Областью действия (областью видимости) идентификатора называется область программы, в которой на данную переменную (как, впрочем, и на идентификатор, константу, функцию и т.п.) можно сослаться. На некоторые переменные можно сослаться в любом месте программы, тогда как на другие — только в определенных ее частях.

Класс памяти определяется, в частности, местом объявления переменной. *Локальные переменные* объявляются внутри некоторого блока или функции. Эти переменные видны только в пределах того блока, в котором они объявлены. *Блоком* называется фрагмент кода, ограниченный фигурными скобками "{ }". *Глобальные переменные* объявляются вне какого-либо блока или функции.

Спецификации класса памяти могут быть разбиты на два класса: *автоматический класс памяти* с *локальным временем жизни* и *статический класс памяти* с *глобальным временем жизни.* Ключевые слова **auto и register** используются для объявления переменных с локальным временем жизни. Эти спецификации применимы только к локальным переменным. Локальные переменные создаются при входе в блок, в котором они объявлены, существуют лишь во время активности блока и исчезают при выходе из блока.

Локальные переменные являются переменными с локальным временем жизни по умолчанию, так что ключевое слово **auto**

Если интенсивно используемые переменные, такие как счетчики или суммы могут сохраняться в аппаратных регистрах, накладные расходы на повторную загрузку переменных из памяти в регистр и обратную загрузку результата в память могут быть исключены. Это сокращает время вычислений.

Компилятор может проигнорировать объявления **register.** Например, может оказаться недостаточным количество регистров, доступных компилятору для использования. К тому же оптимизирующий компилятор способен распознавать часто используемые переменные и решать, помещать их в регистры или нет. Так что явное объявление спецификации **register** используется редко.

Ключевые слова **extern и static** используются, чтобы объявить идентификаторы переменных как идентификаторы статического класса памяти с глобальным временем жизни. Такие переменные существуют с момента начала выполнения программы. Для таких переменных память выделяется и инициализируется сразу после начала выполнения программы.

Существует два типа переменных статического класса памяти: глобальные переменные и локальные переменные, объявленные спецификацией класса памяти **static.** Глобальные переменные по умолчанию относятся к классу памяти **extern** Глобальные переменные создаются путем размещения их объявлений вне описания какой-либо функции и сохраняют свои значения в течение всего времени выполнения программы. На глобальные переменные может ссылаться любая функция, которая расположена после их объявления или описания в файле

Все числовые переменные статического класса памяти принимают нулевые начальные значения, если программист явно не указал другие начальные значения. Статические переменные — указатели, тоже имеют нулевые начальные значения.

Спецификации класса памяти **extern** используются в программах с

Функции

План лекции: Функции. Описание, определение функции. Примеры функций. Передача параметров. Использование общих областей памяти.

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). **Любая функция должна быть объявлена, определена и начинает выполняться в момент вызова.** Объявление функции должно находиться раньше ее вызова для того, чтобы компилятор проверил правильность вызова.

Объявление функции задает ее имя, тип возвращаемого значения и список передаваемых параметров.

Определение функции содержит, кроме объявления, тело функции, состоящее из описаний и операторов в фигурных скобках.

[**класс**] **тип**_функции **имя**_функции ([**список параметров**])
{тело функции}

Рассмотрим составные части определения.

Необязательный **класс** памяти: **extern** – задает глобальную видимость во всех модулях программы (по умолчанию), **static** – задает 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).

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 определении, в объявлении и при вызове одной и той же функции типы и порядок следования должны совпадать.

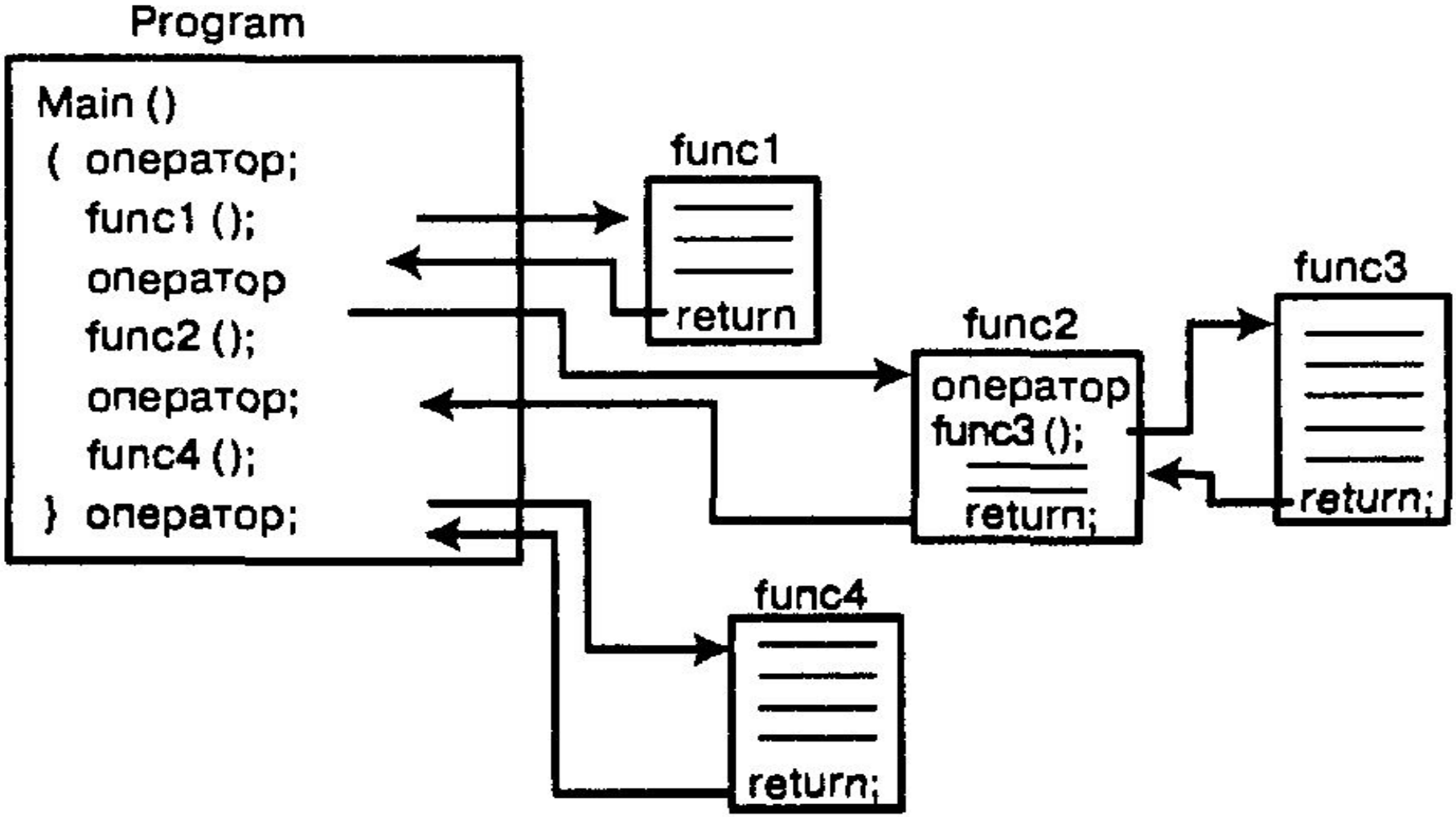
Для *вызова* функции нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов.

имя_функции ([**список аргуметров**])

Вызов функции может быть в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция. Если тип возвращаемого функцией значения не void , она функция может входить в состав выражений или располагаться в правой части оператора присваивания. Пример функции, возвращающей сумму двух целых величин:

```
int main(int argc, char* argv[])
        {int a=2, b=3, c, d;
        c = sum(a, b); // вызов
функции
        scanf("%d ",&d);
printf ("%d ", sum(c, d));
 // вызов функции
       return 0;
    }
int sum (int a, int b)
// Объявление функции
    {
        return (a+b);
     }
```

```
int sum (int a, int b) //
Объявление функции
    {
        return (a+b);
     }
int main(int argc, char* argv[])
        {int a=2, b=3, c, d;
        c = sum(a, b); // вызов
функции
        scanf("%d ",&d);
 printf ("%d ", sum(c, d));
// вызов функции
       return 0;
    }
```



Когда программа вызывает функцию, управление переходит к телу функции, а затем выполнение программы возобновляется со строки, следующей после вызова

Все величины, описанные внутри функции, а также ее параметры являются локальными. Областью их действия является функция.

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

Определение функции с аргументом

Определение нашей функции начинается с двух строк:

```
space (number);
int number; {…}
```

Первая строка информирует компилятор о том, что у функции **space () имеется** аргумент и что его **имя number .** Вторая строка — описание, **указывающее** компилятору, что аргумент **number имеет** тип **int .** Аргумент описывается перед фигурной скобкой, которая отмечает начало тела функции. Можно объединить эти две строки в одну:

```
space (int number) {…}
```

Переменная **number** называется «формальным» аргументом. Фактически это новая переменная, и в памяти компьютера для нее должна быть выделена отдельная ячейка.

Вызов функции с аргументом

Переменной **number** присваивается значение фактического аргумента при вызове функции. Например, случай использования функции space ( ):

space (25):

Фактический аргумент здесь 25, и эта *величина* присваивается формальному аргументу — переменной **number ,** т. е, вызов функции оказывает следующее действие:

number = 25;

Формальный аргумент — переменная в вызываемой программе, а фактический аргумент — конкретное значение, присвоенное этой переменной вызывающей программой. Фактический аргумент может быть константой, переменной или даже более сложным выражением. Независимо от типа фактического аргумента он вначале вычисляется, а затем его

Возвращаемое значение

Механизм возврата из функции в вызвавшую ее функцию реализуется оператором

return [выражение];

Функция может содержать несколько операторов return (если это необходимо алгоритму функции). Если функция описана как void, выражение не указывается. Оператор return можно опускать для функции типа void, если возврат из нее происходит перед закрывающейся фигурной скобкой, и для

```
#include <iostream.h>

int string_length(char string[])
 {
   int i;

   for (i = 0; string[i] != '\0'; i++);  // Ничего не делать,
                                         // но перейти к
                                         // следующему символу

   return(i); // Длина строки
 }

void main(void)
 {
   char title[] = "Учимся программировать на языке C++";
   char lesson[] = "Символьные строки";

   cout << "Строка " << title << " содержит " <<
        string_length(title) << " символов" << endl;
 cout << "Строка " << lesson << " содержит " <<
      string_length(lesson) << " символов" << endl;
}
```

Локальные переменные

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. с. 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. Переменные, известные нескольким функциям, называются «глобальными».Если мы используем одно и то же имя для переменных в двух различных функциях, компилятор «считает» их разными переменными. Мы можем показать это, используя операцию &.

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

Пример. Для целых m,n вычислить выражение

$$y = \frac{m!n!}{(m+n)!}$$

```
#include <conio.h>
#include <stdio.h>
float fak(int x)
{  float p=1;
   for (int j=1;j<=x;j++)
      p*=j;
  return (p);
}
main ()
{
 int m,n;
 float c;
 printf("введите m,n : ");
 scanf("%d%d",&m,&n);
c=fak(m)*fak(n)/fak(m+n);
 printf(" %e\n",c);
getch();
}
```

```
#include <stdio.h>
int max (int m, int n, int k);     // прототип функции
main ()
{ int a,b,c;   // локальные переменные для main, но
глобальные для max
printf (‘a, b,c=’);
scanf (“%i,%i”,&a,&b,&c);
printf (“max (a,b,c)=%i”, max (a,b,c));
return (0);
}
int max (int m, int n, int k)
{
int d;      //  локальные переменные для max
d=(m>n)? m : n;
d=(d<k)? k : d;
return d;
}
```

Нахождение адресов: операция &

В результате выполнения операции & определяется адрес ячейки памяти, которая соответствует переменной. Если **p** — имя переменной, то **& p** — ее адрес. Можно представить себе адрес как ячейку памяти, но можно рассматривать его и как метку, которая используется компьютером для идентификации переменной. Мы имеем оператор

```
p = 24;
```

Пусть также адрес ячейки, где размещается переменная **p** — 12126. Тогда в результате выполнения оператора

```
printf ("%d %d \n", p, &p);
```

получим 24 12126

Воспользуемся указанной выше операцией для проверки того, в каких ячейках хранятся значения переменных, принадлежащих разным функциям, но

```
// Контроль адресов
int  fun (int b)
{
   int a = 10;
   printf(" В fun( ), a = %d и &a = %u\n", a, &a);
   printf(" В fun( ), b = %d и &b = %u\n", b, &b);
 return 0;
}
int main(int argc, char* argv[])
{
   int a = 2, b = 5;
   printf(" В main( ), a = %d и &a = %u\n", a, &a);
   printf(" В main( ), b = %d и &b = %u\n", b, &b);
   fun(a);
}
```

Мы воспользовались форматом % u (целое без знака) для вывода на печать адресов на тот случай, если их величины превысят максимально возможное значение числа типа **int.** В нашей вычислительной системе результат работы этой программы выглядит так:

В main( ), a = 2 и &a = 1245060
В main( ), b = 5 и &b = 1245064
В fun ( ), a = 10 и &a = 1245044
В fun ( ), b = 2 и &b = 1245056

О чем это говорит? Во-первых, две переменные **a** имеют различные адреса. То же самое верно и относительно переменных **b .** Компьютер рассматривает их как четыре разные переменные.

Во-вторых, при вызове fun **(a)** величина (2) фактического аргумента (a из main ( )) передается формальному аргументу (b из fun ( )). Было передано только значение переменной. Адреса двух переменных (a в main ( ) и b в fun ( )) остаются различными. *В языке C++ аргументы передаются по*

Пример, при вызове функции **prim** используется операция взятия адреса, а в теле функции указатели не применяются.

```
 int prim (int &pa, int &pb); // объявление функции
int main ( )

int ia, ib, ic;
cin>>ia;
cin >>ib;
ic=prim (ia, ib); // вызов функции
 cout <<“summa =”<<ic;
return 0;
}
// описание функции
int prim (int &pa, int &pb)
{int q;
q=pa +pb;
 return (q);
}
```

Параметры функции

Механизм параметров – основной способ обмена информацией между вызываемой функцией и вызывающей функцией. Параметры, перечисленные в заголовке описания функции, называются *формальными* параметрами, или просто параметрами, а записанные в операторе вызова – фактическими параметрами, или аргументами.

Существуют 2 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.

-формальные параметры являются собственными переменными функции;

- при вызове функции присваиваются значения фактических параметров формальным (копирование первых во вторые);

- при изменении формальных параметров значения соответствующих им фактических параметров не меняются.

При передаче по адресу в стек заносятся копии адресов аргументов, а функция осуществляет доступ к ячейкам памяти по этим адресам и может изменять исходные значения аргументов.

Пример передачи параметров в функцию по значению.

```
// Обмен 1
change (int u, int v)
 {
   int temp;
   temp = u;
   u = v;
   v = temp;
}
int main(int argc, char* argv[]) {
   int x = 5, y = 10;
   printf(" Вначале x = %d и y  = %d\n", x, y);
   change (x, y);
   printf(" Теперь x = %d и y  = %d\n", x, y);
}
```

Результаты будут выглядеть следующим образом:

Вначале x = 5 и y = 10
Теперь x = 5 и y = 10

Функции change () и main () используют различные переменные, поэтому обмен значениями между переменными u и v не оказывает никакого влияния на x и y. Ниже приводится программа, в которой указатели служат средством, обеспечивающим правильную работ у функции, которая осуществляет обмен значениями переменных.

```
// обмен2
change (int *u, int *v) { // u и v являются указателями
    int temp;
    temp = *u; // temp присваивается значение, на которое указывает u
    *u = *v;
    *v = temp;
}
int main(int argc, char* argv[]) {
    int x=5, y = 10;
    printf("Вначале x = %d и y = %d\n", x, y);
    change(&x, &y); // передача адресов функции
    printf("Теперь x = %d и y = %d\n", x, y);
}
```

Результат:

Вначале x = 5 и y = 10.
Теперь x = 10 и y = 5.

Во-первых, теперь вызов функции выглядит следующим образом:

change (&x, &y);

Вместо передачи ***значении*** **x** и **y** мы передаем их ***адреса.*** Это означает, что формальные аргументы **u** и **v** при обращении будут заменены адресами и, следовательно, они должны быть описаны как указатели. Поскольку x и y — целого типа, u и v являются указателями на переменные целого типа, и мы вводим описание int * u , int * v для описания формальных параметров функции change . Далее в теле функции оператор описания int temp используется с целью резервирования памяти. Мы хотим поместить значение переменной x в переменную temp , поэтому пишем temp = * u ; Мы *не должны* писать, например, так:

temp = u; // неправильно

поскольку при этом происходит запоминание *адреса* переменной x, а не ее *значения;* мы же пытаемся осуществить обмен значениями, а не адресами.

Параметры функции по умолчанию. Такие параметры можно не указывать при вызове функции, т.к. они примут значение по умолчанию, указанное после знака присваивания после данного параметра и списке всех параметров функции.

В этом случае спецификация аргумента имеет вид
тип имя=ЗначениеПоУмолчанию
Если аргумент имеет значение по умолчанию, то все аргументы, специфицированные после него, также должны иметь значения по умолчанию.

Пример Вычислить n в степени k, где чаще всего k=2. Рекурсивная функция со значением k=2 по умолчанию:

```
int power(int n, int k = 2)  // по умолчанию k=2
{
  if (k == 2) return(n*n);
  else return(power(n, k - 1)*n);
}
```

Вызывать эту функции можно двумя способами:

```
t = power(i);  // t = i*i;
q = power(i, 5); // q = i*i*i*i*i;
```

Значение по умолчанию может быть задано либо при объявлении функции, либо при определении функции, но только один раз.

- Область **статических переменных** локальна и их отличие от автоматических переменных: если функция, описывающая эту переменную, завершает свою работу, статические переменные не будут удалены. Компилятор запоминает предыдущие значения статических переменных до следующего вызова функции.
- Например:

```
#include <stdio.h>
 int fun()
   { int av=1;
     static int stv=1;
     printf("av=%d stv=%d\n", av++, stv++);
   }
 main()
 { for(int k=1; k<=3; k++)
     { printf("%d- шаг: ",k); fun(); }
   getch();
 }
```

- Результат:

```
1- шаг: av=1 stv=1
2- шаг: av=1 stv=2
3- шаг: av=1 stv=3
```

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, поэтому следует передавать его размерность через отдельный параметр.

// Заголовок: тип результата имя(параметр 1, параметр 2)

int sum (int A[], int n) // Массив отображается, размерность копируется

// Тело функции (блок)

{ int s,i; // Локальные (автоматические) переменные блока

for (i=s=0; i<n; i++) // Последовательность операторов блока

s +=A[i];

return s ; } // Значение результата в return

```
int c[10] = {1,6,4,7,3,56,43,7,55,33};
int g[10] = {1,6,4,7,3,56,43,7,55,33};
void main()
{ int nn,hh;
nn = sum(c,10);
 hh= sum(g,5);
cout<<“nn=“<<nn<<“  hh= “<<hh;
}
```

```
# include <stdio.h>
void summ_pr(int *prmas);
int main()
{
int inmas[5]= {10,4,16,0,45};
summ_pr (inmas);
return(0);
}
void summ_pr(int *prmas)
{
int t,s;
for (t=s=0; t<5; t++)
 s+=prmas[t];
printf("сумма элементов массива=%i \n",s);
}
```

Прототип функции

В программе должна присутствовать функция, которая автоматически вызывается при загрузке программы в память и при ее выполнении. Более никаких особенностей, кроме указанной, эта функция не имеет.

Объявление функции - информация транслятору о наличии функции с заданным заголовком (прототипом) либо в другом модуле, либо далее по тексту текущего модуля - «вниз по течению».

Объявление функции состоит из прототипа, предваренного словом extern, либо просто из прототипа функции.

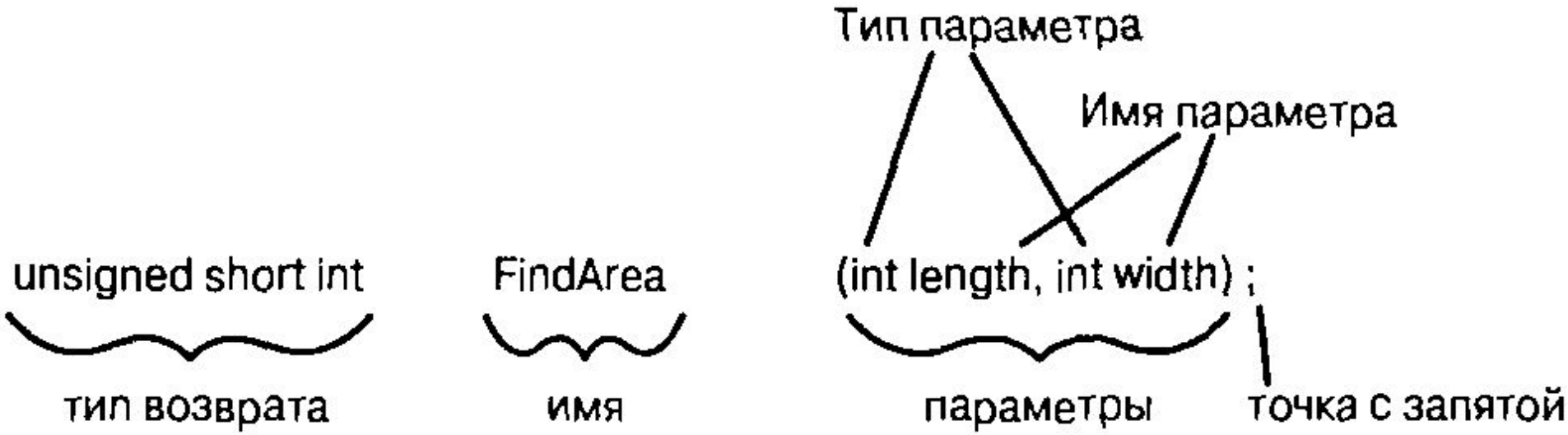
Прототип функции - заголовок функции со списком формальных параметров, заданных в виде абстрактных типов данных.

int clrscr(); // Без контроля соответствия (анахронизм)

int cirscr{void); // Без параметров

int strcmp(char*, char*);

extern int strcmp(); // Без контроля соответствия (анахронизм)



Составные части прототипа функции

В прототипе и в определении функции тип возвращаемого значения и сигнатура должны соответствовать. Если такого соответствия нет, компилятор покажет сообщение об ошибке. Однако прототип функции не обязан содержать имена параметров, он может ограничиться только указанием их типов. Например, прототип, приведенный ниже, абсолютно правомочен:

```
long Area(int, int);
```

Этот прототип объявляет функцию с именем `Area()`, которая возвращает значение типа `long` и принимает два целочисленных параметра. И хотя такая запись прототипа вполне допустима, это не самый лучший вариант. Добавление имен параметров делает ваш прототип более ясным. Та же самая функция, но уже с именованными параметрами, выглядит гораздо понятнее:

```
long Area(int length, int width);
```

Подобно тому, как это сделано в библиотеке стандартных функций компилятора, следует поступать и при разработке своих программ, состоящих из достаточно большого количества функций, размещенных в разных модулях. Прототипы функций и описания внешних объектов (переменных, массивов и т.д.) помещают в отдельный файл, который препроцессорной командой

```
#include "имя_файла"
```

включают в начало каждого из модулей программы. В отличие от библиотечных функций компилятора имя такого заголовочного файла в команде **#include** записывается не в угловых скобках <>, а в кавычках "". При этом не нужно беспокоиться об увеличении размеров создаваемой программы. Прототипы функций нужны только на этапе компиляции и не переносятся в объектный модуль, т.е. не увеличивают машинного кода. А прототипы тех функций, которые не вызываются в модуле, вообще не используются компилятором.

Подставляемые inline функции

Некоторые функции в языке Си++ можно определить с использованием специального служебного слова **inline** . Спецификатор inline позволяет определить функцию как встраиваемую или подставляемую, или "инлайн-функцию. Например, следующая функция определена как подставляемая,

```
inline float module (float x = 0, float y = 0)
{    return sqrt (x*x + y*y);  }
```

Функция module () возвращает значение типа float . Обрабатывая каждый вызов встраиваемой функции, компилятор "пытается" подставить в текст программы код операторов ее тела. Спецификатор inline в общем случае не влияет на результаты вызова функции, она имеет обычный синтаксис определения и описания, подчиняется всем правилам контроля типов и области действия. Однако вместо команд передачи управления единственному экземпляру тела функции компилятор в каждое место вызова функции помещает соответствующим образом настроенные команды кода операторов тела функции. Тем самым при многократных вызовах подставляемой функции размеры программы могут увеличиться, однако исключаются затраты на передачи управления к вызываемой функции и возвраты из нее. Кроме экономии времени при выполнении программы, подстановка функции позволяет проводить оптимизацию ее кода.

Наиболее эффективно использовать подставляемые функции в тех случаях, когда тело функции состоит всего из нескольких операторов. Идеальными претендентами на определение со спецификатором inline являются несложные короткие функции. Удобны для подстановки функции, основное назначение которых - вызов других функций либо выполнение преобразований типов. Так как компилятор встраивает код подставляемой функции вместо ее вызова, то определение функции со спецификатором inline должно находиться в том же модуле, что и обращение к ней, и размещается до первого вызова.

Следующий случай, когда подстановка для функции со спецификатором inline проблематична,

- вызов этой функции с помощью указателя на нее. Реализация такого вызова, как правило, будет

Причины, по которым функция со спецификатором inline будет трактоваться как обычная функция (не подставляемая):

- встраиваемая функция слишком велика, чтобы выполнить ее подстановку;
- встраиваемая функция рекурсивна;
- обращение к встраиваемой функции в программе размещено после ее определения;
- встраиваемая функция вызывается более одного раза в выражении;
- встраиваемая функция содержит цикл, переключатель или оператор перехода

Если же для функции со спецификатором inline компилятор не может выполнить подстановку из-за контекста, в который помещено обращение к ней, то функция считается статической (static) и выдается предупреждающее сообщение.

```
inline int even(int x) {
   return ! (x%2);
}
int main(int argc, char* argv[]) {
   int n=5;
   if (even(n))
   printf(" n " " является четным \n ");
   else
   printf( " n " " является нечетным \n");
  return 0 ;
}
```

В этом примере программа читает введенное целое число и сообщает, является ли оно четным. Функция even () объявлена как встраиваемая. Это означает, что оператор

if (even(n)) cout « n « "является четным\n";

эквивалентен следующему:

```
if (!(n%2))
cout « n « "является четным\n";
```

Класс памяти функции

Функция может иметь классы памяти: **extern** (внешний) и **static**(статический). Если класс памяти опущен, то по умолчанию предполагается внешний Класс памяти функции связан с ее областью действия. Внешний класс памяти означает, что функция является глобальной, т.е. она будет доступна как в данном файле, так и в других отдельно скомпилированных файлах.

Статический класс памяти означает, что функция будет доступна только в данном файле, в котором она определена.

Для главной функции main() класс памяти в ее определении отсутствует, по умолчанию подразумевается внешний глобальный класс памяти.

Перегрузка функций

Цель перегрузки функции состоит в том, чтобы функция с одним именем по-разному выполнялась и возвращала разные значения при обращении к ней с разными по типам и количеству фактическими параметрами. Например, может потребоваться функция, возвращающая максимальное из значений элементов одномерного массива, передаваемого ей в качестве параметра. Массивы, использованные как фактические параметры, могут содержать элементы разных типов, но пользователь функции не должен беспокоиться о типе результата.

Функция всегда должна возвращать значение того же типа, что и тип массива - фактического параметра. Для обеспечения перегрузки функций необходимо для каждого имени определить, сколько разных функций связано с ним, т. е. сколь ко вариантов сигнатур допустимы при обращении к ним. Предположим, что функция выбора максимального значения элемента из массива должна работать для массивов типа int , long , float , double . В этом случае придется написать четыре разных варианта функции с одним и тем же именем. В следующей программе эта задача решена:

```
#include <iostream.h>
long max_element(int n, int array[]) { // Функции для
// массивов с элементами типа int
    int value = array[0];
     for (int i =1; i < n; i++)
    value = value > array[i] ? value : array[i];
    cout << "\nДля (int) : ";
    return long(value);
}
long max_element(int n, long array[]) { // Функция для
   // массивов с элементами типа long
 long value = array[0];
 for (int i = 1; i < n; i++)
    value = value > array[i] ? value : array[i]:
    cout << "\nДля (long) : ";
    return value;
}
```

```
double max_element(int n, float array[]) {
// Функция для массивов с элементами типа double
 float value = array [0];     for (int i = 1; i < n; i++)
    value = value > array[i] ? value : array[i];
    cout << "\n Для (float) : ";
    return double(value);
}
double max_element(int n, double array[]) {
// Функция для массивов с элементами типа double
    double value = array[0];
     for (int i=l; i < n; i++)
     value = value > array[i] ? value : array[i] ;
    cout << "\nДля (double) : ";
    return value;
}
```

```
void main() {
   int x[ ] = { 10, 20, 30, 40, 50, 60 };
   long f [ ] = { 12L, 44L, 5L, 22L, 37L, 30L };
   float y[ ] = { 0.1, 0.2, 0.3, 0.4, 0.5, 0.6 } ;
   double z[ ] = { 0.01, 0.02, 0.03, 0.04, 0.05, 0.06 };
   cout << "max_elem (6,x) = " << max_element(6,x);
   cout << "max_elem (6,f) = " < <max_element(6,f);
   cout << "max_elem (6,y) = " << max_element(6,y);
   cout << "max_elem (6,z) = " << max_element(6,z);
}
```

Результат работы программы:

Для (int) : max_elem(6,x)=60
Для (long) : max_elem(6,f)=44
Для (float) : max_elem(6,y)=0.6
Для (double) : max_elem(6,z)=0.06

В программе для иллюстрации независимости перегруженных функции от типа возвращаемого значения две функции, обрабатывающие целые массивы (int , long), возвращают значение одного типа long ; функции, обрабатывающие вещественные массивы (double , float), обе возвращают значение типа double . Распознавание перегруженных функций при вызове выполняется по их сигнатурам. Сигнатуру функции определяет совокупность формальных параметров. Сигнатура зависит от количества параметров, от их типов и от порядка их размещения в спецификации формальных параметров. Перегруженные функции поэтому должны иметь одинаковые имена, но спецификации их параметров должны различаться по количеству и (или) по типам, и(или) по расположению

При использовании перегруженных функций нужно с осторожностью задавать начальные значения их аргументов. Например, если в одной программе перегружены функции

```
int sum(int a, int b=1)  { return(a+b); }
int sum(int a)           { return(a+a); }
```

то вызов

```
 int r = sum(2);  // ошибка
```

выдаст ошибку из-за неоднозначности толкования sum(), т.е компилятор не сможет принять решение, вызывать функцию

```
sum(2, b=1) {return (2+1);}
```

или

```
 sum(2) {return (2+2);}
```

Шаблон функций

Цель введения шаблонов функций - автоматизация создания функций, которые могут обрабатывать разнотипные данные. В отличие от механизма перегрузки, когда для каждой сигнатуры определяется своя функция, шаблон семейства функций определяется один раз, но это определение параметризуется. Параметризовать в шаблоне функций можно тип возвращаемого функцией значения и типы любых параметров, количество и порядок размещения которых должны быть фиксированы. Для параметризации используется список параметров шаблона.

В определении шаблона семейства функций используется служебное слово **template .** Для параметризации используется список формальных параметров шаблона, который заключается в угловые скобки **<>.** Каждый формальный параметр шаблона обозначается служебным словом **class ,** за которым следует имя параметра (идентификатор).

Шаблон семейства функции состоит из двух частей - заголовка шаблона:

template <список параметров шаблона>

и из обыкновенного определения функции, в котором тип возвращаемого значения и типы любых параметров обозначаются именами параметров шаблона, введенных в его заголовке. Те же имена параметров шаблона могут использоваться и в теле определения функции для обозначения типов локальных объектов.

Пример определения шаблона функций, вычисляющих абсолютные значения числовых величин разных типов:

```
template <class type>
type abs (type x)
 { return x > 0 ? x : -x; }
```

Шаблон семейства функций служит для автоматического формирования конкретных определений функций по тем вызовам, которые транслятор обнаруживает в тексте программы. Например, при обращении **abs (-10.3),** компилятор сформирует такое определение функции:

double abs (double x) (return x > 0 ? x: - x;)

Далее будет организовано выполнение именно этой функции и в точку вызова в качестве результата вернется числовое значение 10.3.

Пример шаблона семейств функций для обмена значений двух передаваемых им параметров:

```
template <class T >
void swap (T* x, T* y)
{ T z = *x; *x = *y; *y = z; }
```

Здесь параметр T шаблона функций используется не только в заголовке для спецификации формальных параметров, но и в теле определения функций, где он задает тип вспомогательной переменной z .

Если в программе присутствует приведенный выше шаблон семейства функций swap () и появится последовательность операторов:

```
long k= 4, d = 8; swap (&k, &d);
```

то компилятор сформирует определение функции:

```
void swap (long* x, long* y) {
    long z = *x;
    *x = *y;
    *y = z;
}
```

Затем будет выполнено обращение именно к этой функции и значения переменных k, d поменяются местами.

Если в той же программе присутствуют операторы:

```
double a = 2.44, b = 66.3; swap (&a, &b);
```

то сформируется и выполнится функция

```
void swap (double* x, double* y ) {
    double z = *x;
    *x = *y;
    *y =z;
}
```

Основные свойства параметров шаблона.

1. Список параметров шаблона функций не может быть пустым, так как при этом теряется возможность параметризации и шаблон функций становится обычным определением конкретной функции.
2. В списке параметров шаблона функций может быть несколько параметров. Каждый из них должен начинаться со служебного слова class . Например, допустим такой заголовок шаблона:

template <class type1, class type2>

Соответственно, неверен заголовок:

template <class type1, type2, type3>

3. Недопустимо использовать в заголовке шаблона параметры с одинаковыми именами, т.е. ошибочен такой заголовок:

template <class t, class t, class t>

4. Имя параметра шаблона видно во всем определении и скрывает другие использования того же идентификатора в области, глобальной по отношению к данному шаблону функций. Если внутри тела определяемой функции необходим доступ к внешним объектам с тем же именем, нужно применять операцию изменения области видимости

```
#include <iostream.h>
int n; // Инициализирована по умолчанию нулевым значением
// Функция определяет максимальное из двух значений параметров
template <class N>
N max(N x, N y)
{n++;
    N a = x;
    cout << "\n Счетчик обращений n = " << n;
    if (a < y) a = y;
    return a;
}
void main() {
    int a = 12, b = 42;
   cout << "\n  max = " << max(a, b);
    float z = 66.3, f = 222.4;
    cout << "\n  max = " <<  max(z, f);
}
```

Результат выполнения программы:

Счетчик обращений n = 1

max =42

Счетчик обращений n = 2

Max=222.4

Как и при работе с обычными функциями, для шаблонов функций существуют определения и описания. В качестве описания шаблона функций используется прототип шаблона:

template <список параметров шаблона>

В списке параметров прототипа шаблона имена параметров не обязаны совпадать с именами тех же параметров в определении шаблона.

Необходимо, чтобы при вызове функции типы фактических параметров, соответствующие одинаково параметризованным формальным параметрам, были одинаковыми. Например, для шаблона функций с прототипом

template <class E> void swap(E, E);

недопустимо использовать такое обращение к функции:

int n = 4; double d = 4.3;
swap (n, d); // Ошибка в типах параметров

Для правильного обращения к такой функции требуется явное приведение типа одного из параметров. Например, вызов

swap (double(n), d); // Правильные типы параметров

```
// Листинг 5.1. Использование прототипов функций
int
#include <iostream.h>
int Area(int length, int width); //прототип функции

int main()
{
  int lengthOfYard;
  int widthOfYard;
  int areaOfYard;

  cout << "\ nHow wide is your yard? ";
  cin >> widthOfYard;
  cout << "\ nHow long is your yard? ";
  cin >> lengthOfYard;

  areaOfYard= Area(lengthOfYard,widthOfYard);

  cout << "\ nYour yard is ";
  cout << areaOfYard;
  cout << " square feet\ n\ n";
            return 0;
}

  int Area(int yardLength, int yardWidth)
{
     return yardLength * yardWidth;
}
```

```
How wide is your yard? 100

How long is your yard? 200

Your yard is 20000 square feet
```

Прототип функции `Area()` объявляется в строке 4. Сравните прототип с определением функции, представленным в строке 25. Обратите внимание, что их имена, типы возвращаемых значений и типы параметров полностью совпадают. Если бы они были различны, то компилятор показал бы сообщение об ошибке. Единственное обязательное различие между ними состоит в том, что прототип функции оканчивается точкой с запятой и не имеет тела.

Обратите также внимание на то, что имена параметров в прототипе — `length` и `width` — не совпадают с именами параметров в определении: `yardLength` и `yardWidth`. Как упоминалось выше, имена в прототипе не используются; они просто служат описательной информацией для программиста. Соответствие имен параметров прототипа именам параметров в определении функции считается хорошим стилем программирования; но это не обязательное требование.

Аргументы передаются в функцию в порядке объявления и определения параметров, но без учета какого бы то ни было совпадения имен. Если в функцию `Area()` первым передать аргумент `widthOfYard`, а за ним — аргумент `lengthOfYard`, то эта функция использует значение `widthOfYard` для параметра `yardLength`, а значение `lengthOfYard` — для параметра `yardWidth`. Тело функции всегда заключается в фигурные скобки, даже если оно состоит только из одной строки, как в нашем примере.

Использование функций в качестве параметров функций

Несмотря на то что вполне допустимо для одной функции принимать в качестве параметра вторую функцию, которая возвращает некое значение, такой стиль программирования затрудняет чтение программы и ее отладку.

В качестве примера предположим, что у вас есть функции `double()`, `triple()`, `square()` и `cube()`, возвращающие некоторое значение. Вы могли бы записать следующую инструкцию:

```
Answer = (double(triple(square(cube(myValue)))));
```

Эта инструкция принимает переменную `myValue` и передает ее в качестве аргумента функции `cube()`, возвращаемое значение которой (куб числа) передается в качестве аргумента функции `square()`. После этого возвращаемое значение функции `square()` (квадрат числа), в свою очередь, передается в качестве аргумента функции `triple()`. Затем значение возврата функции `triple()` (утроенное число) передается как аргумент функции `double()`. Наконец, значение возврата функции `double()` (удвоенное число) присваивается переменной `Answer`.

Вряд ли можно с полной уверенностью говорить о том, какую задачу решает это выражение (было ли значение утроено до или после вычисления квадрата?); кроме того, в случае неверного результата выявить "виноватую" функцию окажется весьма затруднительно.

В качестве альтернативного варианта можно было бы каждый промежуточный результат вычисления присваивать промежуточной переменной:

```
unsigned long myValue = 2;
unsigned long cubed =  cube(myValue);      // 2 в кубе = 8
unsigned long squared = square(cubed);     // 8 в квадрате = 64
unsigned long tripled = triple(squared);   // 64 * 3 = 192
unsigned long Answer = double(tripled);    // 192 *2 = 384
```

Теперь можно легко проверить каждый промежуточный результат, и при этом очевиден порядок выполнения всех вычислений.

Рекурсия

Функция может вызывать самое себя. Это называется *рекурсией*, которая может быть прямой или косвенной. Когда функция вызывает самое себя, речь идет о прямой рекурсии. Если же функция вызывает другую функцию, которая затем вызывает первую, то в этом случае имеет место косвенная рекурсия.

Некоторые проблемы легче всего решаются именно с помощью рекурсии. Так рекурсия полезна в тех случаях, когда выполняется определенная процедура над данными, а затем эта же процедура выполняется над полученными результатами. Оба типа рекурсии (прямая и косвенная) выступают в двух амплуа: одни в конечном счете заканчиваются и генерируют возврат, а другие никогда не заканчиваются и генерируют ошибку времени выполнения. Программисты считают, что последний вариант весьма забавен (конечно же, когда он случается с кем-то другим).

Важно отметить, что, когда функция вызывает самое себя, выполняется новая копия этой функции. При этом локальные переменные во второй версии независимы от локальных переменных в первой и не могут непосредственно влиять друг друга, по крайней мере не больше, чем локальные переменные в функции `main()` могут влиять на локальные переменные в любой другой функции, которую она вызывает

Пример. Алгоритм вычисления факториала- 6!;

```
 include <stdio.h>
double fact(int n);
int main ()
{
int n=6;
double f;
f=fact(n);
printf("6!=%10.0f\n",f);
return (0);
}
double fact(int n)
{
if (n<1) return(1.0);
 else
return (n*fact(n-1);
}
```

// Найти наибольший общий делитель двух чисел по алгоритму Евклида :

```
 include <stdio.h>
 include <math.h>
double euob(double n, double m);
int main ()
{
double f; double n=1470; double m=693; f=euob(n,m);
pnntf("f=%10.0f \n", f);
return(0);
}

double euob(double n, double m)
{
w=floor(fmod(n,m);
if (w=0) return(m);
 else return(euob(m,w));
}
```

Чтобы показать пример решение проблемы с помощью рекурсии, рассмотрим ряд Фибоначчи:

1, 1, 2, 3, 5, 8, 13, 21, 34...

Каждое число ряда (после второго) представляет собой сумму двух стоящих впереди чисел. Задача может состоять в том, чтобы, например, определить 12-й член ряда Фибоначчи.

Один из способов решения этой проблемы лежит в тщательном анализе этого ряда. Первые два числа равны 1. Каждое последующее число равно сумме двух предыдущих. Таким образом, семнадцатое число равно сумме шестнадцатого и пятнадцатого. В общем случае *n*-е число равно сумме *(n-2)*-го и *(n-1)*-го при условии, если $n > 2$.

Для рекурсивных функций необходимо задать условие прекращения рекурсии. Обязательно должно произойти нечто, способное заставить программу остановить рекурсию, или же она никогда не закончится. В ряду Фибоначчи условием останова является выражение `n < 3`.

При этом используется следующий алгоритм:

1. Предлагаем пользователю указать, какой член в ряду Фибоначчи следует рассчитать.
2. Вызываем функцию `fib()`, передавая в качестве аргумента порядковый номер члена ряда Фибоначчи, заданный пользователем.
3. В функции `fib()` выполняется анализ аргумента (`n`). Если `n < 3`, функция возвращает значение `1`; в противном случае функция `fib()` вызывает самое себя (рекурсивно), передавая в качестве аргумента значение `n-2`, затем снова вызывает самое себя, передавая в качестве аргумента значение `n-1`, а после этого возвращает сумму.

Если вызвать функцию `fib(1)`, она возвратит 1. Если вызвать функцию `fib(2)`, она также возвратит 1. Если вызвать функцию `fib(3)`, она возвратит сумму значений, возвращаемых функциями `fib(2)` и `fib(1)`. Поскольку вызов функции `fib(2)` возвращает значение 1 и вызов функции `fib(1)` возвращает значение 1, то функция `fib(3)` возвратит значение 2.

Если вызвать функцию `fib(4)`, она возвратит сумму значений, возвращаемых функциями `fib(3)` и `fib(2)`. Мы уже установили, что функция `fib(3)` возвращает значение 2 (путем вызова функций `fib(2)` и `fib(1)`) и что функция `fib(2)` возвращает значение 1, поэтому функция `fib(4)` просуммирует эти числа и возвратит значение 3, которое будет являться четвертым членом ряда Фибоначчи.

Сделаем еще один шаг. Если вызвать функцию `fib(5)`, она вернет сумму значений, возвращаемых функциями `fib(4)` и `fib(3)`. Как мы установили, функция `fib(4)` возвращает значение 3, а функция `fib(3)` — значение 2, поэтому возвращаемая сумма будет равна числу 5.

Описанный метод — не самый эффективный способ решения этой задачи (пр
вызове функции `fib(20)` функция `fib()` вызывается 13 529 раз!), тем не менее он рабо
тает. Однако будьте осторожны. Если задать слишком большой номер члена ряда Фи
боначчи, вам может не хватить памяти. При каждом вызове функции `fib()` резервиру
ется некоторая область памяти. При возвращении из функции память освобождается
Но при рекурсивных вызовах резервируются все новые области памяти, а при тако
подходе системная память может исчерпаться довольно быстро. Реализация функци
`fib()` показана в листинге 5.10.

Листинг 5.18. Пример использования рекурсии для нахождения члена ряда Фибоначчи

```
1: #include <iostream.h>
2:
3: int fib (int n);
4:
5:  int main()
6:  {
7:
8:   int n, answer;
9:   cout << "Enter number to find: "; 10:    cin >> n;
10:
11:   cout << "\ n\ n";
12:
13:   answer = fib(n);
14:
15:   cout << answer << " is the " << n << "th Fibonacci number\ n"; 17: return 0;
16:  }
17:
18:  int fib (int n)
19:  {
20:   cout << "Processing fib(" << n << ")... "; 23:
21:   if (n < 3 )
22:   {
23:     cout << "Return 1!\ n";
24:     return (1);
25:   }
26:   else
27:   {
28:    cout << "Call fib(" << n-2 << ") and fib(" << n-1 << ").\ n";
29:    return( fib(n-2) + fib(n-1));
30:   }
31:  }
```

Результат

```
Enter number to find: 6

Processing fib(6)... Call fib(4) and fib(5).
Processing fib(4)... Call fib(2) and fib(3).
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(5)... Call fib(3) and fib(4).
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(4)... Call fib(2) and fib(3).
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
8 is the 6th Fibonacci number
```

В строке 9 программа предлагает ввести номер искомого члена ряда и присваивает его переменной n. Затем вызывается функция fib() с аргументом n. Выполнение программы переходит к функции fib(), где в строке 20 этот аргумент выводится на экран.

В строке 21 проверяется, не меньше ли аргумент числа 3, и, если это так, функция fib() возвращает значение 1. В противном случае выводится сумма значений, возвращаемых при вызове функции fib() с аргументами n-2 и n-1. Таким образом, эту программу можно представить как циклический вызов функции fib(), повторяющийся до тех пор, пока при очередном вызове этой функции не будет возвращено некоторое значение. Единственными вызовами, которые немедленно возвращают значения, являются вызовы функций fib(1) и fib(2). Рекурсивное использование функции fib() проиллюстрировано на рис. 5.4 и 5.5.

В примере, изображенном на рисунках, переменная n равна значению 6, поэтому из функции main() вызывается функция fib(6). Выполнение программы переходит в тело функции fib(), и в строке 30 значение переданного аргумента сравнивается с числом 3. Поскольку число 6 больше числа 3, функция fib(6) возвращает сумму значений, возвращаемых функциями fib(4) и fib(5):

```
38:     return( fib(n-2) + fib(n-1));
```

Это означает, что выполняется обращение к функциям fib(4) и fib(5) (поскольку переменная n равна числу 6, то fib(n-2) — это то же самое, что fib(4), а fib(n-1) — то же самое, что fib(5)). После этого функция fib(6), которой в текущий момент пере-

дано управление программой, *ожидает*, пока сделанные вызовы не возвратят какое-нибудь значение. Дождавшись возврата значений, эта функция возвратит результат суммирования этих двух значений.

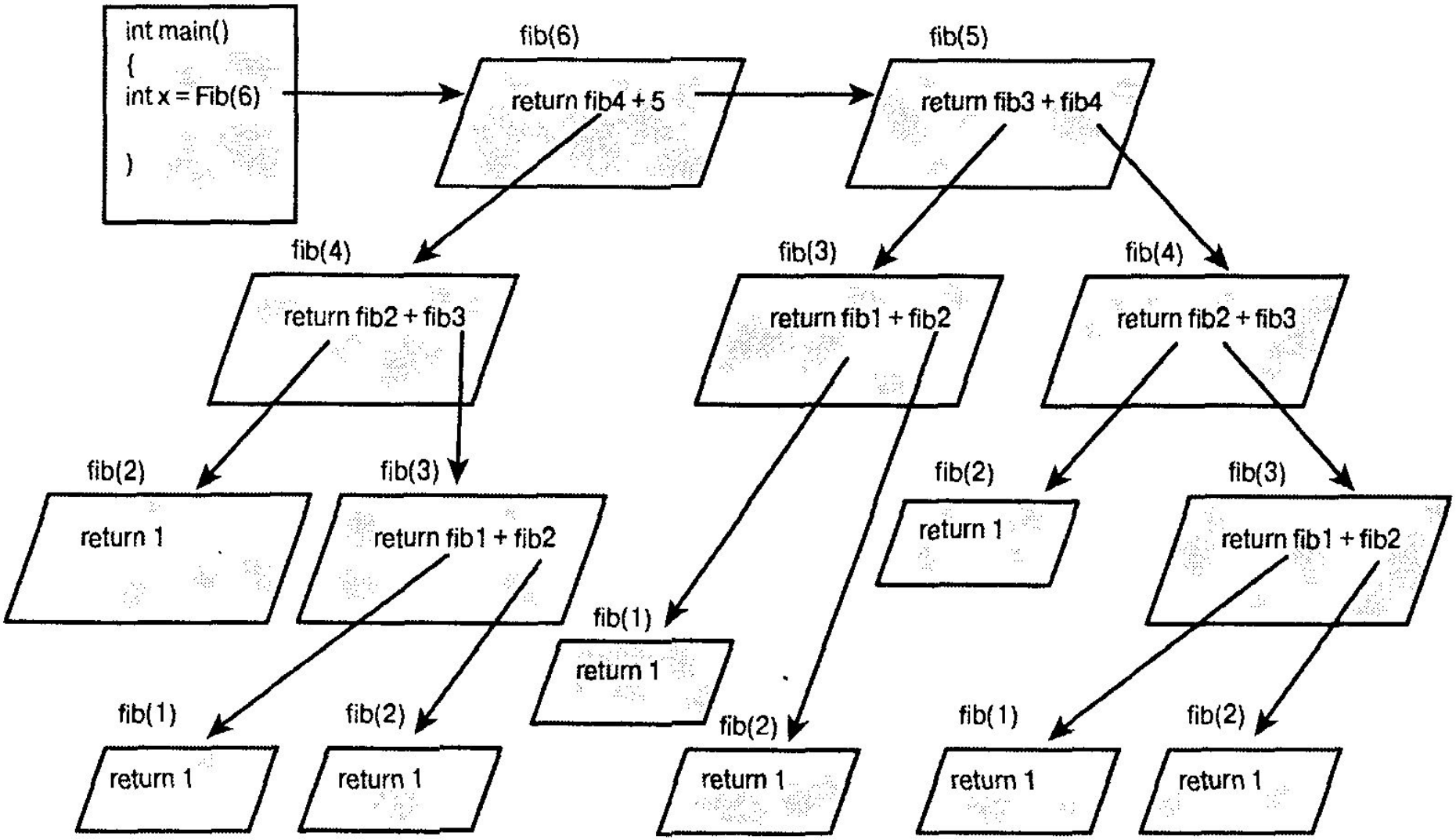


Рис. 5.4. Использование рекурсии

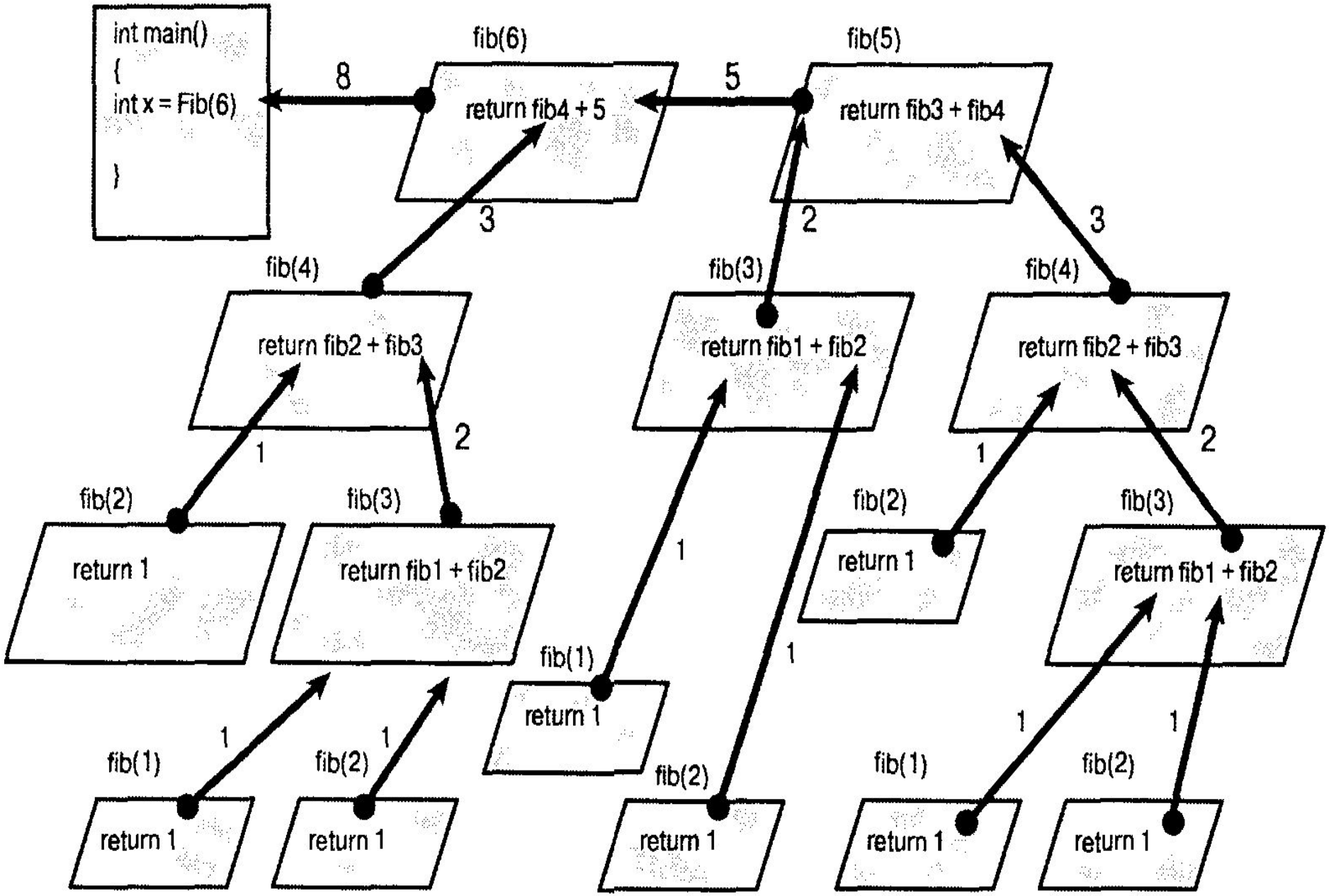


Рис. 5.5. Возвращение из рекурсии

Поскольку при вызове функции `fib(5)` передается аргумент, который не меньше числа 3, функция `fib()` будет вызываться снова, на этот раз с аргументами 4 и 3. А функция `fib(4)` вызовет, в свою очередь, функции `fib(3)` и `fib(2)`.

Результаты и промежуточные этапы работы программы, представленной в листинге 5.10, выводятся на экран. Скомпилируйте, скомпонуйте и выполните эту программу, введя сначала число 1, затем 2, 3, и так доберитесь до числа 6, внимательно наблюдая за отображаемой информацией.

Работа с этой программой предоставляет вам прекрасный шанс проверить возможности своего отладчика. Разместите точку останова в строке 20, а затем заходите в тело каждой вызываемой функции `fib()`, отслеживая значение переменной `n` при каждом рекурсивном вызове функции `fib()`.

В программировании на языке C++ рекурсия не частый гость, но в определенных случаях она является мощным и весьма элегантным инструментом.

Рекурсия относится к одной из самых сложных тем программирования. Данный раздел полезен для понимания основных идей ее реализации, однако не следует слишком расстраиваться, если вам не до конца ясны все детали работы рекурсии.

Функции с переменным числом параметров

В языке C++ можно применять функции, у которых до выполнения программы число параметров заранее не известно. Количество и, соответственно, типы параметров становятся известными при вызове функции. Такие функции называются *функциями* ***с переменным числом параметров***. Формат описания функции с переменным числом параметров следующий:

```
тип имя (список_явных_ параметров, ...)
  {
    тело_функции
  }
```

Здесь:

тип – тип возвращаемого значения;

список_явных_параметров – список параметров, которые известны до вызова функции.

После списка явных параметров ставится необязательная запятая и многоточие, которое сообщает компилятору о том, что контроль типов и количества параметров при вызове функции проводить не следует.

При программировании функций с переменным числом параметров необходимо предусмотреть механизм определения количества параметров и их типов. При реализации этого механизма используется два следующих подхода:

- один из параметров определяет число дополнительных параметров функции;
- в список явных параметров последним задается параметр-индикатор, указывающий на окончание списка параметров.

Переход от одного параметра к другому в обоих случаях осуществляется с помощью указателей.

Пример 1. Задание числа дополнительных параметров функции с помощью одного из параметров.

Пусть требуется составить программу, содержащую функцию с переменным числом параметров. Функция должна вычислять сумму значений дополнительных параметров. Список явных параметров должен состоять из одного параметра, используемого для задания числа дополнительных параметров.

```
/* Функции с переменным числом параметров */
#include <iostream.h>
int sum(int, ...);   // прототип функции
void main()
  {
    cout << "\n 4+6=" << sum(2,4,6);
    cout << "\n 1+2+3+4+5+6=" << sum(6,1,2,3,4,5,6);
    cout << "\n Параметры отсутствуют. Сумма равна:" << sum(0);
  }
```

```
int sum(int n, ...)   // n - число суммируемых параметров
  {
    int *p=&n;  // выход на начало списка дополнительных параметров
    int s=0;
    for (int i=1;i<=n;i++)
      {
        if (n==0) break;
        s+=*(++p);
      }
    return s;
  }
```

Результат выполнения программы:

```
4+6=10
1+2+3+4+5+6=21
Параметры отсутствуют. Сумма равна: 0
```

В приведенном примере для организации анализа аргументов используется значение первого параметра. Для доступа к нему используется указатель *p*. Ему присваивается значение адреса первого аргумента, который соответствует явно заданному параметру *n*. Таким образом указатель устанавливается на начало списка аргументов в памяти. Затем в цикле указатель *p* перемещается по адресам следующих аргументов, соответствующим дополнительным параметрам. С помощью операции разыменования **p* выполняется выборка и суммирование значений аргументов.

Замечание.

Особенность рассмотренного подхода заключается в том, что все параметры (явный и дополнительные) должны иметь одинаковый тип *int* или *double*.

Пример 2. Определение конца списка дополнительных параметров функции с помощью параметра-индикатора.

Пусть требуется составить программу, содержащую функцию с переменным числом параметров. Функция должна вычислять произведение значений дополнительных параметров. Для определения конца списка параметров используем параметр-индикатор.

```
#include <iostream.h>
double pr(double a...)
  {
    double b=1.0;          // b -  произведение
    double *p=&a;
    if(*p==0.0) return 0.0;
    for(; *p; p++)  b* =*p;
    return b;
  }
```

```
void main()
  {
    cout << "\n pr(6e0,5e0,3e0,0e0)=" << pr(6e0,5e0,3e0,0e0);
    cout << "\n pr(1.5,2.0,0.0,3.0,0.0)=" << pr(1.5,2.0,0.0,3.0,0.0);
    cout << "\n pr(0.0)=" << pr(0.0);
  }
```

Результат выполнения программы:

```
pr(6e0,5e0,3e0,0e0)=90
pr(1.5,2.0,0.0,3.0,0.0)=3
pr(0.0)=0
```

В функции *pr()* перемещение по списку аргументов осуществляется за счет изменения значения указателя *p*. При втором обращении к функции в середине списка аргументов находится аргумент с нулевым значением. Он воспринят функцией как индикатор, который сигнаизирует об окончании списка аргументов. В приведенной программе, так же как и в первом примере, аргументы должны иметь один тип, но не обязательно *int* или *double*.

Для использования в программе функций с переменным числом параметров, при вызове которых можно использовать аргументы разных типов, необходимо включать в текст программы специальный набор макроопределений. Эти макроопределения находятся заголовочном файле *stdarg.h*.

СТРУКТУРЫ И ФУНКЦИИ

Если функция не изменяет структуру, вы можете передать структуру в функцию по имени. Например, следующая программа SHOW_EMP.CPP использует функцию *show_employee* для вывода элементов структуры типа *employee*:

```
#include <iostream.h>
#include <string.h>

struct employee {
  char name[64];
  long employee_id;
  float salary;
  char phone[10];
  int office_number;

 };
```

```
void show_employee(employee worker)
 {
   cout << "Служащий: " << worker.name << endl;
   cout << "Телефон: " << worker.phone << endl;
   cout << "Номер служащего: " << worker.employee_id << endl;
   cout << "Оклад: " << worker.salary << endl;
   cout << "Офис: " << worker.office_number << endl;
 }

void main(void)
 {
   employee worker;

    // Копировать имя в строку
   strcpy(worker.name, "Джон Дой");
```

```
    worker.employee_id = 12345;
    worker.salary = 25000.00;
    worker.office_number = 102;

    // Копировать номер телефона в строку
    strcpy(worker.phone, "555-1212");
    show_employee(worker);
}
```

Как видите, программа передает переменную типа данной структуры *worker* в функцию *show_employee* по имени. Далее функция *show_employee* выводит элементы структуры. Однако обратите внимание, что программа теперь определяет структуру *employee* вне *main* и до функции *show_employee*. Поскольку функция объявляет переменную *worker* типа *employee*, определение структуры *employee* должно располагаться до функции.

Функции, изменяющие элементы структуры

Как вы знаете, если функция изменяет параметр, вам следует передавать этот параметр в функцию с помощью адреса. Если функция изменяет элемент структуры, вы должны передавать эту структуру в функцию с помощью адреса. Для передачи переменной типа структуры с помощью адреса вы просто предваряете имя переменной оператором адреса C++ (&), как показано ниже:

```
some_function(&worker);
```

Внутри функции, которая изменяет один или несколько элементов, вам следует работать с указателем. Если вы используете указатель на структуру, легче всего обращаться к элементам структуры, используя следующий синтаксис:

```
pointer_variable->member = some_value;
```

Например, следующая программа CHG_MBR.CPP передает структуру типа *employee* в функцию с именем *get_employee_id*, которая запрашивает у пользователя идентификационный номер служащего и затем присваивает этот номер элементу структуры *employee_id*. Чтобы изменить элемент, функция работает с указателем на структуру:

```
#include <iostream.h>
#include <string.h>

struct employee {
  char name[64];
  long employee_id;
  float salary;
  char phone[10];
  int office_number;
 };
```

```
void get_employee_id(employee *worker)
 {
   cout << "Введите номер служащего: ";
   cin >> worker->employee_id;
 }

void main(void)
 {
   employee worker;

   // Копировать имя в строку
   strcpy(worker.name, "Джон Дой");
   get_employee_id(&worker);

   cout << "Служащий: " << worker.name << endl;
   cout << "Номер служащего: " << worker.employee_id << endl;
 }
```

Как видите, внутри *main* программа передает переменную *worker* типа структуры в функцию *get_employee_id* с помощью адреса. Внутри функции *get_employee_id* значение, введенное пользователем, присваивается элементу *employee_id* с помощью следующего оператора:

```
cin >> worker->employee_id;
```

Рекурсивные и подставляемые функции

В инженерной практике приходится иногда программировать рекурсивные алгоритмы. Такая необходимость возникает при реализации динамических структур данных, таких как стеки, деревья, очереди. Для реализации рекурсивных алгоритмов в C++ предусмотрена возможность создания рекурсивных функций. ***Рекурсивная функция*** представляет собой функцию, в теле которой осуществляется вызов этой же функции.

Пример 1. Использование рекурсивной функции для вычисления факториала.

Пусть требуется составить программу вычисления факториала произвольного положительного числа.

```
#include <iostream.h>
int fact(int n)
  {
    int a;
    if (n<0) return 0;
    if (n==0) return 1;
    a =n * fact(n-1);
    return a;
  }
void main()
  {
    int m;
    cout << "\nВведите целое число:";
    cin >> m;
    cout << "\n Фактериал числа " << m << " равен " << fact(m);
  }
```

Для отрицательного аргумента факториала не существует, поэтому функция в этом случае возвращает нулевое значение. Так как факториал 0 равен 1 по определению, то в теле функции предусмотрен и этот вариант. В случае когда аргумент функции *fact()* отличен от 0 и 1, то вызывается функция *fact()* с уменьшенным на единицу значением параметра и результат умножается на значение текущего параметра. Таким образом, в результате встроенных вызовов функций будет возвращен следующий результат:

```
n * (n-1) * (n-2) * . . . * 2 * 1 * 1
```

Последняя единица при формировании результата принадлежит вызову *fact(0)*.

Поскольку в языке C++ отсутствует операция возведения в степень, то для выполнения возведения в степень вещественного ненулевого числа можно использовать рекурсивную функцию.

Пример 2. Использование рекурсивной функции для возведения в степень.

Пусть требуется составить программу возведения в положительную целую степень вещественного ненулевого числа.

```
/* Возведение основания x в степень n */
#include <iostream.h>
float st(float x,int n)
 {
   if (n==0) return 1;
   if (x==0) return 0;
   if (n>0) return x*st(x,n-1);
   if (n<0) return st(x,n+1)/x;
 }
void main()
 {
   int m;
   float a,y;
   cout << "\nВведите основание степени:";
   cin >> a;
   cout << "\nВведите степень числа:";
   cin >> m;
   y=st(a,m);
   cout << "\n Число " << a << " в степени " << m << "  равно: " << y;
 }
```

В функции предусмотрены четыре ветви выполнения возведения в степень вещественного числа:

- степень числа равна нулю;
- основание равно нулю;
- степень больше нуля;
- степень меньше нуля.

При вызове функции, когда основание степени больше нуля, например, если при вводе заданы значения переменных a=2.0 и m=4, тогда обращение *st(a,m)* приводит к следующему вычислению:

```
2.0 * 2.0 * 2.0 * 2.0 * 1
```

Вызов функции для отрицательной степени, например, когда значения аргументов равны a=3.0 и m=–2 приводит к вычислению следующего выражения:

```
1/3.0/3.0
```

Резюме

На этом занятии вы познакомились с функциями. Функция в действительности представляет собой подпрограмму, которой можно передавать параметры и из которой можно возвращать значение. Каждый запуск программы C++ начинается с выполнения функции `main()`, которая, в свою очередь, может вызывать другие функции.

Функция объявляется с помощью прототипа функции, который описывает возвращаемое значение, имя функции и типы ее параметров. При желании функцию можно объявить подставляемой (с помощью ключевого слова `inline`). В прототипе функции можно также объявить значения, используемые по умолчанию для одного или нескольких параметров функции.

Определение функции должно соответствовать прототипу функции по типу возвращаемого значения, имени и списку параметров. Имена функций могут быть перегружены путем изменения количества или типа параметров. Компилятор находит нужную функцию на основе списка параметров.

Локальные переменные функции и аргументы, передаваемые функции, локальны по отношению к блоку, в котором они объявлены. Параметры, передаваемые как значения, представляют собой копии реальных переменных и не могут влиять на значения этих переменных в вызывающей функции.