Лекция 5
Программирование с использованием функций

Использование модификаторов при декларации производных типов данных

Ключевые слова *int*, *float*, *char* и т.д. называют конечными атрибутами декларации объектов программы. При декларации так называемых производных объектов используют еще дополнительные – промежуточные атрибуты или, как их иногда называют, «***модификаторы***».

К символам модификации текущего типа относятся:

– символ ***** перед идентификатором, обозначающий декларацию указателя на объект исходного типа (левый промежуточный атрибут);

– символы [ ] после идентификатора объекта – декларация массива объектов;

– символы ( ) после идентификатора объекта – декларация функции (правые промежуточные атрибуты).

Допускается использование более одного модификатора типа с учетом следующих правил:

1) чем ближе модификатор к *ID* объекта, тем выше его приоритет;

2) при одинаковом расстоянии от идентификатора объекта модификаторы [ ] и ( ) обладают приоритетом перед атрибутом звездочка *;

3) дополнительные круглые скобки позволяют изменить приоритет объединяемых ими элементов описания;

4) квадратные и круглые скобки, имеющие одинаковый приоритет, рассматриваются слева направо.

Конечный атрибут декларации принимается во внимание в последнюю очередь, т.е. тогда, когда все промежуточные атрибуты уже проинтерпретированы.

Примеры декларации объектов с конечным атрибутом ***int***:

int a; – переменная типа *int*;

int a[5]; – массив из пяти элементов типа *int*;

*int *a*; – указатель на объект типа *int*;

*int **a*; – указатель на указатель на объект типа *int*;

*int *a*[5]; – массив из пяти указателей на элементы типа *int*;

int (**a*)[10]; – указатель на массив из десяти элементов типа *int*;

*int *a*[3][4]; – 3-элементный массив указателей на одномерные целочисленные массивы по четыре элемента каждый;

int a[5][2]; – двухмерный массив элементов типа *int*;

int a(*void*); – функция без параметров, возвращающая значение типа *int*;

*int *a*(*void*); – функция без параметров, возвращающая указатель на элемент типа *int*;

int (**a*)(*void*); – указатель на функцию без параметров, возвращающую значение типа *int*;

*int *a*(*void*)[6]; – функция без параметров, возвращающая указатель на массив элементов типа *int*;

int **a*[4](*void*); – массив указателей на функцию без параметров, возвращающую значение типа *int*.

Существуют и недопустимые последовательности промежуточных атрибутов, например, массив не может состоять из функций, а функция не может возвращать массив или другую функцию.

Функции

С увеличением объема программы ее код становится все более сложным. Одним из способов борьбы со сложностью любой задачи является ее разбиение на части. В языке Си, как и в любом языке программирования высокого уровня, задача может быть разбита на более простые подзадачи при помощи подпрограмм-функций. После этого программу можно рассматривать в более укрупненном виде – на уровне взаимодействия созданных подпрограмм. Использование подпрограмм в коде программы и ведет к упрощению ее структуры.

Часто используемые функции можно помещать в отдельные библиотеки.

Далее функции и связанных с ними данные можно сгруппировать 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.

Для того чтобы использовать модуль, достаточно знать только его интерфейс, а не все детали его реализации.

Разделение программы на максимально обособленные части (подпрограммы) является довольно сложной задачей, которая должна решаться на этапе проектирования программы.

В отличие от других языков программирования высокого уровня в языке C/C++ нет разделения на подпрограммы-процедуры и подпрограммы-функции, здесь вся программа строится только из функций. Функция представляет собой отдельный программный модуль, к которому можно обратиться, чтобы передать через параметры исходные данные и получить один или несколько результатов его работы.

Функция – это именованная последовательность инструкций, выполняющая какое-либо законченное действие.

Таким образом, любая программа на языке C/C+ состоит из функций. Минимальная программа на C/C++ содержит, как уже известно, единственную функцию main (основная, главная), с которой и начинается выполнение программы.

Декларация функции. Прототип функции. Параметры функции. Возвращение значения функций

Как и любой объект программы на языке C/C++, пользовательские функции необходимо декларировать. Объявление функции пользователя, т.е. ее декларация, выполняется в двух формах – в форме описания (объявления) и в форме определения, т.е. любая пользовательская функция должна быть объявлена и определена.

Описанием функции является декларация ее прототипа, который сообщает компилятору о том, что далее будет приведено ее полное определение (текст), т.е. реализация.

Объявление функции (прототип, заголовок) задает ее свойства – идентификатор, тип возвращаемого значения (если такое имеется), количество и типы параметров.

В стандарте языка используется следующий формат декларации (объявления) функций:

тип_результата ID_функции (список);

В списке перечисляются типы параметров данной функции, причем идентификаторы переменных в круглых скобках прототипа указывать необязательно, т.к. компилятор языка их не обрабатывает.

Описание прототипа дает возможность компилятору проверить соответствие типов и количества параметров при фактическом вызове этой функции.

Пример объявления функции fun, которая имеет три параметра типа int, один параметр типа double и возвращает результат типа double:

```
double fun(int, int, int, double);
```

Каждая функция, вызываемая в программе, должна быть определена (только один раз). Определение функции – это ее полный текст, включающий заголовок и код.

Полное определение (реализация) функции имеет следующий вид:

```
тип_результата ID_функции(список параметров)
{
код функции
return выражение;
}
```

Рассмотрим составные части определения пользовательской функции.

Тип результата определяет тип выражения, значение которого возвращается в точку ее вызова при помощи оператора return выражение; (возврат).

Выражение преобразуется к типу результата, указанному в заголовке функции и передается в точку вызова. Тип возвращаемого функцией значения может быть любым базовым типом, а также указателем на массив или функцию. Если функция не должна возвращать значение, указывается тип void. В данном случае оператор return можно не ставить.

Из функции, которая не описана как void, необходимо возвращать значение, используя оператор return. Если тип функции не указан, то по умолчанию устанавливается тип int.

Список параметров состоит из перечня типов и идентификаторов параметров, разделенных запятыми. Список параметров определяет объекты, которые требуется передать в функцию при ее вызове.

В определении и в объявлении одной и той же функции ***типы и порядок следования параметров должны совпадать***. Тип возвращаемого значения и типы параметров совместно определяют тип функции.

Функция может не иметь параметров, но круглые скобки необходимы в любом случае. Если у функции отсутствует список параметров, то при декларации такой функции желательно в круглых скобках указать void. Например,

```
void main(void){ ... }.
```

В функции может быть несколько операторов return, но может и не быть ни одного (тип void – это определяется потребностями алгоритма). В последнем случае возврат в вызывающую программу происходит после выполнения последнего оператора кода функции.

Пример функции, определяющей наименьшее значение из двух целочисленных переменных:

```
int min (int x, int y)
{
return (x<y) ? x : y;
}
```

Функции, возвращающие значение, желательно использовать в правой части выражений языка C/C++, иначе возвращаемый результат будет утерян.

В языке C/C++ каждая функция – это отдельный блок программы, вход в который возможен только через вызов данной функции.

Вызов функции. Аргументы функции

Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечислить список передаваемых ей аргументов. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция.

Простейший вызов функции имеет следующий формат:

```
ID_функции (список аргументов);
```

где в качестве аргументов можно использовать константы, переменные, выражения (их значения перед вызовом функции будут определены компилятором).

Аргументы в списке вызова должны совпадать со списком параметров вызываемой функции по количеству и порядку следования, а типы аргументов при передаче в функцию будут преобразованы, если это возможно, к типу соответствующих им параметров.

Связь между функциями осуществляется через аргументы и возвращаемые функциями значения. Ее можно осуществить также через внешние, глобальные переменные.

Глобальные переменные доступны всем функциям, где они не описаны как локальные переменные. Использовать их для передачи данных между функциями довольно просто, но тем не менее этого делать не рекомендуется. Необходимо стремиться к тому, чтобы функции в программе были максимально независимыми, и чтобы их интерфейс полностью определялся прототипами этих функций.

Функции могут располагаться в исходном файле в любом порядке, при этом исходная программа может размещаться в нескольких файлах.

Все величины, описанные внутри функции, являются локальными. Областью их действия является функция. При вызове функции, как и при входе в любой блок, в стеке выделяется память под 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, например:

```
#include <stdio.h>
void f1(int);
void main(void)
{
f1(5);
}
void f1(int i)
{
int m=0;
puts(" n m p ");
while (i--)
{
   static int n = 0;
   int p = 0;
   printf(" %d %d %d \n", n++ , m++ , p++);
  }
}
```

Статическая переменная *n* будет создана в сегменте данных и проинициализируется нулем только один раз при первом выполнении оператора, содержащего ее определение, т.е. при первом вызове функции f1.

Автоматическая переменная *m* инициализируется при каждом входе в функцию.

Автоматическая переменная *p* инициализируется при каждом входе в блок цикла.

В результате выполнения программы получим

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

Передача аргументов в функцию. Использование указателей как аргументов функции

В языке C/C++ аргументы при стандартном вызове функции передаются по значению. Это означает, что в стеке, как и в случае локальных данных, выделяется место для формальных параметров функции. В выделенное место при вызове функции заносятся значения фактических аргументов, при этом проверяется соответствие типов и при необходимости выполняются их преобразования. При несоответствии типов выдается диагностическое сообщение. Затем функция использует и может изменять эти значения в стеке.

При выходе из функции измененные значения теряются, т.к. время жизни и зона видимости локальных параметров определяется кодом функции. Вызванная функция не может изменить значения переменных, указанных как фактические аргументы при обращении к данной функции.

В случае необходимости функцию можно использовать для изменения передаваемых ей аргументов. В этом случае в качестве аргумента необходимо в вызываемую функцию передавать ***не значение переменной, а ее адрес***.

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. Для обращения к значению аргумента-оригинала используется операция «*».

Пример функции, в которой меняются местами значения x и y:

```
void zam(int *x, int *y)
{
int t = *x;
*x = *y;
*y = t;
}
```

Участок программы с обращением к данной функции:

```
void zam (int*, int*);
void main (void)
{
```

```
int a=2, b=3;
printf(" a = %d , b = %d\n", a, b);
zam (&a, &b);
printf(" a = %d , b = %d\n", a, b);
}
```

При таком способе передачи данных в функцию их значения будут изменены, т.е. на экран монитора будет выведено

a = 2 , b=3
a = 3 , b=2

Если требуется запретить изменение значений какого-либо параметра внутри функции, то в его декларации используют атрибут const, например:

```
void f1(int, const double);
```

Рекомендуется указывать const перед всеми параметрами, изменение которых в функции не предусмотрено. Это облегчает, например, отладку программы, т.к. по заголовку функции видно, какие данные в функции изменяются, а какие нет.

Операция typedef

Любому типу данных, как стандартному, так и определенному пользователем, можно задать новое имя с помощью операции typedef:

```
typedef  тип  новое_имя ;
```

Введенный таким образом новый тип используется аналогично стандартным типам.

Пример.

`typedef unsigned int UINT;` здесь UINT – новое имя;

`typedef char M_s [101];` здесь M_s – тип пользователя, определяющий строки, длиной не более 100 символов.

Декларации объектов введенных типов будут иметь вид

`UINT  i, j;` → две переменные типа unsigned int ;

`M_s  str[10];` → массив из 10 элементов, в каждом из которых можно хранить по 100 символов.

Рассмотренная операция упрощает использование указателей на функции.

Указатели на функции

В языке C/C++ идентификатор функции является константным указателем на начало функции в оперативной памяти и не может быть значением переменной.

Но имеется возможность декларировать указатели на функции, с которыми можно обращаться как с переменными (например, можно создать массив, элементами которого будут указатели на функции).

Рассмотрим методику работы с указателями на функции.

1. Как и любой объект языка C/C++, указатель на функции необходимо декларировать. Формат объявления указателя на функции следующий:

тип (*переменная-указатель)(список параметров);

т.е. декларируется указатель, который можно устанавливать на функции, возвращающие результат указанного типа и которые имеют указанный список параметров. Наличие первых круглых скобок обязательно, так как без них – это декларация функции, которая возвращает указатель на результат.

Например, объявление вида

double (*p_f)(char, double);

говорит о том, что декларируется указатель p_f, который можно устанавливать на функции, возвращающие результат типа double и имеющие два параметра: первый – символьного типа, а второй – вещественного типа.

2. Идентификатор функции является константным указателем, поэтому для того чтобы установить переменную-указатель на конкретную функцию, достаточно ей присвоить ее идентификатор:

переменная-указатель = ID_функции;

Например, имеется функция с прототипом:

double f1(char, double);

тогда операция

p_f = f1;

установит указатель p_f на данную функцию.

3. Вызов функции после установки на нее указателя выглядит так:

(*переменная-указатель)(список аргументов);

Или

переменная-указатель (список аргументов);

После таких действий кроме стандартного обращения к функции:

D_функции(список аргументов);

появляется еще два способа вызова функции:

(*переменная-указатель)(список аргументов);

или

переменная-указатель (список аргументов);

Последняя запись справедлива, так как p_f также является адресом начала функции в оперативной памяти.

Для нашего примера к функции f1 можно обратиться следующими способами:

f1('z', 1.5); – обращение к функции по имени (ID);
(* p_f)('z', 1.5); – обращение к функции по указателю;
p_f('z', 1.5); – обращение к функции по ID указателя.

Основное назначение указателей на функции – это обеспечение возможности передачи идентификаторов функций в качестве параметров в функцию, которая реализует некоторый вычислительный процесс, используя формальное имя вызываемой функции.

Пример. Написать функцию вычисления суммы sum, обозначив слагаемое формальной функцией fun(x). При вызове функции суммирования передавать через параметр реальное имя функции, в которой задан явный вид слагаемого. Например, пусть требуется вычислить две суммы:

$$S_1 = \sum_{i=1}^{2n} \frac{x}{5} \quad \text{и} \quad S_2 = \sum_{i=1}^{n} \frac{x}{2}.$$

Поместим слагаемые этих сумм в пользовательские функции f1 и f2 соответственно. При этом воспользуемся операцией typedef, введя пользовательский тип данных: указатель на функции p_f, который можно устанавливать на функции, возвращающие результат double и имеющие один параметр типа double.

Тогда в списке параметров функции суммирования достаточно будет указать фактические идентификаторы функций созданного типа p_f.

Текст программы для решения данной задачи может быть следующим:

```
. . .
typedef double (*p_f)(double);
double sum(p_f, int, double);    // Декларации прототипов функций
double f1(double);
double f2(double);

void main(void)
{
double x, s1, s2;
int n;
puts (" Введите кол-во слагаемых n и значение x: ");
scanf (" %d %lf ", &n, &x);
s1 = sum (f1, 2*n, x);
```

```
s2 = sum (f2, n, x);
printf("\n\t N = %d, X = %lf ", n, x);
printf("\n\t Сумма 1 = %lf\n\t Сумма 2 = %lf ", s1, s2);
}
```

/* Первый параметр функции суммирования – формальное имя функции, введенное с помощью typedef типа */

```
double sum(p_f fun, int n, double x) {
double s=0;
for(int i=1; i<=n; i++)
s+=fun(x);
return s;
}

//————————— Первое слагаемое ———————————
double f1(double r) {
return r/5.;
}

//————————— Второе слагаемое ———————————
double f2(double r) {
return r/2.;
}
```