ей, даже если не описаны внутри ее.

Свойства

|  |  |  |  |
| --- | --- | --- | --- |
| КЛАСС  ПАМЯТИ | КЛЮЧЕВОЕ  СЛОВО | ПРОДОЛЖИТЕЛЬНОСТЬ  СУЩЕСТВОВАНИЯ | ОБЛАСТЬ  ДЕЙСТВИЯ |
| Автоматический  Регистровый  Статический | auto  register  static | Временно  Временно  Постоянно | Локальная  Локальная  Локальная |
| Внешний  Внешний статический | extern  static | Постоянно  Постоянно | Глобальная (все файлы)  Глобальная (один файл) |

Классы памяти, перечисленные выше пунктирной линии, описываются внутри функции. Классы памяти, перечисленные ниже этой линии, определяются вне функции.

Теперь рассмотрим функцию, использующую внешнюю статическую переменную.

## Пространства имен

Определения функций и переменных в заголовочных файлах неразрывно связаны с понятием пространства имен. Это понятие появилось сравнительно недавно. *До введения понятия пространства все объявления идентификаторов и констант, сделанные в заголовочном файле, помещались компилятором в глобальное пространство имен.* Такое положение вещей приводило к возникновению массы конфликтов, связанных с использованием различными объектами одинаковых имен. Чаще всего недоразумения возникали, когда в одной программе использовались библиотеки, разработанные различными производителями. Введение понятия пространства имен позволило значительно снизить количество подобных конфликтов имен. Когда в программу включается заголовок нового стиля, его содержимое помещается не в глобальное пространство имен, а в пространство имен std. Если в программе требуется определить некоторые идентификаторы, которые, как Вы подозреваете, могут переопределить уже имеющиеся, просто заведите свое собственное, новое пространство имен. Это достигается путем использования ключевого слова **namespace**:

namespace \_имя\_пространства\_

{

// объявления

…

}

Таким образом, объявления внутри нового пространства имен будут находится только внутри видимости определенного имени \_пространства\_имён, предотвращая тем самым возникновение конфликтов. В качестве примера создадим следующее пространство имен:

namеspace NewNameSpace

{

int x, у, z;

void SomeFunction(char smb);

}

Для того чтобы указать компилятору, что следует использовать имена из конкретного именного пространства (в данном случае из **NewNameSpace**), можно воспользоваться операцией разрешения видимости:

NewNameSpace::х = 5;

Однако, если в программе обращения к собственному пространству имен производятся довольно часто, такой синтаксис вызывает определенные неудобства. В качестве альтернативы можно воспользоваться инструкцией **using**, синтаксис которой имеет две формы:

using namespace имя\_пространства имен;

или

using имя\_пространства\_имен::идентификатор;

При использовании первой формы компилятору сообщается, что в дальнейшем необходимо использовать идентификаторы из указанного именного пространства вплоть до того момента, пока не встретится следующая инструкция **using**. Например, указав в теле программы

using namespace NewNameSpace;

можно напрямую работать с соответствующими идентификаторами:

х=0; y=z=4;

SomeFunction('А');

На практике часто после включения в программу заголовков явно указывается использование идентификаторов стандартного пространства имен:

using namespace std;

Вторая форма записи предписывает компилятору использовать указанное пространство имен лишь для конкретного идентификатора. Таким образом, определив

using namespace std;

using NewNameSpace::z;

можно использовать идентификаторы стандартной библиотеки C++ и целочисленную переменную z из пространства имен NewNameSpace без использования операции разрешения видимости:

z=12;

*Следует понимать, что указание нового пространства имен инструкцией* ***using namespace*** *отменяет видимость стандартного пространства* ***std****, поэтому для получения доступа к соответствующим идентификаторам из std потребуется каждый раз использовать операцию разрешения видимости std::.*

**Пространства имен не могут быть объявлены внутри тела какой-либо функции, однако могут объявляться внутри других пространств.** При этом для доступа к идентификатору внутреннего пространства необходимо указать имена всех вышестоящих именованных пространств. Например, объявлено следующее пространство имен:

namespace Highest

{

namespace Middle

{

Namespace Lowest

{

int nAttr;

}

}

}

Использование объявленной переменной nAttr будет выглядеть:

Highest :: Middle:: Lowest:: nAttr = 0;

Допускается объявление нескольких именованных пространств с одним и тем же именем, что позволяет разделить его на несколько файлов. Несмотря на это, содержимое всех частей будет храниться в одном и том же пространстве имен.

**Чтобы объявления переменных и функций в некотором пространстве имен были более упорядоченными, рекомендуется в пределах описания пространства имен объявлять только прототипы функций, помещая определение тела функции отдельно. При этом следует явно указывать, к какому пространству имен относится функция:**

namespace Nspace

{

char с;

int i;

void Fund (char Flag);

}

void Nspace::Fund(char Flag)

{

// тело функции

…

}

Кроме вышесказанного, допускается объявление неименованных пространств имен. В этом случае просто опускается имя пространства после ключевого слова namespace. Например:

namespace

{

char cByte;

long lvalue;

}

Обращение к объявленным элементам производится по их имени, без какого-либо префикса. **Неименованные пространства имен могут быть использованы только в том файле, в котором они объявлены.**

**Стандарт языка C++ предусматривает определение псевдонимов пространства имен, которые ссылаются на конкретное пространство имен. Чаще всего псевдонимы используются для упрощения работы с длинными именами пространств**. Следующий пример иллюстрирует создание более короткого псевдонима и его использование для доступа к переменной.

namespace A\_Very\_Long\_Name\_Of\_NameSpace

{

float у;

}

A\_Very\_Long\_Name\_Of\_NameSpace:: у = 0.0;

namespace Neo = A Very\_Long\_Name\_Of\_NameSpace;

Neo::y = 13.4;

Предположим, у вас есть крупный проект который содержит огромное количество классов и переменных. Естественно над ним не будет работать один человек, а значит возможны ошибки из-за, например, создание второго класса с именем Employee (ведь программист может и не знать, что кто-то уже создал класс с таким именем). Программисты называют это коллизиями имен. Чтобы избежать подобных проблем и придумали объединять большие участки кода в пространство имен. Они добавляют своеобразный префикс к каждому идентификатору в данном пространстве.

Давайте рассмотрим небольшой пример:

#include<iostream>

namespace **myMath {** // наше пространство имен

int **abs(**int **x) {** // функция нахождения модуля

if **(x < 0) {**

return **-x;**

**}** else **{**

return **x;**

**}**

**}**

constdouble **E = 2.718;** // константа

}

int **main() {**

**cout <<** "|-1| = " **<< myMath::abs(-1) << endl;** // пользуемся пространством имен

**cout <<** "e^2 = " **<< myMath::E \* myMath::E << endl;**

**system(**"PAUSE");

return **0;**

}



Возможно вы уже догадались как использовать данную конструкцию, но для полного понимания давайте рассмотрим программу более подробно:

* Сначала мы объявляем пространство имен, в котором будем работать. Делается это при помощи ключевого слова namespace после которого следует имя пространства. Все то, что окажется внутри фигурных скобок будет принадлежать данному пространству имен.
* Далее мы указываем код, который хотим «наградить» специальным префиксом (в нашем случае это myMath::).
* Теперь мы можем использовать написанный ранее код также, как и код вне пространства имен с одной лишь разницей. Теперь мы обязаны указывать принадлежность функции или переменной к конкретному пространству имен. Делается это при помощи следующей конструкции:

<имя пространства>::<имя его члена>

Как видите нет ничего сложного. Но это в некоторых случаях очень неудобно, постоянно указывать префикс. Хотелось бы иметь возможность использовать по умолчанию какое-либо пространство имен. Хорошо, что C++ поддерживает такую возможность.

Как вы можете заметить, в начале каждой программы мы для удобства опускали стандартный префикс std (стандартная библиотека шаблонов находится именно в данном пространстве имен). Делали мы это для удобства чтения кода, а также для того, чтобы не усложнять код наших программ. В реальных проектах рекомендуется этого не делать.

**Так вот, чтобы опустить какой-либо префикс, нужно это поле ключевых слов using namespace указать имя пространства имен, префикс которого мы не хотим указывать. Вот пример:**

#include<iostream>

namespace **CL {** // пространство

int **var = 0;**

}

usingnamespace **CL;** // отбрасываем префикс CL

int **main() {**

**std::cout <<** "var= " **<< var <<** "\n";

**var = 1;** // используем переменную var

**std::cout <<** "var= " **<< var <<** "\n";

**system(**"PAUSE");

return **0;**

}



**Пример**

**MyClass.h**

namespace **myNameSpace {**

class **MyClass {**

int **i;**

public:

**MyClass();**

int **getValue();**

**};**

}

**MyClass.cpp**

#include"MyClass.h"

myNameSpace::MyClass::MyClass() {

**i = 123;**

}

usingnamespace **myNameSpace;**

int **MyClass::getValue() {**

return **i;**

}

**TestCPP.cpp**

#include<iostream>

#include"MyClass.h"

usingnamespace **std;**

usingnamespace **myNameSpace;**

int **main() {**

**MyClass a;**

**cout << a.getValue() << endl;**

**system(**"PAUSE");

return **0;**

}



## Функции и классы памяти.

Как вы видели ранее, в объявлении типа данных может участвовать спецификатор класса памяти. К примеру, переменную можно объявить следующим образом:

static float fyourvariable;

**Функции также могут иметь класс памяти extern и static. Функция объявляется с классом памяти extern, если она описана в другом файле, внешнем по отношению к данной программе. Невозможно объявить функцию static так, чтобы к ней можно было обращаться извне, кроме как из файла, где она описана.**

Мы уже несколько раз касались вопроса о том, что переменные в функции являются ее внутренними переменными и «не известны» вызывающей функции. Аналогично переменные вызывающей функции не известны вызываемой функции. Вот почему для связи с ней, т. е. для передачи значений в нее и из нее, мы пользуемся аргументами и оператором return. Переменные, известные только одной функции, а именно той, которая их содержит, называются «локальными» переменными. До сих пор это был единственный вид переменных, которыми мы пользовались, но в языке Си допускается наличие переменных, известных нескольким функциям. Такие нелокальные переменные называются «глобальными», и мы вернемся к ним позже. Теперь же мы хотим подчеркнуть, что локальные переменные являются действительно локальными. Даже в том случае, если мы используем одно и то же имя для переменных в двух различных функциях, компилятор (и, таким образом, компьютер) «считает» их разными переменными. Мы можем показать это, используя операцию & (не путайте с операцией &&).

### Объявления функций на внешнем уровне.

При объявлении функции на внешнем или внутреннем уровнях можно использовать спецификаторы класса памяти **static** или **extern**. *Функции, в отличие от переменных, всегда имеют глобальное время жизни. Правила видимости для функций немного отличаются от правил видимости переменных.*

**Функции, объявленные как static, видимы только в том исходном файле, в котором они описаны. Функции, находящиеся в том же файле, могут вызывать статическую функцию, а функции из других файлов — не могут. Кроме того, в другом исходном файле можно объявить другую статическую функцию с таким же именем, и не будет конфликта.**

**Функции, объявленные как external, видимы во всех исходных файлах, образующих программу (если вы где-то не переопределите функцию как static). Внешнюю функцию могут вызывать все функции.**

**Замечание**

Объявления функций, в которых опущен спецификатор класса памяти, являются по умолчанию external.

## Функция получения случайных чисел.

Вы не можете обойтись без функции получения случайных чисел. Когда кто-либо требует от вас какое-нибудь число, нужно обратиться к этому полезному средству, вместо того чтобы, заикаясь, каждый раз оправдываться. Вы можете использовать ее во многих машинных играх, что менее практично.

Сейчас мы покажем вам «генератор» псевдослучайных чисел. Это означает, что фактическая последовательность чисел предсказуема (компьютерам не известна их самопроизвольность), но они разбросаны довольно равномерно в пределах возможного диапазона значений.

Схема начинает с числа, называемого «зерно». Она использует его для создания нового числа, которое становится новым зерном. Затем новое зерно можно использовать для создания более нового зерна и т. д. Чтобы эта схема работала, функция случайных чисел должна помнить зерно, которое она использовала при последнем вызове. Ага! Она требует статическую переменную. Вот версия 1. (Скоро будет и версия 2.)

/\* версия1 функции rand() \*/

rand()

{

static int randx = 1;

randx = (randx \* 25173 + 13849) % 65536; /\* магическая формула \*/

return( randx);

}

Статическая переменная randx начинает со значения 1 и изменяется при помощи магической формулы каждый раз при вызове функции. Результатом в нашей системе является число, находящееся в диапазоне от - 32768 до 32767. Системы с разной длиной переменной типа int будут давать различные результаты.

Проверим работу функции при помощи этого простого драйвера:

/\* драйвер1 функции rand() \*/

main()

{

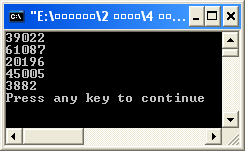
int count;

for(count = 1; count <= 5; count++ )

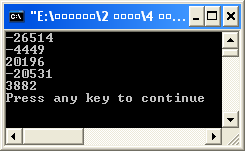
printf(" %d\n" , rand());

}

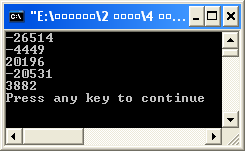
Результаты работы программы:



Получаем результат:



Эта последовательность чисел выглядит довольно случайной. Запустим драйвер еще раз. Теперь имеем



Получилось абсолютно то же самое; это и есть «псевдоэффект». Каждый раз, когда работает основная программа, мы начинаем с одного и того же значения зерна, равного 1. Можно обойти эту проблему, введя вторую функцию srand(), которая позволяет вновь устанавливать зерно в начальное значение. Хитрость заключается в том, чтобы сделать randx внешней статической переменной, известной только функциям rand() и srand(). Храните эти две функции в своем собственном файле и компилируйте этот файл отдельно. Вот модификация программы:

/\* файл для rand() и srand() \*/

static int randx =1;

rand()

{

randx = (randx \* 25173 + 13849) % 65536;

return(randx);

}

srand (unsigned x)

{

randx = x;

}

Используем другой драйвер:

/\* драйвер2 функции rand() \*/

main()

{

int count;

int seed;

printf(" Please, enter your value of a corn\n");

//Пожалуйста, введите свое значение зерна

scanf(" %d" , &seed);

srand(seed);/\* установите зерно в начальное значение \*/

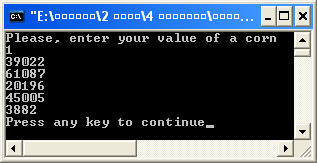
for (count = 1; count <= 5; count++ )

printf(" %d\n" , rand());

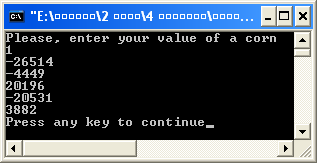
}

}

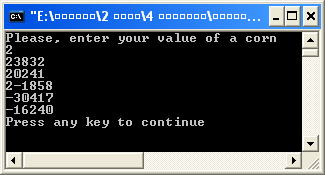
Результаты работы программы



Программа проработала один раз:



Используя значение 1 для переменной seed, получаем те же значения, что и прежде. Давайте теперь попытаемся ввести значение 2:



Очень хорошо! Мы получили другую последовательность чисел. Теперь попытаемся придумать применение для этих функций.

## Игра в кости.

Мы собираемся смоделировать очень распространенную игру в кости. В наиболее популярной форме этой игры используются две фишки, имеющие по 6 сторон (два кубика). Однако есть и другие варианты. Многие игроки и азартные игры используют любые из пяти геометрически возможных фишек, имеющих 4, 6, 8, 12 и 20 сторон. (Древние греки доказали, что есть только 5 правильных геометрических тел с гранями одинаковой формы и размера; эти тела и являются основой вариантов фишек. Можно сделать фишку с иным количеством сторон, но тогда не все грани будут одинаковыми и такие фишки не будут иметь равные шансы при бросании.)

На машинные вычисления эти. геометрические рассуждения не налагают ограничений, поэтому мы изобретем электронную игральную кость, у которой по вашему желанию может быть любое число сторон. Давайте начнем с 6 сторон, затем сделаем обобщение. Нам нужны случайные числа от 1 до 6, а функция rand() создает числа в диапазоне от —32768 до 32767, поэтому мы должны внести некоторые поправки. Вот один подход

Разделим случайное число на 32768. В результате получим число х в диапазоне -1 < = х < 1. (Мы должны превратить его в тип float, чтобы иметь десятичные дроби.)

Добавим 1. Наше новое число удовлетворяет отношению 0 <= х < 2.

Разделим на 2. Теперь имеем 0 < = х < 1.

Умножим на 6. Имеем 0 < = х < 6. (Близко к тому, что нужно, но 0 не является возможным значением.)

Прибавим 1: 1 < = х < 7. (Заметим, что эти числа все еще являются десятичными дробями.)

Преобразуем в целые числа. Теперь мы имеем целые в диапазоне от 1 до 6.

Для обобщения достаточно заменить значение 6 в п. 4 на число сторон.

Вот функция, которая выполняет эти действия:

/\* электронное бросание костей \*/

#define SCALE 32768.0

rollem (sides);

float sides;

{

float roll;

roll = ( (float) rand()/SCALE + 1.0) \* sides/2.0 + 1.0;

return ( (int) roll);

}

Мы включили в программу два явных описания типа, чтобы показать, где выполняются преобразования типов.

Обратимся к программе, которая использует эти средства:

/\* многократное бросание кости \*/

main()

{

int dice, count, roll, seed; float sides;

printf(" Введите, пожалуйста, значение зерна. \n");

scanf(" %d", &seed);

srand (seed);

printf(" Введите число сторон кости, 0 для завершения\n");

scanf(" %d" , &sides);

while (sides > 0)

{

printf(" Сколько костей? \n" );

scanf(" %d", dice);

for ( roll = 0, count = 1; count <= dice; count++ )

roll += rollem(sides); /\* бросание всех костей набора \*/

printf("У вас выпало %d, для %d %. 0f-сторонних костей.\n" , roll, dice, sides);

printf(" Сколько сторон? Введите 0 для завершения.\n");

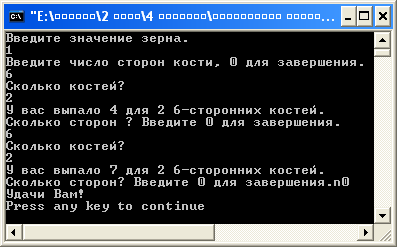
scanf("%f", &sides);

}

printf(" Удачи вам!\n");

}

Теперь давайте используем эту программу:



Вы можете использовать функцию rollem() по-разному. Пусть число сторон (sides) равно двум, тогда бросание монеты даст следующий результат: «орел» выпал 2 раза, а «решка» — один (или наоборот, смотря, что вы предпочитаете). Можно легко модифицировать программу, чтобы показать как отдельные результаты, так и итог. Или вы можете построить имитатор игры «крапе». Если вам нужно большое число бросаний, вы можете легко модифицировать свою программу и получить результат, подобный следующему:

Введите значение зерна.

10

Введите количество ходов; введите 0 для завершения.

18

Сколько сторон и сколько костей?

6 3

Здесь 18 ходов для 3 6-сторонних костей.

7 5 9 7 12 10 7 12 10 14

9 8 13 9 10 7 16 10

Сколько ходов? Введите 0 для завершения.

0

Использование функции rand() [но не rollem()] изменило бы вашу программу угадывания чисел: компьютер стал бы выбирать, а вы угадывать, вместо того чтобы сделать наоборот.

Разработаем еще некоторые функции. Сначала мы хотим создать функцию, которая читает целые числа.

## Функция получения целых чисел: getint().

Вероятно, наш проект покажется вам очень простым. В конце концов, мы можем использовать функцию scanf() с форматом %d, если хотим прочитать целое число. Такой подход очень прост, но он имеет один большой недостаток. Если вы ошибочно напечатали, скажем, Т вместо 6, scanf() попытается интерпретировать Т как целое число. Мы хотим создать функцию, которая проверяет вводимую информацию и предупреждает вас, если введено нецелое число. Теперь, может быть, наш проект уже не кажется таким простым. Однако не беспокойтесь — мы хорошо начали: у нас есть имя для новой функции. Мы назовем ее getint().

***План***

К счастью, мы уже выработали стратегию. Во-первых, заметим, что любую вводимую информацию можно читать как строку символов. Целое число 324, например, можно прочитать как строку из трех символов: символ ' 3', символ ' 2' и символ ' 4'. Это подсказывает нам следующий план:

Прочитать вводимую информацию как символьную строку.

Проверить, состоит ли строка только из символов цифр и стоит ли перед ними знак плюс или минус.

Если все это имеет место, превратить символьную строку в правильное числовое значение.

Если нет, выдать предупреждение.

Этот план так хорош, что он должен работать. (Тот факт, что он представляет собой стандартный подход, существовавший на протяжении многих лет, придает нам также некоторую уверенность в возможности его выполнения. Но, прежде чем начать программировать, нужно подумать, что будет делать наша функция. В частности, до того как мы займемся содержанием нашей функции getint(), нужно точно решить, как она должна взаимодействовать со своим окружением: с какой информацией? Какую информацию она должна получать от вызывающей программы? Какую информацию должна возвращать? В каком виде должна быть эта информация? Снова мы рассматриваем функцию как черный ящик. Мы хотим знать, что входит в функцию и что выходит из нее и, наконец, что находится внутри ее. Этот подход помогает обеспечивать более однородное взаимодействие между различными частями программы. Иначе вы можете оказаться в положении человека, пытающегося установить трансмиссию автомашины «Вольво" в автомобиль «Тойота». Сама функция правильная, но интерфейс является проблемой.

***Поток информации для getint()***

Какой выход должна иметь наша функция? Во-первых, несомненно, что она должна была бы выдавать значение прочитанного числа. Конечно, функция scanf() уже делает так. Во-вторых, и это очень существенно, мы собираемся создать такую функцию, которая будет выдавать сообщения о состоянии, т. е. найдено или нет целое число. Чтобы функция была действительно полезной, она должна также сообщать о нахождении ею символа EOF. Тогда мы могли бы использовать функцию getint() в цикле while, который читает целые числа до тех пор, пока не обнаружит символ EOF. Короче говоря, нам нужно, чтобы getint() возвращала два значения целое число и состояние.

Так как мы хотим иметь два значения, то с одной функцией return с этой задачей нам не справиться. Нам следует иметь два указателя. Однако полное решение задачи мы получим, если используем указатели для выполнения основной работы функции и функцию return для выдачи некоторого кода состояния. Именно это и делает функция scanf(). Она возвращает количество символов, которые нашла, и символ EOF, если встречает его. Ранее мы не пользовались такой возможностью, но могли бы, если бы применяли вызов вида

status = scanf(" %d" , &number);

Теперь будем это делать. Тогда наш вызов функции выглядел бы следующим образом:

status = getint(&number);

Правая часть равенства использует адрес переменной number, чтобы получить ее значение, a return применяется для получения значения переменной status.

11_02

Мы должны выбрать коды для выдачи сообщения о состоянии. Так как предполагается, что неописанные функции имеют тип int, наши коды должны состоять из целых чисел. Используем следующие коды для сообщения о состоянии:

-1 означает, что найден символ EOF

1 означает, что найдена строка, содержащая не цифры.

0 означает, что найдена строка, содержащая только цифры.

Нашу функцию getint() можно представить себе как имеющую один вход и два выхода. На ее вход поступает адрес целой переменной, значение которой считывается. На первом выходе имеем значение считанного целого, полученного через указатель. (Таким образом, аргумент-указатель является двусторонним каналом передачи информации.) На втором выходе получаем код состояния, что обеспечивается функцией return. Отсюда следует, что «скелет» нашей функции должен выглядеть примерно так:

getint(ptint)

int \*ptint; /\* указатель на целое число \*/

{

int status;

...

return(status);

}

Замечательно! Теперь мы должны просто заполнить внутренность функции.

***Содержание getint()***

Наш основной план для getint() в общих чертах на псевдокоде выглядит примерно так:

* читаем на входе информацию в виде символов
* помещаем символы в строку, пока не встретим символ EOF
* если встретился символ EOF, устанавливаем состояние в STOP в противном случае
* проверяем строку, преобразуем символы в целое число, если возможно,
* и выдаем сообщение о состоянии (YESNUM или NONUM)

Здесь мы используем STOP, YESNUM и NONUM как символические константы, равные —1, 0 и 1, как описано выше.

Рассмотрим еще некоторые вопросы. Как функция будет решать, что она достигла конца входной строки? Должны ли мы ограничивать длину строки?

Мы вошли в область, где нам предстоит решать, что предпочесть: удобство программиста или удобство пользователя. Самым простым было бы предложить пользователю нажимать на клавишу [ввод], когда строку надо закончить. Это означало бы один ввод на строку. Для пользователя все же было бы приятнее, если бы функция могла размещать несколько чисел в одной и той же строке:

2 34 4542 2 98

Мы решили предоставить привилегию пользователю. Пусть функция будет считать, что строка начинается с символа, не являющегося пробелом или «новой строкой», и заканчивается символом пробела или «новой строкой». Такой ввод может производиться в одну строку или в несколько строк.

Мы ограничим вводимую строку 80 символами. Так как строки заканчиваются нуль-символом, нам нужен массив из 81 символа для включения в него этого символа. Это слишком щедро, потому что нам нужно только 6 символов для 16-разрядного целого числа и знака. Вы можете вводить более длинные числа, но их размер будет сокращен до размера строки.

Чтобы сделать программу более модульной, мы поручим преобразование «строка в целое число» другой функции, и назовем ее stoi(). У нас будет также возврат функцией stoi() соответствующего кода состояния в функцию getint(), a getint(), в свою очередь, может передать код состояния своей вызывающей программе. Функция stoi() выполнит последние две строки нашего плана (на псевдокоде).

/\* getint() \*/

#include <stdio.h>

#define LEN 81 /\* максимальная длина строки \*/

#define STOP -1 /\* коды состояний \*/

#define NONUM 1

#define YESNUM 0

getint(ptint)

int \*ptint; /\* указатель на вывод целого числа \*/

{

char intarr[LEN]; /\* запоминание вводимой строки \*/

int ch;

int ind = 0; /\* индекс массива \*/

/\* обход начальных символов «новая строка», пробелов и табуляций \*/

while ((ch = getchar()) == ' \n' || ch == " || ch == ' \t');

while (ch != EOF && ch != '\n' && ch != ' ' && ind < LEM)

{

intarr[ind++] = ch; /\* запись символа в массив \*/

ch = getchar(); /\* получение очередного символа \*/

}

intarr[ind] = ' \0'; /\* конец массива по нуль-символу \*/

if (ch == EOF)

return(STOP);

else

return ( stoi(intarr, ptint) ); /\* выполнение преобразования \*/

}

Мы получаем символ ch. Если он является символом пробела, или «новой строки», или табуляции, мы берем следующий символ и так продолжаем до тех пор, пока не получим символ, отличающийся от перечисленных. Затем, если этот символ не EOF, помещаем его в массив. Продолжаем брать символы и помещать их в массив, пока не найдем запрещенный символ или не достигнем предельного размера строки. Далее помещаем нуль-символ ('\0') в следующую позицию массива, чтобы отметить конец строки. Таким образом, мы создали массив в виде стандартной символьной строки. Если EOF был последним прочитанным символом, возвращаем STOP; иначе идем дальше и пытаемся преобразовать строку. Мы вызываем новую функцию stoi(), чтобы выполнить эту работу. Что делает stoi()? При вводе она берет символьную строку и указатель на целую переменную, использует указатель для присваивания значения самой переменной, а также return для пересылки сообщения о состоянии, которое getint() передает затем функции getarray(). Поразительно! Двойная игра! Вот менее компактный способ использования функции stoi():

status = stoi(intarr, print);

return (status);

Здесь status была бы переменной типа int. Первый оператор дает значение, на которое указывает ptint; она также присваивает значение переменной Status. Второй оператор возвращает это значение программе, которая вызвала getint(). Наша единственная строка программы имеет точно такой же эффект, за исключением того, что нам не нужна промежуточная переменная status. Теперь напишем функцию stoi().

**Преобразование строки в целое: stoi()**

Сначала опишем, каким должен быть вход и выход у этой функции. Вход будет символьной строкой, поэтому stoi() будет иметь символьную строку в качестве аргумента. На выходе должно быть получено два значения: состояние и преобразованное целое число, Мы применяем return для состояниями поэтому должны использовать указатель для возврата другого значения. Таким образом, появится второй аргумент — указатель на целое число. Скелет нашей функции будет выглядеть примерно так:

stoi(string, intptr)

char string[]; /\* вводимая строка \*/

int \*intptr; /\* указатель на переменную, получающую целое значение\*/

{

int status;

...

return(status);

}

Прекрасно, а что можно сказать об алгоритме выполнения преобразования? На некоторое время проигнорируем знак и предположим, что строка содержит только цифры. Возьмем первый символ и преобразуем его в числовой эквивалент. Предположим, это символ ' 4'. Он имеет в коде ASCII числовое значение 52 и в таком виде запоминается. Если мы из него вычтем 48, то получим 4, т. е.

'4' - 48 = 4

Но 48 — это ASCII-код символа ' 0', поэтому

'4' - '0' =4

Действительно, этот последний оператор был бы справедлив в любом коде, в котором используются последовательные числа для представления последовательных цифр. Поэтому если num — числовое значение, a chn — символ цифры, то

num = chn — ' 0';

Итак, мы используем этот метод для преобразования первой цифры в число. Теперь возьмем следующий элемент массива. Если он ' \0', то у нас была только одна цифра, и мы закончили работу. Предположим, однако, что этот элемент ' 3'. Превратим его в числовое значение 3. Но если оно равно 3, то 4 должно было быть числом 40, а оба числа вместе 43:

num = 10 \* num + chn - ' 0';

Теперь просто продолжим этот процесс, умножая старое значение num на 10 каждый раз, когда мы находим следующую цифру. Наша функция будет использовать этот метод.

Вот определение функции stoi(). Мы храним ее в том же файле, что и getint(), так что она может использовать те же самые директивы #define.

/\* превращает строку в целое число и сообщает о состоянии \*/

stoi(string, intptr)

char string[]; /\* строка, подлежащая преобразованию в целое\*/

int \*intptr; /\* значение целого \*/

{

int sign =1; /\* проверяет наличие знака + или — \*/

int index = 0;

if (string[index] == ' -' || string[index] == ' + ')

sign = (string[index++] == '-') ? -1 : 1;/\* установить знак \*/

\*intptr = 0; /\* начальное значение \*/

while(string[index] >= '0' && string [index] <= '9')

\*intptr = 10 \* (\*intptr) + string[index++ ] - '0';

if (string [index] == ' \0')

{

\*intptr = sign \* (\*intptr);

return(YESNUM);

}

else /\* найден символ, отличный от цифры, знака или '\0'\*/

return(NONUM);

}

Оператор while продолжает работу, преобразуя цифры в числа, пока не достигнет нецифрового символа. Если это символ ' \0', все прекрасно, потому что он означает конец строки. Любой другой нецифровой символ отсылает программу к else для сообщения об ошибке.

Стандартная библиотека языка Си содержит функцию atoi() (перевод кода ASCII в целое число), очень похожую на stoi(). Основная разница заключается в том, что stoi() проверяет на нецифровые строки, a atoi() использует return вместо указателя, для возврата числа, и пропускает пробел, как мы это делали в getint(). Можно было бы осуществить все проверки состояния в getint() и использовать atoi() вместо stoi(), но мы полагаем, что было бы интереснее разработать нашу собственную версию.

***Проверка***

Так ли уж правильны наши рассуждения? Давайте проверим нашу функцию на учебной программе:

/\* проверка функции getint()\*/

#define STOP -1

#define NONUM 1

#define YESNUM 0

main()

{

int num, status;

printf(" Программа прекращает считывание чисел, если

встречает EOF.\n" );

while ( (status = getint(&num)) != STOP)

if (status == YESNUM)

printf(" Число %d принято \n", num);

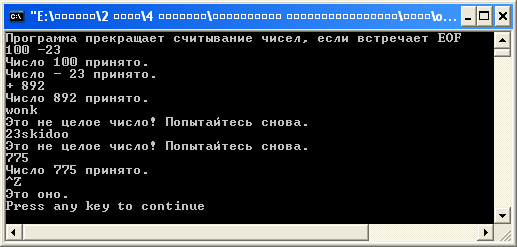
else

printf(" Это не целое число! Попытайтесь снова\n");

printf("Это оно.\n");

}

Вот пример работы программы:



Как видите, программа выполняется. Обратите внимание на то, как мы сумели организовать цикл для неограниченного считывания целых чисел до тех пор, пока не будет введен символ EOF. Это удобное свойство.

Есть ли здесь ошибки? По меньшей мере, одна. Если непосредственно за числом следует символ EOF без разделяющего пробела или символа новой строки, ввод прекращается, и это число не принимается во внимание:

706 EOF /\* 706 принято\*/

706EOF /\* 706 не принято\*/

Мы не хотели делать пример слишком сложным, поэтому допустили возможность этой ошибки. Дальнейшую разработку программы, как мы думаем, может осуществить сам читатель в качестве упражнения.

Теперь, когда у нас есть удобная функция для получения целых чисел, займемся новой задачей, в которой она будет использоваться.

## Сортировка.

Одним из наиболее распространенных тестов для машин является сортировка. Мы хотим разработать программу для сортировки целых чисел. Снова применим принцип черного ящика и подумаем в терминах ввода и вывода. Наш общий замысел, показанный на рисунке, довольно прост.

11_03

На данном этапе программа еще недостаточно определена, чтобы ее кодировать. Следующий шаг заключается в определении главных задач, которые должна выполнить программа для достижения поставленных целей. Их три:

Считывание чисел.

Сортировка чисел.

Печать отсортированных чисел.

На рисунке показано это разделение при движении от верхнего уровня конструкции вниз к более детальному.

11_04

Теперь мы имеем три черных ящика, каждый со своим входом и выходом. Можно передать каждую часть разным группам программистов, если мы уверены в том, что числа, выводимые блоком «считывание», представлены в той же самой форме, которую использует блок «сортировка» для ввода.

Как вы можете видеть, делается упор на модульность. Мы разделили исходную задачу на три более мелких, но лучше управляемых.

Что дальше? Теперь приложим наши усилия к каждому из трех модулей в отдельности, разделяя их на более простые элементы до тех пор, пока не достигнем момента, когда программа станет очевидной. Делая это, обратим внимание на такие важные проблемы, как выбор представления данных, локализация ошибок и передача информации.

Вернемся к нашему примеру и займемся сначала задачей считывания.

***Считывание числовых данных***

Многие программы включают считывание чисел, поэтому идеи, которые мы развиваем здесь, будут полезны везде. Общий вид первой части программы ясен: использовать цикл для считывания чисел до тех пор, пока все числа не будут считаны. Но в этом есть нечто большее, чем вы можете себе представить!

11_05

Выбор представления данных

Как мы представляем группу чисел? Можно использовать группу переменных, по одной на каждое число. Об этом даже страшно подумать. Можно использовать массив, по одному элементу на каждое число. Это значительно лучше, поэтому давайте использовать массив.

Однако какого типа будет массив? Типа, int? Типа double? Нам нужно знать, как такую программу можно будет применять. Предположим, что она должна работать с целыми числами. (А что если она должна применять и целые, и нецелые числа? Это возможно, но потребуется работы больше, чем нам бы хотелось сейчас.) Будем использовать массив целых чисел для запоминания чисел, которые мы считываем.

Завершение ввода

Как программа «узнает», сколько ей нужно считать чисел? Ранее мы обсудили несколько решений этой проблемы, большинство из которых были неудовлетворительны. Однако теперь, когда есть функция getint(), у нас нет проблем. Вот один подход:

* читаем число до тех пор пока не встретится символ EOF
* заносим число в массив и
* читаем следующее число, если массив не заполнен

Заметим, что здесь есть два разных условия, приводящих к завершению этой части программы: символ EOF и заполнение массива.

***Дальнейшие рассуждения***

Прежде чем реализовать все это на языке Си, нам нужно еще решить, что будем делать с проверкой ошибок? Должны ли мы превратить эту часть программы в функцию?

Сначала мы решим, что делать, если пользователь вводит ошибочные данные, скажем букву вместо целого числа? Без функции getint() мы полагались бы на «гипотезу идеального пользователя», согласно которой пользователь не делает ошибок при вводе. Однако мы считаем, что эту гипотезу нельзя применять ни к одному пользователю, кроме себя. К счастью, можно использовать способность функции getint() сообщать о состоянии, что поможет нам выйти из затруднительного положения.

Теперь займемся программированием, которое можно легко реализовать в main(). Для соблюдения модульности следует использовать разные функции для каждой из трех основных частей программы, что мы как раз и сделаем. Входом для этой функции будут числа с клавиатуры или файл, а выходом — массив, содержащей неотсортированные числа. Было бы хорошо, если бы такая функция помогла основной программе узнать, сколько было считано чисел, поэтому предусмотрим это для выхода. В конце концов, нужно подумать и о пользователе. Мы заставим функцию печатать сообщение, указывающее ее пределы, и осуществлять эхо-печать входной информации.

***main() и getarray()***

Вызовем нашу функцию getarray(), предназначенную для считывания. Мы определили эту функцию в терминах ввода и вывода и наметили в общих чертах схему на псевдокоде. Теперь давайте напишем функцию и покажем, как она включается в основную программу:

Сначала напишем main():

/\* сортировка \*/

#define MAXSIZE 100

/\* предельное количество сортируемых целых чисел \*/

main()

{

int numbers [MAXSIZE]; /\* массив для ввода \*/

int size; /\* количество вводимых чисел \*/

size = getarray(numbers, MAXSIZE); /\* запись чисел в массив \*/

sort(numbers, size); /\* сортировка массива \*/

print(numbers, size); /\* печать отсортированного массива \*/

Это общий вид программы. Функция getarray() размещает введенные числа в массиве numbers и выдает сообщение о том, сколько значений было считано; эта величина записывается в size. Затем идут функции sort() и print(), которые мы еще должны написать; они сортируют массив и печатают результаты. Включая в них size, мы облегчаем им работу и избавляем от необходимости выполнять самим подсчет. Мы также снабдили getarray() переменной MAXSIZE, которая сообщает размер массива, необходимого для запоминания.

Теперь, когда мы добавили size к передаваемой информации, нужно модифицировать рисунок нашего черного ящика, смотри рисунок.

11_06

Теперь рассмотрим функцию getarray():

/\* getarray(), использующая getint() \*/

#define STOP - 1 /\* признак EOF \*/

#define NONUM 1 /\* признак нецифровой строки \*/

#define YESNUM 0 /\* признак строки цифр \*/

getarray(array, limit);

int array [], limit;

{

int num, status;

int index = 0; /\* индекс массива \*/

printf (" Эта программа прекращает считывание чисел после %d значений\n" , limit);

printf (" или если введен символ EOF.\n" );

while(index < limit && (status = getint(&num)) != STOP)

{/\* прекращает считывание после достижения limit или EOF \*/

if (status == YESNUM)

{

array [index++ ] = num;

printf(" число %d принято. \n" , num);

}

else

if (status == NONUM)

printf (" Это было не целое число! Попытайтесь снова\n" );

else

printf (" Этого не может быть! Что-то неправильно. \n" );

if (index == limit) /\* сообщить, если массив заполнен \*/

printf (" Все %d элементов массива заполнены. \n" , limit);

return (index);

}

}

Это значительная часть программы, и у нас есть немало замечаний.

***Разъяснения***

Так как немного трудно вспомнить значение, скажем кода — 1, мы используем мнемонические символические константы для представления кодов состояния.

Применяя эти коды, мы создаем getarray() для управления каждым из возможных значений состояния. Состояние STOP вызывает прекращение цикла чтения, если getint() находит на своем «пути» EOF. Состояние YESNUM говорит о запоминании числа в предлагаемом массиве. Кроме того, отсылается «эхо-число» пользователю, чтобы он знал, что оно принято. Состояние NONUM предписывает пользователю попытаться выполнить задачу еще раз. (Это признак «дружелюбия»).

У нас есть еще оператор else. Единственный путь достижения этого оператора возможен, если getint() возвращает значение, отличное от -1, 0 или 1. Однако это единственные значения, которые могут быть возвращены, поэтому else является, по-видимому, бесполезным оператором. Почему он включен в программу? Мы вставили его как пример «защитного программирования», как способ защиты программы от будущей ошибки. Когда-нибудь мы (или кто-нибудь еще), может быть, решим обратиться к функции getint() и добавить в ее репертуар немного больше возможных значений состояния. Наиболее вероятно, что мы забудем (а они могут никогда не узнать), что getarray() предполагает только три возможных ответа. Поэтому мы включаем это последнее else, чтобы «поймать» любые новые ответы, которые появятся, и значительно упростить будущую отладку.

Размер массива устанавливается в main(). Поэтому мы не задаем его, когда описываем аргумент-массив в getarray(). Мы ставим только квадратные скобки в оператор, чтобы указать, что аргумент является массивом.

int numbers [MAXSIZE]; /\* размер задается в main \*/

int array[] /\* нет определения размера в вызвавшей функции \*/

Мы решили применить ключевое слово return для возврата числа прочитанных элементов. Таким образом, вызов нашей функции

size = getarray(numbers);

присваивает значение переменной size и дает значения массиву numbers.

Вы можете спросить, почему мы не использовали указатели в вызове

size = getarray(numbers);

ведь у нас функция изменяет значение чего-то (массива) в вызывающей программе? Ошибаетесь — мы использовали указатель! В языке Си имя массива является также указателем на первый элемент массива, т. е.

numbers == &numbers[0]

Когда функция getarray() создает массив array, то адрес элемента array[0] совпадает с адресом элемента numbers[0] и т. д. для всех других индексов. Поэтому все манипуляции, которые выполняет getarray() с array[], фактически выполняются с numbers[]. Мы будем более подробно говорить о связи между указателями и массивами позднее. Теперь же нам нужно усвоить, что функция воздействует на массив в вызывающей программе, если мы используем массив в качестве аргумента функции.

В функциях, содержащих счетчики и пределы, таких как getarray(), наиболее вероятным местом появления ошибок являются «граничные условия», где значения счетчиков достигают своих пределов. Мы собираемся прочитать максимальное количество чисел, указанное в MAXSIZE, или же мы намерены ограничиться одним? Хотим обратить внимание на детали, такие, как ++index или index++ и < или < =. Мы также должны помнить, что у массивов индексы начинаются с 0, а не с 1. Проверьте вашу программу и посмотрите, работает ли она так, как должна. Самое простое — предположить, что limit равен 1 и пройти по программе шаг за шагом.

Обычно наиболее трудной частью программы является обеспечение ее взаимодействия с пользователем удобным и надежным способом. Это относится и к нашей программе. Теперь, когда мы справились с getarray(), находим, что функция sort() должна быть проще и print() — тоже. Теперь перейдем к функции sort().

***Сортировка данных***

Рассмотрим еще раз функцию main():

11_07

main()

{

int numbers [MAXSIZE]; /\* массив для ввода \*/

int size; /\* количество введенных элементов \*/

size = getarray(numbers, MAXSIZE); /\* помешает ввод в массив \*/

sort(numbers, size); /\* сортировка массива \*/

printf(numbers, size); /\* печать отсортированного массива \*/

}

Мы видим, что функция sort() имеет на входе массив целых чисел, предназначенных для сортировки, и счетчик количества элементов, подлежащих сортировке. На выходе получается массив, содержащий отсортированные числа. Мы все еще не решили, как выполнять сортировку, поэтому мы должны дополнительно уточнить это описание.

Очевидно, в начале трудно определить направление сортировки. Собираемся ли мы вести сортировку от большего к меньшему, или наоборот? Мы свободны в выборе и допустим, что хотим сортировать от большего к меньшему. (Можно сделать программу, работающую любым из этих методов, но тогда нам нужно придумать способ сообщить ей о своем выборе.)

Рассмотрим теперь метод, который будем использовать для сортировки. В настоящее время разработано много алгоритмов сортировки; возьмем один из самых простых.

Вот наш план на псевдокоде:

от n = первому элементу до n = ближайшему к- последнему элементу находим самое большое из оставшихся чисел и помещаем его в n-ю позицию

Он выполняется примерно так. Сначала пусть n = 1. Мы просматриваем весь массив, находим самое большое число и помещаем его в первый элемент. Затем n = 2, и мы опять просматриваем весь массив, кроме первого элемента, находим самое большое из оставшихся чисел и помещаем его во второй элемент. Продолжаем этот процесс до тех пор, пока не достигнем ближайшего - к - последнему элементу. Теперь осталось только два элемента. Мы сравниваем эти числа и помещаем большее в элемент, ближайший - к - последнему. Оставшееся самое меньшее из всех чисел помещаем в последний элемент.

Это выглядит очень похоже на задачу с циклом for, но мы все же должны описать процесс «найти и поместить» более детально. Как сделать так, чтобы мы находили каждый раз самое большое из оставшихся чисел? Вот один способ. Сравните первый и второй

элементы оставшегося массива. Если второй больше, поменяйте их местами. Теперь сравните первый элемент с третьим. Если третий больше, поменяйте местами эти два. Каждый раз больший элемент перемещается вверх. Продолжаем таким образом, пока не сравним первый элемент с последним. Если вы дошли до конца, самое большое число теперь будет в первом элементе оставшегося массива. По существу мы имеем отсортированный массив для первого элемента, но остаток массива находится в беспорядке. На псевдокоде это можно выразить так:

для n = от второго до последнего элемента сравниваем n-й элемент с первым; если n-й больше, меняем их местами.

Это напоминает еще один цикл for. Его следует вставить в первый цикл for. Внешний цикл показывает, какой элемент массива должен быть заполнен, а внутренний цикл находит значение, которое следует поместить туда. Записывая обе части на псевдокоде и переводя их на язык Си, мы получаем следующую функцию:

/\* сортировка массива целых чисел в порядке убывания \*/

sortarray(array, limit)

int array[], limit;

int top, search;

{

for (top = 0; top < limit - 1; top++ )

for (search = top + 1; search < limit; search\*+ )

if (array[search] > array[top])

interchange( &array[search], &array[top] );

}

Мы помним, что первый элемент имеет индекс 0. Кроме того, раньше была создана функция обмена, поэтому мы использовали ее здесь. Так как функция interchange «работает» с двумя элементами массива, а не со всем массивом, мы должны использовать адреса только двух интересующих нас элементов. (В то время как имя array является указателем на весь массив, нам нужно применить операцию &, чтобы указывать на отдельные элементы.) Мы использовали top в качестве индекса для элемента массива, который следует заполнить, так как он является вершиной неотсортированной части массива. Индекс search перемещает по массиву в порядке убывания текущий элемент. Большинство текстов использует обозначения i и j для этих индексов, однако это осложняет ситуацию, если нужно посмотреть, что происходит.

Этот алгоритм иногда называют «пузырьковой сортировкой», так как самое большое значение медленно поднимется вверх.

Именно теперь мы должны написать функцию print().

***Печать данных***

11_08

Эта программа достаточно проста:

/\* печать массива \*/

print(array, limit)

int array[], limit;

{

int index;

for (index = 0; index <= limit; index++ )

printf(" %d\n", array[index]);

}

Если мы хотим сделать что-нибудь другое, например печатать по строкам, а не в столбец, можно всегда вернуться и изменить эту функцию, оставив другие функции неизменными. Аналогично, если мы нашли алгоритм сортировки, который нам больше нравится, можно заменить этот модуль программы. Это один из приятных моментов модульной программы.

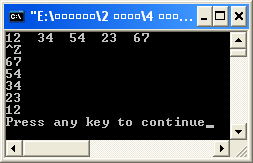
***Результаты***

Давайте скомпилируем и протестируем нашу программу сортировки. Чтобы упростить проверку граничных условий, временно изменим MAXSIZE на 5.

В нашем первом тесте будем снабжать программу числами до тех пор, пока она не откажется их принимать.

Эта программа прекращает считывание чисел после 5 значений,

или если встретился символ EOF.



Программа считала 5 чисел и отсортировала их. Теперь посмотрим, как будет выглядеть результат, если она остановится, встретив символ EOF.

Эта программа прекращает считывание чисел после 5 значений, или если встретился символ EOF.

456 928

-23 +16

Клавиша [control z] (передает EOF в нашу систему)

928

456

16

-23

Быстрее чем вы сможете сказать «экология это наука о домашнем хозяйстве», целый огромный массив отсортирован.

Успех! Это было не просто, но не невозможно. Разделив задачу на небольшие части и продумав, какой информацией должна обмениваться каждая из них, мы свели задачу к частям, поддающимся управлению. Кроме того, отдельные модули, которые мы создали, можно использовать как части подобных программ.

**Обзор**

Чего же мы достигли? Если посмотреть с практической стороны, то мы разработали генератор случайных чисел и программу сортировки целых чисел. При этом создали функцию getint(), которую можно использовать в других программах. Если посмотреть с педагогической точки зрения, то мы проиллюстрировали несколько основных принципов и концепций, полезных при создании программ.

Следует обратить внимание на самый существенный момент: программы: нужно проектировать, а не создавать их методом проб и ошибок. Вы должны внимательно подумать о форме и содержании ввода и вывода для программы. Необходимо разделить программу на хорошо определенные задачи, затем раздельно запрограммировать, принимая во внимание взаимодействие их друг с другом. Идея заключается в достижении модульности. Если необходимо, разбивайте модули на еще более мелкие модули. Используйте функции для повышения степени модульности и простоты программы.

При проектировании программы, попытайтесь предвидеть, что может идти неправильно и программируйте, исходя из этого. Используйте локализацию ошибок, чтобы контролировать действия в местах потенциальных затруднений, или по крайней мере предупреждать пользователя, что может возникнуть осложнение. Гораздо лучше дать пользователю еще одну возможность ввести данные, чем продолжать выполнять программу и прийти к аварийной ситуации.

Если создается функция, сначала определите, как она будет взаимодействовать с вызывающей программой. Решите также, какая информация будет входить в нее, а какая выходить. Какими должны быть аргументы? Хотите ли вы использовать указатели, или возврат, или то и другое? Если вы примете во внимание все эти параметры, то можете обратить внимание на работу самой функции.

Используйте эти идеи, и ваша программа будет более надежной и менее подверженной аварийным ситуациям. Вы получаете тело функции, которое можете применять в других программах. Программирование в таком случае потребует меньше времени. Вообще все это похоже на хороший рецепт здорового программирования.

Не забывайте о классах памяти. Переменные можно определять вне функции; в этом случае их называют внешними (или глобальными) и они доступны более чем для одной функции. Переменные, определенные внутри функции, являются локальными для нее и не известны другим функциям. Если можно, используйте автоматическую разновидность локальных переменных. Они охраняют переменные одной функции от воздействия других функций.

**Что вы должны были узнать.**

* Как представлять функцию: черный ящик с информационным потоком.
* Что такое «проверка ошибок» и почему эта процедура хороша.
* Алгоритм сортировки.
* Как заставить функцию изменять массив: function(array).
* Как преобразовать строку цифр в число.
* Классы памяти: auto, extern, static и register.
* Область действия каждого класса памяти.
* Какой класс памяти использовать: обычно auto.

**Вопросы и ответы.**

**Вопросы.**

Что может сделать наш алгоритм сортировки неэффективным?

Как следует изменить нашу программу сортировки, чтобы она сортировала в порядке возрастания, а не убывания?

Измените функцию print () таким образом, чтобы она печатала по 5 чисел в строке.

Как следует изменить функцию stoi(), чтобы обрабатывать строки, представляющие восьмеричные числа?

Какие функции «знают» каждую переменную из описанных ниже? Есть ли здесь какие-нибудь ошибки?

/\* файл1 \*/

int daisy;

main()

{

int lily;

}

petal()

{

extern int daisy, lily;

}

/\* файл2 \*/

static int lily; int rose,

stem()

{

int rose;

}

root()

{

extern int daisy;

}

**Ответы.**

Предположим, что вы сортируете 20 чисел. Программа производит 19 сравнений, чтобы найти одно самое большое число. Затем делается 18 сравнений, чтобы найти следующее самое большое. Вся информация, полученная во время первого поиска «забывается» за исключением самого большого числа, поставленного на первое место. Второе самое большое число можно временно поместить на место с номером 1, а затем при сортировке опустить вниз. Много сравнений, выполнявшихся в первый раз, повторяется второй, третий раз и т. д.

Замените

array[search] > array [top] на

array[search] < array[top]

/\* печать массива \*/

print(array, limit)

int array[], limit);

int index;

for ( index = 0; index < limit; index++ )

{

printf(" %10d" , array[index]);

if (index % 5 == 4)

printf(" \n" );

}

printf(" \n");

}

Во-первых, обеспечьте, чтобы разрешенные символы были только цифрами от 0 до 7. Во-вторых, умножайте на 8 вместо 10 каждый раз, когда обнаружите новую цифру.

daisy известна функции main() по умолчанию и функциям petal() и root() благодаря extern-описанию. Она не известна функции stem(), потому что они находятся в разных файлах. Первая lily локальна для main: ссылка на lily в petal() является ошибочной, потому что в каждом из этих файлов нет внешней lily.

Есть внешняя статическая lily, но она известна только функциям второго файла.

Первая, внешняя rose, известна функции root(), а функция stem() отменила ее своей собственной локальной rose.

**Упражнения.**

Некоторые пользователи, возможно испугаются, если их попросить ввести символ EOF.

* Модифицируйте getarray() и вызываемые ею функции так, чтобы использовать символ # вместо EOF.
* Модифицируйте затем их так, чтобы можно было использовать либо EOF, либо #.

Создайте программу, которая сортирует числа типа float.

Создайте программу, превращающую смешанный текст из прописных и строчных букв в текст, состоящий только из прописных букв.

Создайте программу, которая удваивает пробелы в тексте с одиночными пробелами.