Модульное программирование

Модульность — фундаментальный аспект всех успешно работающих крупных систем.

Б. Страуструп

1. ФУНКЦИИ
2. Директивы препроцессора
3. Области действия идентификаторов

Способом борьбы со сложностью любой задачи является ее разбиение на части. В C++ задача может быть разделена на более простые
с помощью функций, после чего программу можно рассматривать в более укрупненном виде — на уровне взаимодействия функций.
Разделение программы на функции позволяет избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы. Процесс отладки программы, содержащей функции, можно лучше структурировать. Часто используемые функции помещают в библиотеки. Таким образом создаются более простые в отладке и сопровождении программы.

Следующим шагом в повышении уровня абстракции программы является группировка функций и связанных с ними данных в отдельные файлы (модули), компилируемые раздельно. Получившиеся в результате компиляции объектные модули объединяются в исполняемую программу с помощью компоновщика. Разбиение на модули уменьшает время перекомпиляции и облегчает процесс отладки, скрывая несущественные детали за интерфейсом модуля и позволяя отлаживать программу по частям (или разными программистами).

Модуль содержит данные и функции их обработки. Другим модулям нежелательно иметь собственные средства обработки этих данных, они должны пользоваться для этого функциями первого модуля. Для того чтобы использовать модуль, нужно знать только его интерфейс, а не все детали его реализации.

Чем более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции —фрагмента кода в функции и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы. Модульность поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и раздельной компиляции (раздельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов. Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение. Любая программа на C++ состоит из функций, одна из которых должна иметь имя main (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*.

Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. Определение функции содержит, кроме объявления, тело функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([список_параметров ])[thr0w ( исключения)]
        { тело функции }
```

□ С помощью необязательного модификатора класс можно явно задать область видимости функции, используя ключевые слова extern и static:

■ extern — глобальная видимость во всех модулях программы (по умолчанию);

■ static — видимость только в пределах модуля, в котором определена функция.

□ Тип возвращаемого функцией значения может быть любым, кроме массива и функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, указывается тип v0id.

□ Список параметров определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. Для каждого параметра, передаваемого в функцию, указывается его тип и имя (в объявлении имена можно опускать).

□ Об исключениях, обрабатываемых функцией,... (*позже*)

В определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать. На имена параметров ограничений по соответствию не накладывается, поскольку функцию можно вызывать с различными аргументами, а в прототипах имена компилятором игнорируются (они служат только для улучшения читаемости программы).

Функцию можно определить как встроенную с помощью модификатора inline, который рекомендует компилятору вместо обращения к функции помещать ее код непосредственно в каждую точку вызова. Модификатор inline ставится перед типом функции. Он применяется для коротких функций, чтобы снизить накладные расходы на вызов (сохранение и восстановление регистров, передача управления).

Директива inline носит рекомендательный характер и выполняется компилятором по мере возможности. Использование inline-функций может увеличить объем исполняемой программы. Определение функции должно предшествовать ее вызовам, иначе вместо inline-расширения компилятор сгенерирует обычный вызов. Тип возвращаемого значения и типы параметров совместно определяют тип функции.

Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция. Если тип возвращаемого функцией значения не v0id, она может входить в состав выражений или, в частном случае, располагаться в правой части оператора присваивания.

Пример функции: *сумма двух целых величин*

```
#include <i0stream.h>
int sum(int a, int b); // объявление функции
int main(){ int a = 2, b = 3, c, d;
c = sum(a, b); // вызов функции
cin » d; c0ut « sum(c. d); // вызов функции
 return 0; }
int sum(int a, int b){ // определение функции
return (a + b); }
```

Пример функции, *выводящей на экран поля переданной ей структуры:*

```
#include <i0stream.h>
struct W0rker{
char fi0[30]; int date, c0de; d0uble salary; };
v0id print_w0rker(W0rker); //объявление функции
int main(){
W0rker staff[100]; ... /* формирование массива staff */
f0r (int i = 0; i<100; i++)print_w0rker(staff[i]); // вызов функции
return 0; }
v0id print_w0rker(W0rker w){ //определение функции
c0ut « w.fi0 « ' ' « w.date « ' ' « w.c0de « ' ' « w.salary « endl; }
```

Все величины, описанные внутри функции, а также ее параметры, являются локальными. Областью их действия является функция. При вызове функции, как и при входе в любой блок, в стеке выделяется память иод локальные автоматические переменные. Кроме того, в стеке сохраняется содержимое регистров процессора на момент, предшествующий вызову функции, и адрес возврата из функции для того, чтобы при выходе из нее можно было продолжить выполнение вызывающей функции.

При выходе из функции соответствующий участок стека освобождается, поэтому значения локальных переменных между вызовами одной и той же функции не сохраняются.

Если этого требуется избежать, при объявлении локальных переменных используется модификатор static:

```
#include <i0stream.h>
v0id f(int a){
int m = 0;
c0ut « "n m p\n";
while (a--){
static int n = 0; int p = 0;
c0ut « n++ « ' ' « m++ « ' ' « p++ « '\n';
}
}
int main(){ f(3); f(2): return 0;}
```

Статическая переменная n размещается в сегменте данных и инициализируется один раз при первом выполнении оператора, содержащего ее определение. Автоматическая неременная m инициализируется при каждом входе в функцию. Автоматическая переменная p инициализируется при каждом входе в блок цикла. Программа выведет на экран:

```
n m p
0 0 0
1 1 0
2 2 0
n m p
3 0 0
4 1 0
```

При совместной работе функции должны обмениваться информацией. Это можно осуществить с помощью глобальных переменных, через параметры и через возвращаемое функцией значение.

Глобальные переменные

Глобальные переменные видны во всех функциях, где не описаны локальные переменные с теми же именами, поэтому использовать их для передачи данных между функциями очень легко. Тем не менее это не рекомендуется, поскольку затрудняет отладку программы и препятствует помещению функций в библиотеки общего пользования. Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

Возвращаемое значение

Механизм возврата из функции в вызвавшую ее функцию реализуется оператором return [выражение]; Функция может содержать несколько операторов return (это определяется потребностями алгоритма). Если функция описана как void, выражение не указывается. Оператор return можно опускать для функции типа void, если возврат из нее происходит перед закрывающей фигурной скобкой, и для функции main. (Тогда при компиляции примеров выдается предупреждение.) Выражение, указанное после return, неявно преобразуется к типу возвращаемого функцией значения и передается в точку вызова функции.

Примеры:

```
int fl(){return 1;} //правильно
void f2(){return 1;} // неправильно. f2 не должна возвращать значение
double f3(){return 1;} // правильно. 1 преобразуется к типу double
```

Нельзя возвращать из функции указатель на локальную переменную, поскольку память, выделенная локальным переменным при входе в функцию, освобождается после возврата из нее.
Пример:

```
int* f(){
int a = 5;
return &a; // нельзя!
}
```

Параметры функции

Механизм параметров является основным способом обмена информацией между вызываемой и вызывающей функциями. Параметры, перечисленные в заголовке описания функции, называются формальными параметрами, или просто параметрами, а записанные в операторе вызова функции — фактическими параметрами, или аргументами. При вызове функции в первую очередь вычисляются выражения, стоящие на месте аргументов; затем в стеке выделяется память под формальные параметры функции в соответствии с их типом, и каждому из них присваивается значение соответствующего аргумента. При этом проверяется соответствие типов и при необходимости выполняются их преобразования. При несоответствии типов выдается диагностическое сообщение.

Существует два способа передачи параметров в функцию: по значению и по адресу. *При передаче по значению* в стек заносятся копии значений аргументов, и операторы функции работают с этими копиями. Доступа к исходным значениям параметров у функции нет, а, следовательно, нет и возможности их изменить. *При передаче по адресу* в стек заносятся копии адресов аргументов, а функция осуществляет доступ к ячейкам памяти по этим адресам и может изменить исходные значения аргументов:

```
#include <iostream.h>
void f(int i, int* j, int& k);
int main(){
int i = 1, j = 2, k = 3;
cout <<"i j k\n"; cout « i «' '« j «' '« k «'\n‘;
f(i, &j, k);
cout « i «' '« j «' '« k;
return 0; }
void f(int I, int* j, int& k){ i++; (*j)++; k++; }
```

Результат работы программы:

i j k
1 2 3
1 3 4

Первый параметр (i) передается по значению.

Его изменение в функции не влияет на исходное значение.

Второй параметр (j) передается по адресу с помощью указателя, при этом для передачи в функцию адреса фактического параметра используется операция взятия адреса, а для получения его значения в функции требуется операция разыменования.

Третий параметр (к) передается по адресу с помощью ссылки.

При передаче по ссылке в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются. Поэтому использование ссылок вместо указателей улучшает читаемость программы, избавляя от необходимости применять операции получения адреса и разыменования. Использование ссылок вместо передачи по значению более эффективно, поскольку не требует копирования параметров, что имеет значение при передаче структур данных большого объема. Если требуется запретить изменение параметра внутри функции, используется модификатор const:

```
int f(const char*);
char* t(char* a, const int* b);
```

Таким образом, исходные данные, которые не должны изменяться в функции, предпочтительнее передавать ей с помощью константных ссылок.

По умолчанию параметры любого типа, кроме массива и функции (например, вещественного, структурного, перечисление, объединение, указатель), передаются в функцию по значению.

Передача массивов в качестве параметров

При использовании в качестве параметра массива в функцию передается указатель на его первый элемент, иными словами, массив всегда передается по адресу. При этом информация о количестве элементов массива теряется, и следует передавать его размерность через отдельный параметр (в случае массива символов, то есть строки, ее фактическую длину можно определить по положению нуль-символа).

```
#include <iostream.h>
int sum(const int* mas, const int n);
int const n = 10;
int main(){
int marks[n] = {3. 4. 5. 4, 4};
cout « "Сумма элементов массива: " « sum(marks, n);
return 0; }
int sum(const int* mas, const int n){
    // варианты: int sum(int mas[]. int n)
    // или int sum(int mas[n], int n)
    // (величина n должна быть константой)
int s = 0; for (int i = 0; i<n; i++) s += mas[i];
return s; }
```

При передаче многомерных массивов все размерности, если они не известны на этапе компиляции, должны передаваться в качестве параметров. Внутри функции массив интерпретируется как одномерный, а его индекс пересчитывается в программе.

Пример: *с помощью функции подсчитывается сумма элементов двух двумерных массивов.* Размерность массива b известна на этапе компиляции, под массив a память выделяется динамически:

```
#include <stdio.h>
#include <stdlib.h>
int sum(const int *a, const int nstr, const int nstb);
int main(){ int b[2][2] = {{2, 2}, {4, 3}};
printf("Сумма элементов b: %d\n“, sum(&b[0][0], 2, 2));
// имя массива передавать в sum нельзя
//из-за несоответствия типов
int i, j, nstr, nstb, *a;
printf("Введите количество строк и столбцов: \n");
scanf(%d%d“, &nstr, &nstb);
a = (int *)malloc(nstr * nstb * sizeof(int));
for (i = 0; i<nstr; i++) for (j = 0;j<nstb; j++)
scanf("*d“, &a[i * nstb + j]);
printf(“Сумма элементов a: %d\n”, sum(a, nstr, nstb));
return 0;}
int sum(const int *a, const int nstr, const int nstb){
int i, j, s = 0;
for (i = 0; i<nstr; i++)   for (j = 0; j<nstb; j++)
s += a[i * nstb + j];
return s; }
```

Для работы с двумерным массивом естественным образом, можно применить альтернативный способ выделения памяти:

```
#include <iostream.h>
int sum(int **a, const int nstr, const int nstb);
int main(){ int nstr, nstb; cin » nstr » nstb; int **a, i, j;
// Формирование матрицы a:
a = new int* [nstr];
for (i = 0; i<nstr; i++) a[i] = new int [nstb];
 for (i = 0; i<nstr; i++) for (j = 0; j<nstb; j++) cin » a[i][j];
cout « sum(a, nstr, nstb); return 0; }
int sum(int **a, const int nstr, const int nstb){
int i, j, s= 0;
for (i = 0; i<nstr; i++) for (j = 0; j<nstb; j++)
s += a[i][j]; return s; }
```

В этом случае память выделяется в два этапа: сначала под столбец указателей на строки матрицы, а затем в цикле под каждую строку.

Передача имен функций в качестве параметров

Функцию можно вызвать через указатель на нее. Для этого объявляется указатель соответствующего типа и ему с помощью операции взятия адреса присваивается адрес функции:

```
void f(int a ){/*...*/ } // определение функции
void (*pf)(int); // указатель на функцию
pf = &f; // указателю присваивается адрес функции
          // (можно написать pf = f;)
pf(10);  // функция f вызывается через указатель pf
         // (можно написать (*pf)(10) )
```

Для того чтобы сделать программу легко читаемой, при описании указателей на функции используют переименование типов (typedef). Можно объявлять массивы указателей на функции (это может быть полезно, например, при реализации меню):

```
// Описание типа PF как указателя
// на функцию с одним параметром типа int:
typedef void (*PF)(int);
// Описание и инициализация массива указателей:
PF menu[] = {&new, &open, &save};
menu[1](10); // Вызов функции open
```

Здесь new, open и save — имена функций, которые должны быть объявлены ранее.

Указатели на функции передаются в подпрограмму таким же образом, как и параметры других типов:

```
#include <iostream.h>
typedef void (*PF)(int);
void fl(PF pf){
// функция fl получает параметром указатель типа PF
pf(5); // вызов функции, переданной через указатель
}
void f(int i ){cout « i;}
int main(){
 fl(f);
return 0;
}
```

Тип указателя и тип функции, которая вызывается посредством этого указателя, должны совпадать в точности.

Параметры со значениями по умолчанию

Чтобы упростить вызов функции, в ее заголовке можно указать значения параметров по умолчанию. Эти параметры должны быть последними в списке и могут опускаться при вызове функции. Если при вызове параметр опущен, должны быть опущены и все параметры, стоящие за ним. В качестве значений параметров по умолчанию могут использоваться константы, глобальные переменные и выражения:

```
int f(int a, int b = 0);
void f1(int, int = 100, char* = 0);
/* обратите внимание на пробел между * и =
(без него - операция сложного присваивания *=) */
void err(int errValue = errno); // errno - глобальная переменная
•••
f(100); f(a, 1); // варианты вызова функции f
f1(a); f1(a, 10); f1(a, 10, "Vasia"); // варианты вызова функции f1
f1(a,,"Vasia"); // неверно!
```

Функции с переменным числом параметров

Если список формальных параметров функции *заканчивается многоточием*, это означает, что при ее вызове на этом месте можно указать еще несколько параметров. Проверка соответствия типов для этих параметров не выполняется, char и short передаются как int, а float — как double.

В качестве примера можно привести функцию printf, прототип которой имеет вид: int printf (const char*, ...);

Это означает, что вызов функции должен содержать по крайней мере один параметр типа char* и может либо содержать, либо не содержать другие параметры:

```
printf("Введите исходные данные"); // один параметр
printf("Сумма: %5.2f рублей", sum); // два параметра
printf(“%d %d %d %d", a, b, c, d); // пять параметров
```

Для доступа к необязательным параметрам внутри функции используются макросы библиотеки vastart, vaarg и vaend, находящиеся в заголовочном файле <stdarg.h>.

Поскольку компилятор не имеет информации для контроля типов, вместо функций с переменным числом параметров предпочтительнее пользоваться параметрами по умолчанию или перегруженными функциями, хотя можно представить случаи, когда переменное число параметров является лучшим решением.

Рекурсивные функции

функция, которая вызывает саму себя. Такая рекурсия называется прямой. Существует еще косвенная рекурсия, когда две или более функций вызывают друг друга. Если функция вызывает себя, в стеке создается копия значений ее параметров, как и при вызове обычной функции, после чего управление передается первому исполняемому оператору функции. При повторном вызове этот процесс повторяется. Ясно, что для завершения вычислений каждая рекурсивная функция должна содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата. При завершении функции соответствующая часть стека освобождается, и управление передается вызывающей функции, выполнение которой продолжается с точки, следующей за рекурсивным вызовом.

Классический пример - вычисление факториала (это не означает, что факториал следует вычислять именно так). Для того чтобы получить значение факториала числа n, требуется умножить на n факториал числа (n-1). Известно также, что 0!=1 и 1!=1.

```
long fact(long n){
    if (n==0 || n==1) return 1;
    return (n * fact(n - 1));
}
```

То же самое можно записать короче:

```
long fact(long n){
    return (n>1) ? n * fact(n - 1) : 1;
}
```

Рекурсивные функции чаще всего применяют для компактной реализации рекурсивных алгоритмов, а также для работы со структурами данных, описанными рекурсивно, например, с двоичными деревьями. Любую рекурсивную функцию можно реализовать без применения рекурсии, для этого программист должен обеспечить хранение всех необходимых данных самостоятельно. Достоинством рекурсии является компактная запись, а недостатками — расход времени и памяти на повторные вызовы функции и передачу ей копий параметров, и, главное, опасность переполнения стека.

Перегрузка функций

Часто бывает удобно, чтобы функции, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Если это имя мнемонично, то есть несет нужную информацию, это делает программу более понятной, поскольку для каждого действия требуется помнить только одно имя. Использование нескольких функций с одним и тем же именем, но с различными типами параметров, называется *перегрузкой функций*.

Компилятор определяет, какую именно функцию требуется вызвать, по типу фактических параметров. Этот процесс называется разрешением перегрузки (перевод английского слова resolution в смысле «уточнение»). Тип возвращаемого функцией значения в разрешении не участвует.

Механизм разрешения основан на достаточно сложном наборе правил, смысл которых сводится к тому, чтобы использовать функцию с наиболее подходящими аргументами и выдать сообщение, если такой не найдется. Допустим, имеется четыре варианта функции, определяющей наибольшее значение:

```
// Возвращает наибольшее из двух целых:
      int max(int, int);
// Возвращает подстроку наибольшей длины:
      char* max(char*, char*);
// Возвращает наибольшее, из первого параметра и длины второго:
      int max (int, char*);
// Возвращает наибольшее из второго параметра и длины первого:
      int max (char*, int);
void f(int a, int b, char* c, char* d){
      cout « max (a, b) « max(c, d) « max(a, c) « max(c, b);
}
```

При вызове функции max компилятор выбирает соответствующий типу фактических параметров вариант функции (в приведенном примере будут последовательно вызваны все четыре варианта функции). Если точного соответствия не найдено, выполняются продвижения порядковых типов в соответствии с общими правилами, например, bool и char в int, float в double и т. п. Далее выполняются стандартные преобразования типов, например, int в double или указателей в void*. Следующим шагом является выполнение преобразований типа, заданных пользователем, а также поиск соответствий за счет переменного числа аргументов функций. Если соответствие на одном и том же этапе может быть получено более чем одним способом, вызов считается неоднозначным и выдается сообщение об ошибке.

Неоднозначность может появиться при:

□ преобразовании типа;

□ использовании параметров-ссылок;

□ использовании аргументов по умолчанию.

Пример *неоднозначности при преобразовании типа:*

```
#include <iostream.h>
float f(float i){ cout « "function float f(float i)" « endl;return i; }
double f(double i){
cout « "function double f(double i)" « endl;
return i*2; }
int main(){
float x = 10.09; double y = 10.09;
cout « f(x) « endl; // Вызывается f(float)
cout « f(y) « endl; // Вызывается f(double)
/* cout « f(10) « endl; Неоднозначность - как преобразовать 10;
во float или double? */
return 0; }
```

Для устранения этой неоднозначности требуется явное приведение типа для константы 10.

Пример неоднозначности при использовании параметров-ссылок: если одна из перегружаемых функций объявлена как int f(int a, int b), а другая — как int f (int a, int &b), то компилятор не сможет узнать, какая из этих функций вызывается, так как нет синтаксических различий между вызовом функции, которая получает параметр по значению, и вызовом функции, которая получает параметр по ссылке.

Пример неоднозначности при использовании аргументов по умолчанию:

```
#include <iostream.h>
int f(int a){return a;}
int f(Int a, int b = I){return a * b;}
int main(){ cout « f(10, 2); // Вызывается f(int, int)
/* cout « f(10): Неоднозначность - что вызывается: f(int, int)
или f(int) ? */
return 0; }
```

Правила описания перегруженных функций.

□ Перегруженные функции должны находиться в одной области видимости, иначе произойдет сокрытие аналогично одинаковым именам переменных во вложенных блоках.

□ Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию.

□ Функции не могут быть перегружены, если описание их параметров отличается только модификатором const или использованием ссылки (например, int и const int или int и int&).

Шаблоны функций

Многие алгоритмы не зависят от типов данных, с которыми они работают (классический пример — сортировка). Естественно желание параметризовать алгоритм таким образом, чтобы его можно было использовать для различных типов данных. Первое, что может прийти в голову — передать информацию о типе в качестве параметра (например, одним параметром в функцию передается указатель на данные, а другим — длина элемента данных в байтах). Использование дополнительного параметра означает генерацию дополнительного кода, что снижает эффективность программы, особенно при рекурсивных вызовах и вызовах во внутренних циклах; кроме того, отсутствует возможность контроля типов. Другим решением будет написание для работы с различными тинами данных нескольких перегруженных функций, но в таком случае в программе будет несколько одинаковых по логике функций, и для каждого нового типа придется вводить новую.

В C++ есть мощное средство параметризации — шаблоны. Существуют шаблоны функций и шаблоны классов. С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тип данных передается функции в виде параметра на этапе компиляции. Компилятор автоматически генерирует правильный код, соответствующий переданному типу. Таким образом, создается функция, которая автоматически перегружает сама себя и при этом не содержит накладных расходов, связанных с параметризацией. Формат простейшей функции-шаблона:

```
template <class Type> заголовок{
    /* тело функции */
}
```

Вместо слова Type может использоваться произвольное имя.

В общем случае шаблон функции может содержать несколько параметров, каждый из которых может быть не только типом, но и просто переменной, например:

```
template <class A, class B, int i> void f(){ ... }
```

Например, функция, сортирующая методом выбора массив из n элементов любого типа, в виде шаблона может выглядеть так:

```
template <class Type> void sort_vybor(Type *b, int n){
 Type a; //буферная переменная для обмена элементов
 for (int i = 0; i<n-l: i++){
        int imin = i;
        for (int j = i + 1; j<n; j++)
            if (b[j] < b[imin]) imin = j;
            a = b[i]; b[i] = b[imin]; b[imin] = a;
        }
}
```

Главная функция программы, вызывающей эту функцию-шаблон, может иметь вид:

```
#include <iostream.h>
template <class Type> void sort_vybor(Type *b, int n);
int main(){ const int n = 20; int I, b[n];
for (i = 0; i<n; i++) cin » b[i];
sort_vybor(b, n); // Сортировка целочисленного массива
for (i = 0; i<n; i++) cout « b[i] « ' ';
cout « endl;
double a[] = {0.22, 117, -0.08, 0.21, 42.5};
sort_vybor(a, 5); // Сорт. массива вещественных чисел
for (i = 0; i<5; i++) cout « a[i] « ' '; return 0; }
```

Первый же вызов функции, который использует конкретный тип данных, приводит к созданию компилятором кода для соответствующей версии функции. Этот процесс называется инстанцированием шаблона (instantiation). Конкретный тип для инстанцирования либо определяется компилятором автоматически, исходя из типов параметров при вызове функции, либо задается явным образом.

При повторном вызове с тем же типом данных код заново не генерируется. На месте параметра шаблона, являющегося не типом, а переменной, должно указываться константное выражение.

Пример явного задания аргументов шаблона при вызове:

```
 template<class X, class Y, class Z> void f(Y, Z);
void g(){ f<int, char*, double>(“Vasia“, 3.0);
f<int, char*>("Vasia“, 3.0); // Z определяется как double
f<int>("Vasia“, 3.0);
// Y определяется как char*, а Z - как double
 // f(“Vasia“, 3.0); ошибка; X определить невозможно
}
```

Чтобы применить функцию-шаблон к типу данных, определенному пользователем (структуре или классу), требуется перегрузить операции для этого типа данных, используемые в функции.

Как и обычные функции, шаблоны функций могут быть *перегружены как с помощью шаблонов, так и обычными функциями*.

Можно предусмотреть специальную обработку отдельных параметров и типов с помощью специализации шаблона функции. Допустим, требуется более эффективно реализовать общий алгоритм сортировки для целых чисел. В этом случае можно «вручную» задать вариант шаблона функции для работы с целыми числами:

```
void sort_vibor<int>(int *b, int n){
 ... // Тело специализированного варианта функции
 }
```

Сигнатура шаблона функции включает не только ее тип и типы параметров, но и фактический аргумент шаблона. Обычная функция никогда не считается специализацией шаблона, несмотря на то, что может иметь то же имя и тип возвращаемого значения.

Функция main()

Функция, которой передается управление после запуска программы, должна иметь имя main. Она может возвращать значение в вызвавшую систему и принимать параметры из внешнего окружения.

Возвращаемое значение должно быть целого типа.

Стандарт предусматривает два формата функции:

```
// без параметров:
тип main(){ /* ... */ }
// с двумя параметрами:
тип main(int argc, char* argv[ ]){ /* ... */ }
```

При запуске программы параметры разделяются пробелами. Имена параметров в программе могут быть любыми, но принято использовать argc и argv.

Первый параметр (argc) определяет количество параметров, передаваемых функции, включая имя самой программы, второй параметр (argv) является указателем на массив указателей типа char*. Каждый элемент массива содержит указатель на отдельный параметр командной строки, хранящийся в виде С-строки, оканчивающейся нуль-символом.

Первый элемент массива (argv[0]) ссылается на полное имя запускаемого на выполнение файла, следующий (argv[1]) указывает на первый параметр, argv[2] — на второй параметр, и так далее. Параметр argv[argc] должен быть равен 0. Если функция main() ничего не возвращает, вызвавшая система получит значение, означающее успешное завершение. Ненулевое значение означает аварийное завершение. Оператор возврата из main() можно опускать.

```
#include <iostream.h>
void main(int argc, char* argv[]){
 for (int i = 0; i<argc; i++) cout « argv[i] « '\n';
}
```

Пусть исполняемый файл программы имеет имя main.exe и вызывается из командной строки:

```
d:\cpp\main.exe one two three
```

На экран будет выведено:

```
D:\CPP\MAIN.EXE one two three
```

Функции стандартной библиотеки

Любая программа на C++ содержит обращения к стандартной библиотеке, в которой находятся определения типов, констант, макросов, функций и классов. Чтобы использовать их в программе, требуется с помощью директивы #include включить в исходный текст программы заголовочные файлы, в которых находятся соответствующие объявления. Сами библиотечные функции хранятся в скомпилированном виде и подключаются к программе на этапе компоновки. В программах на C++ могут использоваться функции, унаследованные от библиотеки C.

Функции библиотеки можно разбить на группы по их назначению: ввод/вывод, обработка строк, математические функции, работа с динамической памятью, поиск и сортировка и т. д.

Функции ввода/вывода

Ввод/вывод в C++ реализуется либо с помощью функций, унаследованных от библиотеки C, либо с помощью потоков C++. Смешивать эти два способа в одной программе можно только синхронизировав ввод с помощью функции sync_with_stdio(). Каждый способ имеет свои преимущества. Преимущество использования потоков в том, что они легче в использовании в простых случаях ввода/вывода, не требующих форматирования, а, главное, потоковые операции можно переопределить для собственных классов. Ввод/вывод в стиле C удобнее использовать при форматированном выводе в программах, не использующих объектно-ориентированную технику.

Для использования функций ввода/вывода в стиле C необходимо подключить к программе заголовочный файл <stdio.h> или <cstdio>. При вводе/выводе данные рассматриваются как поток байтов. Физически поток представляет собой файл или устройство (например, клавиатуру или дисплей, рассматривающиеся как частный случай файла).

Открытие потока

Работа с потоком начинается с его открытия. Поток можно открыть для чтения и/или записи в двоичном или текстовом режиме. Функция открытия потока имеет формат:

```
FILE* fopen(const char* filename, const char* mode);
```

При успешном открытии потока функция возвращает указатель на предопределенную структуру типа FILE, содержащую всю необходимую для работы с потоком информацию, или NULL в противном случае.
Первый параметр — имя открываемого файла в виде С-строки, второй — режим открытия файла:
“r" — файл открывается для чтения;
"w" — открывается пустой файл для записи (если файл существует, он стирается);
"a" — файл открывается для добавления информации в его конец;
"r+" — файл открывается для чтения и записи (файл должен существовать);
"w+" — открывается пустой файл для чтения и записи (если файл существует, он стирается);
"a+" — файл открывается для чтения и добавления информации в его конец.

Режим открытия может также содержать символы t (текстовый режим) или b (двоичный режим), отличающиеся обработкой символов перехода на новую строку.

По умолчанию файл открывается в текстовом режиме, при котором комбинация символов «возврат каретки» и «перевод строки» (0x13 0x10) при вводе преобразуются в одиночный символ перевода строки (при выводе выполняется обратное преобразование). В двоичном режиме эти преобразования не выполняются.

Пример:

```
FILE *f = fopen("d:\\cpp\\data\ "rb+");
```

Указатель f используется в дальнейших операциях с потоком. Его передают функциям ввода/вывода в качестве параметра. При открытии потока с ним связывается область памяти, называемая буфером. При выводе вся информация направляется в буфер и накапливается там до заполнения буфера или до закрытия потока. Чтение осуществляется блоками, равными размеру буфера, и данные читаются из буфера. Буферизация позволяет более быстро и эффективно обмениваться информацией с внешними устройствами. Следует иметь в виду, что при аварийном завершении программы выходной буфер может быть не выгружен, и возможна потеря данных.

С помощью функций setbuf и setvbuf можно управлять размерами и наличием буферов.

Существует пять предопределенных потоков, которые открываются в начале работы программы:

1. стандартный ввод stdin,
2. стандартный вывод stdout,
3. стандартный вывод сообщений об ошибках stderr,
4. стандартный дополнительный поток stdaux,
5. стандартная печать stdprn.

Первые три потока по умолчанию относятся к консоли. Эти указатели можно использовать в любой функции ввода/вывода там, где требуется указатель потока.

Ввод/вывод в поток

Ввод/вывод в поток можно осуществлять различными способами: в виде последовательности байтов, в виде символов и строк или с использованием форматных преобразований. Для каждого вида операций определен свой набор функций.

Операции ввода/вывода выполняются начиная с текущей позиции потока, определяемой положением указателя потока. Указатель устанавливается при открытии на начало или конец файла (в соответствии с режимом открытия) и изменяется автоматически после каждой операции ввода/вывода. Текущее положение указателя можно получить с помощью функций ftell и fgetpos и задать явным образом с помощью функций fseek и fsetpos. Эти функции нельзя использовать для стандартных потоков.

Функции ввода/вывода

□ Чтение и запись потока байтов выполняют функции fread и fwrite.

□ Чтение символа из потока — getc, fgetc, из стандартного потока stdin — getchar.

□ Запись символа в поток — putc, fputc, в стандартный поток stdout — putchar.

□ Чтение строки из потока — fgets, из стандартного потока stdin — gets.

□ Запись строки в поток — fputs, в стандартный поток stdout — puts.

□ Форматированный ввод из потока — fscanf, из стандартного потока stdin — scanf, из строки — sscanf.

□ Форматированный вывод в поток — fprintf, в стандартный поток stdout — printf, в строку — sprintf.

Закрытие потока

при завершении программы,
явным образом с помощью функции fclose:
int fclose(FILE*); Перед закрытием информация из связанных с ним буферов выгружается на диск.
Рекомендуется явным образом закрывать потоки, открытые для записи, чтобы избежать потери данных.

Обработка ошибок

Функции работы с потоком возвращают значения, которые рекомендуется анализировать в программе и обрабатывать ошибочные ситуации. При работе с файлами часто используются функции feof и ferror:
int feof (FILE*) возвращает не равное нулю значение, если достигнут конец файла, в противном случае 0;
int ferror (FILE*) возвращает не равное нулю значение, если обнаружена ошибка ввода/вывода, в противном случае 0.

Пример работы с потоками

В файле хранятся сведения о мониторах. В каждой строке указан тип, оптовая и розничная цены и примечание. Для простоты данные в каждой строке записаны единообразно:

первые 20 символов занимает тип монитора,

далее по 5 символов целые числа, представляющие оптовую и розничную цены,

затем примечание длиной не более 40 символов.

Программа построчно считывает данные из текстового файла в буферную переменную s, затем формирует из них структуру mon и записывает ее в двоичном режиме в выходной файл.

Пример считывания из этого файла произвольной записи.

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(){
FILE *fi, *fo;
if((fi = fopen("d:\\c\\file.txt", "r")) == 0){
cout « "Ошибка открытия входного файла";
return 1;}
if ((fo = fopen("d:\\c\\binfile.out“, "w+b")) == 0){
cout « "Ошибка открытия выходного файла": return 1;}
const int dl = 80;
char s[dl];
struct{
char type[20]:
int opt, rozn;
char comm[40]; }mon;
int kol =0; // Количество записей в файле
```

```
while (fgets(s, dl, fi)){
// Преобразование строки в структуру:
strncpy(mon.type, s, 19);
// Описание strncpy?
 mon.type[19]='\0';
mon.opt = atoi(&s[20]);
// Описание atoi ?
 mon.rozn = atoi(&s[25]);
strncpy(mon.comm, &s[30], 40);
fwrite(&mon, sizeof mon, 1, fo);
kol++; }
fclose(fi);
int i; cin » i: // Номер записи
if (i >= kol){cout « "Запись не существует"; return 1;}
// Установка указателя текущей позиции файла на запись i:
fseek(fo, (sizeof mon)*I, SEEK_SET);
fread(&mon, sizeof mon, 1, fo);
cout « "mon.type " « mon.type « " opt " « mon.opt « " rozn " «
mon.rozn « endl; fclose(fo); return 0; }
```

Функции работы со строками и символами

Строка представляет собой массив символов, заканчивающийся нуль-символом. В C++ есть две возможности работы со строками: функции, унаследованные из библиотеки C (заголовочный файл <string.h> или <cstring>), и библиотечный класс C++ string, предоставляющий более широкие возможности представления, обработки и контроля строк.

Библиотека C содержит функции копирования строк (strcpy, strncpy), сравнения (strcmp, strncmp), объединения строк (strcat, strncat), поиска подстроки (strstr), поиска вхождения символа (strchr, strrchr, strpbrk), определения длины строки (strlen) и другие.

В заголовочных файлах <stdlib.h> и <cstdlib> содержатся полезные функции преобразования строк в числа (обратные преобразования можно сделать с помощью функции sprintf):

```
double atof(const char* p); //преобразует переданную строку в double;
int atoi(const char* p); //преобразует переданную строку в int;
long atol(const char* p); //преобразует переданную строку в long.
```

Пробелы и табуляции в начале строки пропускаются. Преобразование прекращается при встрече недопустимого символа или конца строки. Если строку нельзя преобразовать в число, возвращается 0. Если число выходит за пределы диапазона данного типа, переменной errno (заголовочный файл <cerrno>) присваивается значение ERANGE и возвращается допустимое число.

Пример (заполняется массив типа double из строки):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(){
char s[ ] = "2, 38.5, 70, 0, 0, 1“; *p = s;
double m[10];
int i = 0;
do{ m[i++] = atof(p); if (i>9)break;
}while(p = strchr(p, ‘,’), p++);
for( int k = 0; k<i; k++) printf(“%5.2f “, m[k]);
return 0; }
```

Для работы с символами в стандартной библиотеке (заголовочные файлы <ctype. h> и <cctype>) есть следующие функции:

Имя	Проверка на принадлежность символа множеству
isalnum	букв и цифр (A–Z, a–z, 0–9)
isalfa	букв (A–Z, a–z)
iscntrl	управляющих символов (с кодами 0..31 и 127)
isdigit	цифр (0–9)
isgraph	печатаемых символов, кроме пробела (isalfa | isdigit | ispunct)
islower	букв нижнего регистра (a–z)
isprint	печатаемых символов
ispunct	знаков пунктуации
isspace	символов-разделителей
isupper	букв верхнего регистра (A–Z)
isxdigit	шестнадцатеричных цифр (A–F, a–f, 0–9)

Функции принимают величину типа int и возвращают значение true, если условие выполняется. Рекомендуется пользоваться стандартными функциями, а не писать собственные циклы проверки, так как это снижает количество ошибок в программе.

Кроме описанных выше, в библиотеке есть функции tolower и toupper, переводящие символ латинского алфавита соответственно в нижний и верхний регистр. Для каждой из перечисленных функций есть ее аналог для многобайтных символов типа wchar_t, содержащий в названии букву w.

Математические функции

C++ унаследовал из C стандартные математические функции, описание которых находится в заголовочных файлах <math.h> (<cmath>). Они позволяют получить абсолютное значение (abs, fabs), округленное число (ceil, floor), квадратный корень (sqrt), степень (pow), значения тригонометрических функций (sin, cos, tan, sinh, cosh, tanh, asin, acos, atan, atan2), экспоненту (exp), логарифм (log, log10), дробную и целую части числа (modf), остаток от деления (fmod) и другие. Ошибки индицируются установкой errno из <errno.h> (<cerrno>) в значение EDOM для ошибок, связанных с выходом из области определения, и ERANGE для ошибок выхода за пределы диапазона.

Директивы препроцессора

Препроцессором называется первая фаза компилятора. Инструкции препроцессора называются директивами. Они должны начинаться с символа #, перед которым в строке могут находиться только пробельные символы

Директива #include

#include <имя_файла>

вставляет содержимое указанного файла в ту точку исходного файла, где она записана. Включаемый файл также может содержать директивы #include. Поиск файла, если не указан полный путь, ведется в стандартных каталогах включаемых файлов. Вместо угловых скобок могут использоваться кавычки (" ") — в этом случае поиск файла ведется в каталоге, содержащем исходный файл, а затем уже в стандартных каталогах.

Директива #include является простейшим средством обеспечения согласованности объявлений в различных файлах, она включает в них информацию об интерфейсе из заголовочных файлов. Заголовочные файлы обычно имеют расширение .h и могут содержать:

- определения типов, констант, встроенных функций, шаблонов, перечислений;
- объявления функций, данных, имен, шаблонов;
- пространства имен;
- директивы препроцессора;
- комментарии.

В заголовочном файле не должно быть определений функций и данных. Эти правила не являются требованием языка, а отражают разумный способ использования директивы. При указании заголовочных файлов стандартной библиотеки расширение .h можно опускать. Это сделано для того, чтобы не ограничивать способы их хранения. Для каждого файла библиотеки C с именем <name.h> имеется соответствующий файл библиотеки C++ <cname>, в котором те же средства описываются в пространстве имен std. Например, директива #include <cstdio> обеспечивает те же возможности, что и #include <stdio.h>, но при обращении к стандартным функциям требуется указывать имя пространства имен std.

Директива #define

Директива #define определяет подстановку в тексте программы. Она используется для определения:

□ символических констант: #define имя текст_подстановки (все вхождения имени заменяются на текст подстановки);

□ макросов, которые выглядят как функции, но реализуются подстановкой их текста в текст программы: #define имя(параметры) текст_подстановки

□ символов у управляющих условной компиляцией. Они используются вместе с директивами #ifdef и #ifndef.

Формат: #define имя

Примеры:

```
#define VERSION 1
#define VASIA "Василий Иванович"
#define MAX(x.y) ((x)>(y)?(x):(y))
#define MUX
```

Имена рекомендуется записывать прописными буквами, чтобы зрительно отличать их от имен переменных и функций. Параметры макроса используются при макроподстановке, например, если в тексте программы используется вызов макроса y = MAX(sum1, sum2);, он будет заменен на y = ((sum1)>(sum2)?(sum1):(sum2)); Отсутствие круглых скобок может привести к неправильному порядку вычисления, поскольку препроцессор не оценивает вставляемый текст с точки зрения синтаксиса.

Например, если к макросу #define sqr(x) (x*x) обратиться как sqr(y+1), в результате подстановки получится выражение (y+1*y+1). Макросы и символические константы унаследованы из языка C, при написании программ на C++ их следует избегать. Вместо символических констант предпочтительнее использовать const или enum, а вместо макросов — встроенные функции или шаблоны.

Директивы условной компиляции

Директивы условной компиляции #if, #ifdef и #ifndef применяются для того, чтобы исключить компиляцию отдельных частей программы. Это бывает полезно при отладке или, например, при поддержке нескольких версий программы для различных платформ.

Формат директивы #if:

```
#if константное_выражение
…
[ #elif константное_выражение
…]
[ #elif константное_выражение
…]
[ #else
…]
#endif
```

Количество директив #elif — произвольное. Исключаемые блоки кода могут содержать как описания, так и исполняемые операторы.
Пример условного включения различных версий заголовочного файла:

```
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h" /* и так далее */
#else
 #define INCFILE "versN.h"
#endif
#include INCFILE
```

В константных выражениях может использоваться проверка, определена ли константа, с помощью defined(имя_константы), например:

```
#if defined( BORLANDC__ ) && __B0RLANDC__ == 0x530 // BC5.3:
typedef istream_iterator<int, char, char_traits<char>, ptrdiff_t>
istream__iter;
#elif defined(__B0RLAND_) // BC5.2:
typedef istream_iterator<int, ptrdiff_t> istream_iter;
#else                                    // VC5.0:
 typedef istream_iterator<int> istream_iter;
#endif
```

Другое назначение директивы — временно закомментировать фрагменты кода, например:

```
 #if 0
      int I, j;
       double x, y;
#endif
```

Поскольку допускается вложенность директив, такой способ весьма удобен.

Наиболее часто в программах используются директивы #ifdef и #ifndef, позволяющие управлять компиляцией в зависимости от того, определен ли с помощью директивы #define указанный в них символ (хотя бы как пустая строка, например, #define 32_BIT_SUPP0RT):

```
#ifdef символ
// Расположенный ниже код компилируется, если
символ определен
#ifndef символ
// Расположенный ниже код компилируется, если символ
не определен
```

Действие этих директив распространяется до первого #elif, #else или #endif. Директива #ifndef часто применяется для того, чтобы обеспечить включение заголовочного файла только один раз:

```
#ifndef HEADER_INCLUDED
#include "myheader.h"
#define HEADER_INCLUDED
#endif
```

Директива #undef

Директива #undef имя удаляет определение символа. Используется редко, например, для отключения какой-либо опции компилятора.

Предопределенные макросы

В C++ определено несколько макросов, предназначенных в основном для того, чтобы выдавать информацию о версии программы или месте возникновения ошибки.

cplusplus — определен, если программа компилируется как файл C++. Многие компиляторы при обработке файла с расширением .c считают, что программа написана на языке C. Использование этого макроса позволяет указать, что можно использовать возможности C++:

```
#ifdef cplusplus // Действия, специфические для C++
#endif
```

Применяется, если требуется переносить код из C в C++ и обратно.

__DATE__ — содержит строку с текущей датой в формате месяц день год, например:

```
printf(“ Дата компиляции - %s \n“, _DATE_ );
```

FILE — содержит строку с полным именем текущего файла.

LINE — текущая строка исходного текста.

TIME — текущее время, например:

```
printf(“ Ошибка в файле %s \n Время компиляции: %s\n “,
_FILE_ , _TIME_ );
```

Области действия идентификаторов

Каждый программный объект имеет область действия, которая определяется видом и местом его объявления. Существуют следующие области действия: блок, файл, функция, прототип функции, класс и поименованная область. **Блок**. Идентификаторы, описанные внутри блока, являются локальными. Область действия идентификатора начинается в точке определения и заканчивается в конце блока, видимость — в пределах блока и внутренних блоков, время жизни — до выхода из блока. После выхода из блока память освобождается. **Файл**. Идентификаторы, описанные вне любого блока, функции, класса или пространства имей, имеют глобальную видимость и постоянное время жизни и могут использоваться с момента их определения.

Функция. Единственными идентификаторами, имеющими такую область действия, являются метки операторов. В одной функции все метки должны различаться, но могут совпадать с метками других функций. Прототип функции. Идентификаторы, указанные в списке параметров прототипа (объявления) функции, имеют областью действия только прототип функции. **Класс**. Элементы структур, объединений и классов (за исключением статических элементов) являются видимыми лишь в пределах класса. Они образуются при создании переменной указанного типа и разрушаются при ее уничтожении. **Поименованная область**. C++ позволяет явным образом задать область определения имен как часть глобальной области с помощью оператора namespace.

Область видимости совпадает с областью действия за исключением ситуации, когда во вложенном блоке описана переменная с таким же именем. В этом случае внешняя переменная во вложенном блоке невидима, хотя он и входит в ее область действия. Тем не менее к этой переменной, если она глобальная, можно обратиться, используя операцию доступа к области видимости (::). Способ обратиться к скрытой локальной переменной отсутствует.

В каждой области действия различают так называемые пространства имен. Пространство имен — область, в пределах которой идентификатор должен быть уникальным.

В разных пространствах имена могут совпадать, поскольку разрешение ссылок осуществляется по контексту идентификатора в программе, например:

```
struct Node{
int Node; int i;
}Node;
```

В данном случае противоречия нет, поскольку имена типа, переменной и элемента структуры относятся к разным пространствам.

В C++ определено четыре раздельных класса идентификаторов, в пределах каждого из которых имена должны быть уникальными.

□ К одному пространству имен относятся имена переменных, функций, типов, определенных пользователем (typedef) и констант перечислений в пределах одной области видимости. Все они, кроме имен функций, могут быть переопределены во вложенных блоках.

□ Другой класс имен образуют имена типов перечислений, структур, классов и объединений. Каждое имя должно отличаться от имен других типов в той же области видимости.

□ Отдельный класс составляют элементы каждой структуры, класса и объединения. Имя элемента должно быть уникально внутри структуры, но может совпадать с именами элементов других структур.

□ Метки образуют отдельное пространство имен.

Внешние объявления

Любая функция автоматически видна во всех модулях программы. Если требуется ограничить область действия функции файлом, в котором она описана, используется модификатор static. Для того чтобы сделать доступной в нескольких модулях переменную или константу, необходимо:

□ определить ее ровно в одном модуле как глобальную;

□ в других модулях объявить ее как внешнюю с помощью модификатора extern.

Другой способ — поместить это объявление в заголовочный файл и включить его в нужные модули. Все описания одной и той же переменной должны быть согласованы.

Пример описания двух глобальных переменных в файлах one.cpp и two.cpp с помощью заголовочного файла my_header.h:

```
// my_header.h - внешние объявления
extern int a;
extern double b;
…
//--------------
// one.cpp
#include "my_header.h"
int a;
//
// two.cpp
#include "my_header.h"
double b; …
```

Обе переменные доступны в файлах one. cpp и two. cpp.

Если переменная описана как static, область ее действия ограничивается файлом, в котором она описана. При *описании типа* следует придерживаться *правила одного определения*, то есть тип, используемый в программе, должен быть определен ровно один раз. Как правило, это делается в заголовочном файле, который затем подключается к модулям, использующим этот тип. Нарушение этого правила приводит к ошибкам, которые трудно обнаружить, поскольку компиляторы, как правило, не обладают возможностью сличать определения одного и того же типа в различных файлах.

Поименованные области

служат для логического группирования объявлений и ограничения доступа к ним. Чем больше программа, тем более актуально использование поименованных областей. Простейшим примером применения является отделение кода, написанного одним человеком, от кода, написанного другим. При использовании единственной глобальной области видимости формировать программу из отдельных частей очень сложно из-за возможного совпадения и конфликта имен. Использование поименованных областей препятствует доступу к ненужным средствам. Объявление поименованной области (ее также называют пространством имен) имеет формат:

```
namespace [ имя_области ]{ /* Объявления */ }
```

Поименованная область может объявляться неоднократно, причем последующие объявления рассматриваются как расширения предыдущих. Таким образом, поименованная область может объявляться и изменяться за рамками одного файла. Если имя области не задано, компилятор определяет его самостоятельно с помощью уникального идентификатора, различного для каждого модуля. Объявление объекта в неименованной области равнозначно его описанию как глобального с модификатором static. Помещать объявления в такую область полезно для того, чтобы сохранить локальность кода. Нельзя получить доступ из одного файла к элементу неименованной области другого файла.

Пример.

```
namespace demo{
int i = 1; int k = 0;
void func1 (int);
void func2(int) { /* ... */ }
}
namespace demo{ // Расширение
// int i = 2; Неверно - двойное определение
void func1 (double); // Перегрузка
void func2(int); // Верно (повторное объявление)
 }
```

В объявлении поименованной области могут присутствовать как объявления, так и определения.

Логично помещать в нее только объявления, а определять их позднее с помощью имени области и оператора доступа к области видимости ::,
например: void demo::func1(int) { /* ... */ }
Это применяется для разделения интерфейса и реализации. Таким способом нельзя объявить новый элемент пространства имен.
Объекты, объявленные внутри области, являются видимыми с момента объявления. К ним можно явно обращаться с помощью имени области и оператора доступа к области видимости ::,
например:
demo::i = 100; demo::func2(10);
Если имя часто используется вне своего пространства, можно объявить его доступным с помощью оператора using: using demo::i; После этого можно использовать имя без явного указания области.

Если требуется сделать доступными все имена из какой-либо области, используется оператор using namespace:

```
using namespace demo;
```

Операторы using и using namespace можно использовать и внутри объявления поименованной области, чтобы сделать в ней доступными объявления из другой области:

```
namespace Department_of_Applied_Mathematics{ using
demo::I;
// ... }
```

Имена, объявленные в поименованной области явно или с помощью оператора using, имеют приоритет по отношению к именам, объявленным с помощью оператора using namespace (это имеет значение при включении нескольких поименованных областей, содержащих совпадающие имена).

Короткие имена пространств имен могут войти в конфликт друг с другом, а длинные непрактичны при написании реального кода, поэтому допускается вводить синонимы имен:

```
namespace DAM = Department_of_Applied_Mathematics;
```

Пространства имен стандартной библиотеки.

Объекты стандартной библиотеки определены в пространстве имен std.
Например, объявления стандартных средств ввода/вывода С в заголовочном файле <stdio.h> помещены в пространство имен следующим образом:

```
// stdio.h
namespace std{
int feof(FILE *f);
}
using namespace std;
```

Это обеспечивает совместимость сверху вниз. Для тех, кто не желает присутствия неявно доступных имей, определен новый заголовочный файл <cstdio>:

```
// cstdio
namespace std{
int feof(FILE *f);
}
```

Если в программу включен файл <cstdio>, нужно указывать имя пространства имен явным образом:

```
std::feof(f);
```

Механизм пространств имен вместе с директивой #include обеспечивают необходимую при написании больших программ гибкость путем сочетания логического группирования связанных величии и ограничения доступа.

Как правило, в любом функционально законченном фрагменте программы можно выделить интерфейсную часть (например, заголовки функций, описания типов), необходимую для использования этого фрагмента, и часть реализации, то есть вспомогательные переменные, функции и другие средства, доступ к которым извне не требуется. Пространства имен позволяют скрыть детали реализации и, следовательно, упростить структуру программы и уменьшить количество потенциальных ошибок. Продуманное разбиение программы на модули, четкая спецификация интерфейсов и ограничение доступа позволяют организовать эффективную работу над проектом группы программистов.

The End