2.6 C++ 函数：

函数是一个独立完成某种功能的程序块，它封装了一些程序代码和数据，实现了更高级的抽象和数据隐藏。使得编程者只关心函数的功能和使用方法，而不必关心函数功能的具体实现细节。由于函数与函数之间通过输入参数和返回值（输出）来联系，因此可以把函数看作一个“黑盒（black box）”，除了输入输出，其他什么都封装隐藏在“黑盒”内。这就象“我们只需了解怎样连接电源和天线及按钮操作等，就能操作电视机收看节目”一样，而电视机内部的电子元器件是如何工作的具体细节，都封装隐藏在“黑盒”内无须我们操心。

用函数可将大的任务（Task）分割成一个个小的任务。ANSI C++标准鼓励和提倡编程者把一个大的问题划分成许多小模块，每一个模块编写一个函数，一个函数完成一个功能单一而独立的任务。因此C++语言源程序通常是由许多小的函数组成，由多个小函数建立大函数。以后将会经常看到：一个函数只有2~3行语句，并且只被调用一次。

（一） 函数的定义和说明：

函数的定义格式：除了标准函数库外，任何函数都必须用如下格式加以定义：

```
<存储类> <类型> 函数名（类型 参数名1 ，
函数头                    。。。，类型 参数名 n）
函数体{
        若干条语句；
}
```

说明：

(1) 函数的存储类只有extern（外部）static（静态）。当函数的存储类说明缺省时，定义的函数为外部函数。外部函数可被任何源文件所调用。

(2) 函数的类型是函数返回值（也称函数值）的数据类型，可为各种基本数据类型(char、int、。。。。、double)和构造数据类型，其中包含指针和引用类型。

例如：

```
double sum_double(double x , double y)
{
      return x + y;
}
```

只有当函数的返回类型为int型时?，类型说明可缺省。例如：

```
sum(int x,int y) ⇔ int sum(int x,int y)
{
      return x + y;
}
```

当函数不需要获得返回值时，只是一个过程调用，则将函数的类型指定为“void”。例如：

```
void delay(long t)
{
      for(int i = 1; i < t; i ++)
            ;    //循环体为空语句
}                  //延迟一段时间
```


(3) 函数名是一种标识符，按标识符的规定命名，最好“见名知意”。由一对圆括号包围的的部分称为参数表，函数定义时或函数原型中的参数表称为形式参数表，简称形参表。形参表可由零个、一个或多个参数组成函数的输入界面，参数个数为零表示没有参数，但圆括号不能省。多个参数间应该用逗号分隔开来，每个参数包括参数名和参数类型说明。由类型说明、函数名和形参表构成了“函数头”，它既说明了函数的返回值，也说明了每个参数的数据类型。

(4) 由大括号括起来的程序部分称为函数体。函数定义部分的函数体，在句法上可看成一个复合语句。函数定义部分是由函数头和函数体组成。函数体内允许编写调用另一个函数的语句，也可以说明该函数所使用的局部变量、外部变量以及要使用的外部函数。

C++ 程序设计

例如：

```
double sh(double x)
{   extern double exp( );
return ((exp(x)- exp(-x)) / 2.0;
}
```

但不允许在函数体内定义另一个函数。例如：

```
void main( )
{   int x ;
    x = sub(20, 10);
sub(int a, int b)
{
    return a - b;
}
      。。。
}
```

main( )函数的定义部分

不允许定义另一个函数sub( )

C++ 程序设计

<若干条语句>可以是0条、1条或多条语句。当函数体是0条语句时，称该函数为空函数。空函数作为一种什么都不执行的函数有时也是有意义的。函数体内无论有多少条语句，大括号是不能省的。例如：

```
 void nothing( )
{     }        0条语句
```

2.函数的说明：函数的说明又称为函数的声明（declare）。在ANSI C++标准中，函数的说明原则如下：如果一个函数定义在先，调用在后，调用前可以不必说明；如果一个函数定义在后，调用在先，调用前必须用“函数原型”加以声明。

(1) 所谓“函数原型”就是“函数头；”向编译系统声明该函数。即：

```
<存储类> <类型> 函数名（类型 参数名1，。。。
                        ，类型 参数名 n）；
```


C++ 程序设计

(2) “函数原型”也可以不写出参数名，如：

```
<存储类><类型> 函数名（类型1，。。。，类型n）；
```

•在较简单的程序中，可将被调用函数的定义部分写在调用函数的前面，则可省略这类声明语句。

•对于比较大的源程序，编程者为了清晰可读，常在文件的开头、在所有函数的外部，集中用说明语句声明所有被调用的函数。或者自行编写一个头文件，例如menu.h，声明所有被调用的函数。那末在源程序的开头处必须写上：#include “menu.h”

C++ 程序设计

```
#include <iostream>
using namespace std;
double sum_double(double x , double y)
{
      return x + y;
}
void main( )
{   double a , b;
    cout << "Input a : ";
       cin >> a;
    cout << "Input b : ";
       cin >> b;
    double sum = sum_double(a,b);
    cout << "sum = " << sum << endl;
}
```

```
#include <iostream>
using namespace std;
sum(int   , int);
void main( )
{   int a , b;
    cout << "Input a : ";
        cin >> a;
    cout << "Input b : ";
        cin >> b;
        cout << "sum = " << sum(a , b) << endl;
}
sum(int x , int y)
{
      return x + y;
}
```


(3) 函数的存储类只有外部(extern)型和静态(static)型。若定义为extern型时，则在整个源程序的所有源文件内都可以调用它。关键字extern可以缺省.

(二)函数的调用：

1. 函数的调用格式：已定义的函数可以直接调用，其调用格式：

函数名（实参表）;

可以用以下三种函数调用方式：

(1)函数调用语句：把函数调用作为一个语句，如：

函数原型:void setcolor(int color);

调用语句：　　setcolor(RED);

(2)函数表达式：将函数调用语句放在一个表达式中，这时要求函数带回一个确定的返回值以参加表达式计算。例如：　v = volume(3, 4, 5);

```
c = 2 * max(a, b);
```


(3)作为函数的实参。例如：

```
m = max(a, max(b, c));
```

其中max(b, c)是第一次函数调用，它的返回值作为max( )第二次调用的实参。m的值取a, b, c中最大值。再如：　printf("%d", max(a, b));

当一个函数调用另一个函数时(前者称为调用函数，后者称为被调用函数)，必须进行参数间的传递，即调用函数怎样将参数传递给被调用函数，随后被调用函数根据调用函数传递过来的参数进行计算和处理，并将计算的结果返回给调用函数。

说明：(1)形参为自动变量，在未调用该函数时，它们并没有分配内存空间。只有在调用它时形参才被分配内存空间。并将对应的实参值传递给形参。调用结束后，形参所占的内存空间即被释放。

(2)函数被调用时，实参是用来对形参初始化的，因此实参的类型必须与函数定义时形参的类型一一对应，即实参与形参的个数应相等，类型对应一致，按其位置顺序一一对应传递数据。


```
# include <stdio.h>
plus(int x, int y);
void main( )
{
    printf("\n SUM = %d", plus(3, 4));
}
plus(int x, int y)
{  return  x + y;  }
```

int x　x = 3

int y　y = 4

(3) 当需要返回一个函数值时，在函数体内使用return语句。 return语句有两种形式：

return 表达式 ; ①

return; ②

语句①是求得表达式的值，如果表达式的类型与函数的类型不相同时，则将表达式的类型自动地转换成函数的类型，作为函数值返回给调用侧。该表达式可以是任意表达式。当无返回值时，采用第②种形式。此时的return语句若放在函数体的末端则可缺省。与其他语言一样，一个函数体中可以有多个return语句，它们大多出现在选择性的流程控制语句，象if，switch语句中。

2. 函数的传值调用：又称“函数调用的值传递方式”。使用值传递方式调用时，实参可以是常量、已经赋值的变量或表达式值、甚至是另一个函数，只要它们有一个确定的值，被调用函数的形参用变量。调用时系统先计算实参的值，再将实参的值按位置对应地赋给形参，即对形参进行初始化。

因此，传值调用的实现机制是系统将实参拷贝一个副本给形参。在被调用函数体内，形参的改变只影响副本中的形参值，而不影响调用函数中的实参值。所以说，值传递方式调用的特点是“单方向”，形参值的改变不影响实参。


```
#include<iostream>
using namespace std;
void swap1(int x, int y)
{ int temp;
  temp = x;
  x = y;
  y = temp;
  cout << "In swap1( ) : x = " << x
       << " , y = " << y << endl;
}
```

副 本

int x x = 5 ⇒ 9

int y y = 9 ⇒ 5


```
void main( )
{ int a(5), b(9);
  cout << "Before called swap1( ):a = "
       << a << " , b = " << b << endl;
  swap1(a, b);
  cout << "After called swap1( ):a = "
       << a << " , b = " << b << endl;
}
```

int a a = 5

int b b = 9

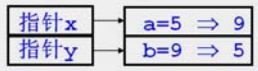
执行该程序，输出如下结果：

```
Before called swap1( ): a = 5 , b = 9
In swap1( ) : x = 9 , y = 5
After called swap1( ): a = 5 , b = 9
```

从程序的输出结果来看，在调用swap1( )函数时，系统将实参拷贝一个副本给形参，实参a和 b的值确实传递给形参x和y。在swap1( )函数体内，形参x和y做了一次交换。但在main( )中，实参a和 b的值还是原来的值，由此可见在swap1( )函数的形参值变化，对main( )中的实参a和 b没有影响，因此其数据的传递是单方向的。

3.函数的传址调用：又称“函数调用的地址传递方式”。如果想让形参的改变影响实参，即函数返回时需要获得几个结果值，应采用地址传递方式，即调用函数的实参用变量的地址值(而不是变量本身值)，被调用函数的形参用指针，调用时系统将实参的地址值赋给对应的形参指针，使形参指针指向了实参变量。所以，在被调用函数体内，凡是对形参指针所指内容的操作，都会使实参变量发生相同的变化。它的实现机制是让形参指针直接指向实参。其特点是可以通过改变形参所指向的变量值来影响实参。这是函数间传递信息的一种手段。


```
void swap2(int * x, int * y)
{ int temp;
  temp = * x;
  * x = * y;
  * y = temp;
  cout << "In swap2( ) : x = " << * x
       << " , y = " << * y <<endl;
}
```




```
void main( )
{ int a(5), b(9);

  cout << "Before called swap2( ):a = "
       << a << " , b = " << b << endl;
  swap2(&a, &b);
  cout << "After called swap2( ): a = "
       << a << " , b = " << b << endl;
}
```

执行该程序，输出如下结果：

```
Before called swap2( ): a = 5 , b = 9
In swap2( ) : x = 9 , y = 5
After called swap2( ): a = 9 , b = 5
```

从程序的输出结果来看，在调用swap2( )函数时，系统将实参的地址值赋给对应的形参指针，相当于执行了“int * x = &a;”、“int * y = &b;”，使形参指针指向实参变量。所以，在swap2( )函数体内，对形参指针所指内容(*x)和(*y)做了一次交换操作，会使实参变量a和b也做了交换，即在main( )函数中的a和b值也进行了交换。因此，地址传递方式可以通过改变形参所指向的变量值来影响实参值，从而达到函数之间的信息传递。

函数的引用调用：如前所述，引用是给一个已存在的变量启用替换名，对引用的操作等价于对被引用变量的操作。引用主要是用来作函数的形参和函数的返回值。使用引用作为函数形参时，调用函数的实参要用变量名。函数调用时，将实参变量名赋给形参的引用，即对形参引用进行了初始化，相当于在被调用函数中使用了实参的别名。于是在被调用函数体内，对引用的改变，实质上就是直接通过引用来改变实参的变量值。由于引用与被引用变量是通过地址相联系，所以引用调用实质上仍然是地址传递方式。所不同的是，实参变量对形参引用的初始化操作是被当作值本身来传递，不必写取地址运算和取内容运算，因此更直接、更方便。所以，在C++中经常使用引用作函数的参数来实现在被调用函数中改变调用函数的实参值，实际上让函数返回了多个结果值。


```
#include <iostream.h>
void swap3(int & x, int & y)
{ int temp;
  temp = x;   &a a = 5 ⇒ 9 &b b = 9 ⇒ 5
  x = y;      &x x = 5 ⇒ 9 &y b = 9 ⇒ 5
  y = temp;
  cout << "In swap3( ) : x = " << x
       << " , y = " << y <<endl;
}
void main( )
{ int a(5), b(9);
  cout << "Before called swap3( ):a = "
       << a << " , b = " << b << endl;
  swap3(a, b);
  cout << "After called swap3( ):a = "
       << a << " , b = " << b << endl;
}
```


执行该程序，输出如下结果：

```
Before called swap3( ): a = 5 , b = 9
In swap3( ) : x = 9 , y = 5
After called swap3( ): a = 9 , b = 5
```

由此可见，

(1)在引用调用中，实参用变量名，形参用引用。

(2)调用时系统将实参的变量名赋给对应的形参引用，相当于执行了引用的初始化操作(“int & x = a;”、“int & y = b;”)。所以，在swap3( )函数体内，对形参引用x和y所做的交换操作，就等价于对实参变量a和b的操作，即在main( )函数中的a和b值也进行了交换。

(3)凡是用指针作形参进行地址传递的函数，都可以将形参指针改为“引用”，调用函数的实参可直接写变量名；被调用函数体内的形参可象变量一样使用，而不必使用取地址运算和取内容运算。

1.引用作为函数的返回值：返回引用的函数是非常有用的，该函数的调用表达式可以写在赋值运算符的左边，称为“左值表达式”，可直接对函数进行赋值、增量、减量等运算。因此，在ANSI C++中，函数可返回到一个引用，返回的引用作为左值直接增量。例如：

```
char elem1(char * s , int n)
{ return s[n];  }
```

该函数执行一次带下标的操作，它读取一个字符数组，返回值为第n个字符，即读取第n个字符。这类函数是不能写在赋值运算符的左边成为“左值表达式”。也就是说，elem1( )函数只能读取第n个字符，而不能改写第n个字符。即：

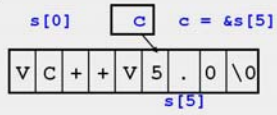
```
char *str = “VC++V5.0”;
char ch = elem1(str , 5);
elem1(str , 5) = ‘6’;    //出错。
```

如果将elem1( )函数改成elem2( )，利用它的返回值来改变数组某元素的值，即改写第n个字符，可使该函数返回一个指向字符的指针，即定义成指针函数，以解决这一问题。


```
#include <stdio.h>
char * elem2(char *s , int n)
{  return &s[n];  }

void main( )
{  char *c;
   char str[] = "VC++V5.0";

   printf("%s\n", str);
   c = elem2(str, 5);
   *c = '6';
   printf("%s\n", str);
}
```



合并成一条语句

我们甚至可以把函数调用表达式elem2(str, 5)直接写在赋值运算符的左边，作为“左值表达式”，这样就不必通过字符指针c ，可把它从程序中删掉，将两条赋值语句合并成一条语句：

```
*elem2(str, 5) = '6';
```

然而，在ANSI C++中还有更好的方法，该函数不用传回一个指针，而返回一个引用，它的调用表达式也可写在赋值运算符的左边。即：


```
#include <stdio.h
char & elem2(char * s , int n)
{ return s[n]; }

void main( )
{
   char str[ ] = "VC++V5.0";
   printf("%s\n", str);
   elem2(str , 5) = '6';
   printf("%s\n", str);
}
```

(1)返回引用的函数其定义格式为：

```
存储类   类型 & 函数名（形参表）
{            。。。
        return    表达式;
}
```

与一般函数的定义格式类同，只是其中，<类型 &>指明函数的返回值为引用，在函数体内与之对应有一个“return 表达式;”语句。当然，正如下面例程那样，也可以有多个这样的return语句，它们大多出现在选择性的流程控制语句，象if，switch语句中。

(2)这种返回引用的函数，只能用全局变量或静态变量作返回值，而不能用该函数的自动变量，因为自动变量在该函数体外自动消失。也就是说，return语句的表达式只能写全局变量名或静态变量名，而不能写自动变量名。


```
#include <iostream>
using namespace std;
int array[6] = {66, 78, 86, 92, 88, 96};
int typeA = 0 , typeB = 0;
int & level(int k)
{   if(array[k] >= 80)
        return typeA;  //A类学生数
else
    return typeB;  //B类学生数
}
void main( )
{   for(int i = 0; i < 6; i++)
          level(i) ++;
    cout << "number of typeA is "
         << typeA << endl;
    cout << "number of typeB is "
         << typeB << endl;
}
```


该程序是统计学生中，A类学生和B类学生各占多少。A类学生的标准是成绩在80分以上，其余都是B类学生。六位学生的成绩存放在array[ ]数组中。其输出结果为：

```
number of typeA is 4
number of typeB is 2
```

由于level( )函数返回的是引用，所以可以作为“左值表达式”直接进行增量操作。其返回语句中的typeA或typeB都是全局变量。（方法1）

(3)为了避免使用全局变量，可采用方法2。即需要给level( )函数传递两个引用参数。为此，给它增设两个形参引用int &tA 、int &tB，分别作为两个实参typeA和typeB的别名，将它们传递给被调用函数level( )。因此这两个形参必须定义成引用，而不能定义为int tA、int tB，即不能采用值传递。


```
#include <iostream>
int array[6] = {66, 78, 86, 92, 88, 96 };
int & level(int k , int &tA , int &tB)
{   if(array[k] >= 80)
        return tA; //等价于return typeA;
else                  int &tA = typeA;
        return tB; //等价于return typeB;
}                     int &tB = typeB;
using namespace std;
void main( )
{
    int typeA = 0 , typeB = 0;
    for(int i = 0; i < 6; i++)
        level(i , typeA , typeB) ++;
cout << "number of typeA is "
     << typeA << endl;
cout << "number of typeB is "
     << typeB << endl;          }
```

在调用level( )函数时，typeA、typeB分别传递给形参引用tA、tB。即相当于执行了“int & tA = typeA;”“int & tB = typeB;”。那末在level( )函数体内，对tA 、tB的操作就等价于对typeA、typeB的操作。因此对level( )进行增量操作，就等价于对typeA或typeB的增量操作。

（三）设置函数参数的默认值(缺省值Default)：

在ANSI c++中，允许在函数原型的说明语句中（最好是在说明语句中）或函数定义时，给一个或多个参数指定默认值。以后在调用该函数时，若省略其中某一参数时，系统以默认值作为该参数的实参值。


```
#include <iostream>
using namespace std;
int m(8);     //或 int m = 8;
int add_int(int x, int y = 7, int z = m);
void main( )
{  int a(5), b(15), c(20);
   int s = add_int(a, b);
   cout << "sum = " << s << endl;
}
int add_int(int x, int y, int z)
{  return x + y + z;  }
```


1.函数参数的默认值是在形参表内用常量表达式、已经赋值的变量或任意表达式进行指定。若作如下修改，参数z采用表达式进行指定默认值。

```
int m(8);
int & n = m;
int add_int(int x,int y=7,int z=2 * n);
```

2.在一个函数中，可指定多个参数的默认值，甚至指定全部参数的默认值。但必须是从右至左连续指定，即指定和不指定不能交叉。 例如：

```
int add_int(int x=6,int y=7,int z=m);
```

以下非法：　　左 ⇐ 右

```
int add_int(int x,int y=7,int z);
int add_int(int x = 6,int y,int z);
```


3.当函数调用时，实参数目少于形参时，编译器按从左至右的顺序用默认值来补足所缺少的实参。

```
#include <iostream>
using namespace std;
void fun(int a=1, int b=3, int c=5)
{     cout << "a = " << a << " , b = " << b
         << " , c = " << c << endl;
}
void main( )
{
      fun( );          // a=1,  b=3,  c=5
      fun(7);          // a=7,  b=3,  c=5
      fun(7,9);        // a=7, b=9,  c=5
      fun(7,9,11);     // a=7, b=9, c=11
      cout << "OK!";
}
```

调用时，也必须是从右至左连续采用默认值，“采用”和“不采用”不能交叉。若第1个参数采用默认值，则其后的参数必须都采用默认值，如“fun(7,9,11);”。也可以第1个参数不采用默认值，从第2个参数开始采用默认值，如“fun(7);”。但不能写：“fun(,9);”//出错，形参a采用默认值，而b不采用。

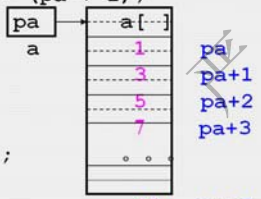
（四）使用数组名作函数参数：数组名是地址常量，既可以作为函数的形参，也可以作为函数的实参。指针变量也可作为函数的形参和实参，只不过作为实参时它们应具有确定的地址值。在函数调用时，将实参的地址值赋给(传递给)形参。它们有如下四种组合形式：

形参	实参
(1) 指针	数组名
(2) 数组名	数组名
(3) 数组名	指针 = 数组名(调用侧写地址赋值语句)
(4) 指针	指针 = 数组名(调用侧写地址赋值语句)

以上四种组合实质上只有两种。因为作为实参的指针必须具有确定的地址值，即已经用地址赋值运算定向指向了某数组，此时的指针实质上是数组名，因此(3)就转化为(2)，(4) 就转化为(1)。

(1)当函数的实参为数组名，而形参为指针时，传递给函数形参的初值是数组首地址，而作为形参的指针接收该地址值。


```
#include <iostream>
using namespace std;
int a[8] = {1 ,3 ,5 ,7 ,9 ,11 ,13};
void fun(int * pa , int n)
{  for(int i = 0 ; i < n - 1 ; i++)
       *(pa + 7) += * (pa + i);
}
void main( )
{  int m = 8;
   fun(a , m);
   cout << "a[7] = "
     << a[7] << endl;
}
```

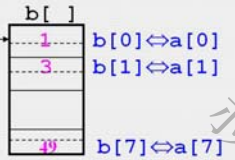


在执行“fun(a , m);”函数调用语句时，实参赋给形参的传递过程中，相当于执行了“int * pa = a; int n = m; ”。由于pa指向了数组a，所以a[i]、pa[i]、*(a + i)、*(pa + i)等是四种等效不等价的，访问第i个元素的表达式。

(2)当函数的形参和实参都是数组名时，在函数调用前形参数组(b[ ])并没有分配内存空间。在调用函数(fun( ))时，实际上是把实参数组的首地址传递给形参数组，这样一来形参数组和实参数组共享同一段内存空间，即形参数组并没有开劈新的存储单元，而是以实参数组(a[ ])的首地址作为形参数组(b[ ])的首地址。

```
#include <iostream>
using namespace std;
int a[8] = {1 ,3 ,5 ,7 ,9 ,11 ,13};
void fun(int b[ ] , int n)
{
     for(int i = 0 ; i < n - 1 ; i++)
         b[7] += b[i];
}

void main( )
{  int m = 8;
   fun(a , m);
   cout << "a[7] = "
        << a[7] << endl;
}
```



这样实参数组的第0号元素(a[0])和形参数组的第0号元素(b[0])共占用同一个内存单元。同理a[1]⇔ b[1], a[2]⇔ b[2],……。因此在函数调用过程中，执行函数体内的语句时，使形参数组b[ ]的元素值发生变化也就使实参数组a[ ]的元素值发生了相同变化。

★数组名（一维）作为形参时，其方括号内的元素个数可以缺省，即不指明元素个数，由实参数组才最后确定元素个数。这样处理可以使函数通用性强，不因元素个数的限制，而减少适用范围。(二维数组的列得标明)

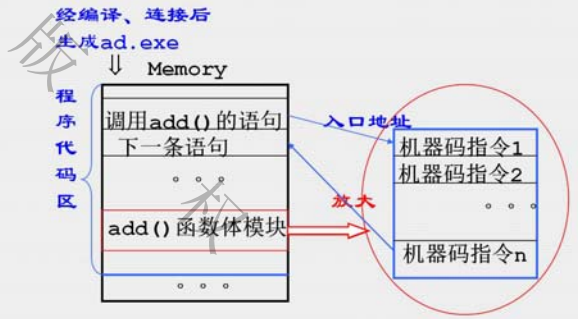
✱由于形参数组和实参数组共享同一段内存空间，因此在函数调用过程中，函数体对形参数组元素的运算和操作，也会使实参数组发生相同的变化。

（五）内联函数(Inline Function)

1.函数名为该函数的入口地址：如前所述，数组名表示该数组的首地址，可把数组名赋给一个指向相同数据类型的指针变量，令指针指向该数组。

```
#include <iostream>
using namespace std;
int add(int a, int b)
{   return a + b; }

void main( )
{   int x = add(2, 4);
    cout << "x = " << x << endl;  }
```

函数名也具有数组名的这种特性。函数名表示该函数目标代码(即机器码，它是源程序经编译、连接后所生成的filename.exe文件，称为目标代码文件。在执行该程序时，由操作系统自动地将filename.exe文件装入内存)存储首地址。即函数执行的入口地址。例如，ad.cpp源程序经编译、连接后所生成的ad.exe文件。在执行该程序时，操作系统自动地将ad.exe文件调入内存，分成程序代码区、数据区和堆栈区装配好。在程序代码区中必定有add( )函数的目标代码模块。它是由编译、连接所生成的一系列机器码指令1、机器码指令2、机器码指令3、。。。。、机器码指令n等组成。函数名add是表示该函数机器码指令1的存储地址，即函数目标代码模块的存储首地址。在 main( )函数中每调用一次add( )函数时，实质上是控制程序的执行转移到由函数名所给出的该函数目标代码首地址。因此也称为控制程序的执行转移到函数的入口地址。

2.内联函数引入的原因

函数调用时必然导致一些额外的开销而影响程序执行速度。例如，要保护现场将参数压入堆栈，并查到被调用函数的入口地址，同时把返回地址压入堆栈。调用结束时，要恢复现场并从堆栈中弹出返回值和返回地址实现返回，显然函数调用要有一定的时间和空间的开销从而影响执行效率。特别对于一些函数体代码不大，但又频繁被调用的函数，C++的设计者为编程者提供”内联函数”，以解决这类函数的执行效率问题。

3.内联函数的定义方法

当定义函数时，在函数头的前面加上关键字”inline”，则指明该函数为内联函数。编译时该函数不被编译为单独一段可调用代码，而是将生成的函数体代码直接插入到对该函数的每个调用处，即以空间换时间。

例如：

```
#include <iostream.h>

inline int add(int a, int b)
{ return a + b; }

void main( )
{ int x = add(2, 4);
  cout << "x = " << x << endl;
}
```

函数体代码块

调用处

4.使用内联函数应注意的事项

(1)在内联函数内不允许用循环语句和开关语句。(编译系统使其无效)

(2) 内联函数必须在使用它之前定义好。由于内联函数必须预先编译生成了函数体代码块，才能插入到程序的调用处。

(3)内联函数将使程序代码增大。因此内联函数只适合于有1~5行的小函数，对一个含有许多语句的大函数，函数的调用和返回的开销相对来说是微不足道的，所以也就没有必要用内联函数实现。

(4)递归函数需自己调用自己，不能被用来做内联函数。

（六）函数重载：

所谓“函数重载”是指在同一作用域内多个函数体可以使用相同的函数名，这类函数称为“重载函数”。

```
void print(char * str)
{
    cout << "print(char *) called !\n
            str = \"";
    cout << str << "\"\n";
}
void print(int value)
{
   cout << "print(int) called !\n
           value = ";
   cout << value << endl;
}
```



```
void print(long value)
{    cout << "print(long) called !\n
                value = ";
     cout << value << endl;
}

void print(double value)
{     cout << "print(double) called !\n
                value = ";
    cout << value << endl;
}
```



```
void main( )
{ int a(6);
    float b(8.8f);
    cout << "(1)";
print("How are you ?");
//匹配 void print(char * str)
    cout << "(2)";
print(double(a));
//匹配 void print(double value)
    cout << "(3)";
print(long(a));
//匹配 void print(long value)
    cout << "(4)";
print(a);
//匹配 void print(int value)
    cout << "(5)";
print(1.6);
//匹配 void print(double value)
```



```
    cout << "(6)";
print('a');
//匹配 void print(int value)
    cout << "(7)";
print(3.14159);
//匹配 void print(double value)
    cout << "(8)";
print(b);
//匹配 void print(double value)
}
```

通常对于完成不同任务的函数应该选择不同的函数名，但对于不同数据类型的对象完成相同任务的那些函数，若启用同样的函数名会给调用这些函数带来很大方便。如例程中把分别输出字符串、整数和双精度浮点数的三个函数都取名为print( )，编译系统通过比较实参和形参的个数和类型，来决定调用正确的重载函数完成相应的操作。编程者还可以把print( ) 函数扩充到输出向量和矩阵。

1.重载函数的使用说明：

(1)重载函数间首先用函数参数的个数不同加以区分，然后再用参数类型加以区分，其中至少要有一个参数类型不同，并且应确保参数类型的确实不同。例如，用typedef语句定义的类型只是一个已有类型的别名，并没有创建新的类型，所以如下的程序段是错误的：typedef int INT;

```
void func(int x);
void func(INT x);
```

(2)函数返回类型的不同并不能区分重载函数，如下声明是非法的：

```
int process(int i);
void process(int i);
```

(3)普通重载函数(非成员函数)的作用域是文件。

2.重载函数的匹配规则：当调用一个重载函数如print( )时，编译系统必须搞清函数名究竟是指哪一个函数，这是通过把实参个数和类型与所有被调用的print( )函数的形参个数和类型一一比较来判定的，其查找顺序为：

(1)首先寻找与实参的个数、类型完全相同的函数，如果找到了就用这个函数。

(2)寻找一个严格匹配，如果找到了就用这个函数。
对于int型的形参，实参是 0、char型short int型都是严格匹配。对于double型的形参，实参是float型也是严格匹配。例如，例程中的print('a');语句是调用void print(int value)函数。

(3)按照C++内部类型转换规则(参见§2.6运算符之1. 表达式中的类型转换)，寻找一个匹配的函数，如果找到了就用这个函数。

由于C++允许int到 long、int到double的转换，因此若实参是int型，而重载函数一个是long型形参，另一个是double型时，应该用显式转换指明。如：

```
void print(long);
void print(double);
void fun(int a)
{
   print(a);    ↖ print(long(a));
}               ➢ print(double(a));
```