Chapter 3 Functions

C ++ inherits all the C syntax, including the definition and usage of functions. In process-oriented programming (also known as structured programming), function is the basic unit of module division, and an abstract of problem-solving process. Function is also important in object-oriented programming, in which it is an abstract of functionalities.

To develop or debug a complex system, engineers will usually divide it into several sub-systems, and then develop or debug based on these sub-systems. Sub-programs in high-level program languages are used to realize this kind of module division. In C and C++ sub-programs are embodied as functions. We usually abstract independent and frequently used functionality modules into **functions**. Once a function is written, we can reuse it only according to its functions and usage, without the need to know its specific implementation. In this way code reuse is achieved, development efficiency and program reliability are improved, and collaboration, modification and maintenance can be realized much easily.

3.1 Definition and Use of Function

A C++ program consists of one main function and several sub-functions. A program is executed starting from its main function. The main function may call sub-functions, and sub-functions may in turn call other sub-functions.

The function which calls other functions is named "**calling function**", and the function called by others is named "**called function**". A function may call another function and also be called by some other. Thus it can be a calling function in some occasion and a called function in another.

3.1.1 Definition of Function

1. Syntax Form of Function Definition

```
Type identifier  Function name (Formal Parameter list)
{
   Statements
}
```

2. Type of Function and Return Value

The type identifier defines the type of the function, also the type of the return value of the function. The return value of the function is the result that the function returned to its calling function, which is given by *return* statement, such as "*return* 0".

The function without a return value has a type identifier of *void*, and there need not be *return* statement in the function.

3. Formal Parameters

Here is the form of formal parameter list:

```
type 1 name 1, type 2 name2, …, type n namen
```

*type*1, *type*2,…, *type n* are type identifiers which represent the types of formal parameters, and *name*1, *name*2,…, *name n* are the names of formal parameters. Formal parameters are used to realize the connection between the called function and the calling function. We often let the data that needs to process, factors that affect the function's behavior, or the processing results of the function to be the function's formal parameters. Functions without formal parameters should have *void* on the position of the parameter list.

Main function can also have formal parameters and return value. The formal parameters of the *main* function are also called command line parameters, which are initialized by the operating system when starting the program. The return value of *main* function is returned to the operating system. The types and the number of formal parameters of the *main* function have a special format. Refer to the experiment instructions in <*Student's Book*> to write programs with command line parameters.

A function is only a piece of text before it has been called, and its formal parameters at the time are just symbols, indicating what type of data should appear on the position of the formal parameter. A function starts execution when it is called, and at that time does the calling function assigns the actual parameters to the formal parameters. This is similar to the definition of function in mathematics:

$$f(x) = x^2 + x + 1$$

The function f will not be calculated until its argument has been assigned a value.

3.1.2 Function Calls

1. Form of Function Calls

Before calling a function we firstly need to declare the **function prototype**. The declaration can be in the calling function or before all the functions, with the following form:

```
Type identifier Function name(Formal parameter list with type declarations);
```

If a function prototype is declared before all the functions, it is effective in the whole program file. That is, we can call the corresponding function anywhere in the file according to the prototype. If a function prototype is declared inside a calling function, it is only effective in this calling function.

After declaring the function prototype, we can make a function call by the following form:

```
Function name(Actual parameter list)
```

The actual parameter list should provide parameters that accurately match the formal parameters in number and in types. A function call can be used as a statement, where the return value of the function is not needed; a function call can also appear in an expression, where an explicit return value of the function is needed.

Similar to the declaration and definition of variable, declaring a function only tells the compiler its relevant information (about function name, parameters, return type, etc) without generating any codes, while defining of a function mainly gives the function code besides its relevant information.

Example 3-1 Writing a function to calculate the *n*th power of *x*.

```
//3_1.cpp
#include<iostream>
using namespace std;
double power (double x, int n);
int main()
{
    cout<<  "5 to the power 2 is "<<power(5,2)<<endl;
    //Function call is counted as an expression in the output statement.
    return 0;
}
double power (double x, int n)
{
    double val=1.0;
    while (n--)
    val *=x;
    return(val);
}
```

Running result:

```
5 to the power 2 is 25
```

Example 3-2 Enter an 8-bit binary number, convert it to its decimal form and then output the result.

Analysis: To convert a binary number to its decimal form, we need to multiply every bit of the binary number with the corresponding weight, and then add them up. For example: $00001101_2=0(2^7)+0(2^6)+0(2^5)+0(2^4)+1(2^3)+1(2^2)+0(2^1)+1(2^0)=13_{10}$. So when the input is 1101, the output should be 13.

Here we make a function call on the function *power* in Example 3-1 to calculate 2^n.

Source code:

```
//3_2.cpp
#include<iostream>
```

```
using namespace std;
double power (double x, int n);

int main()
{
    int  i;
    int  value=0;
    char  ch;

    cout<<"Enter an 8 bit binary number  ";
    for (i=7; i >=0; i--)
    {
        cin >>ch;
        if (ch=='1')
            value+=int(power(2,i));
    }
    cout<<"Decimal value is  "<<value<<endl;
    return 0;
}

double power (double x, int n)
{
    double val=1.0;

    while (n--)
      val *=x;
    return(val);
}
```

Running result:

```
Enter an 8 bit binary number  01101001
Decimal value is  105
```

Example 3-3 Write a program to compute the value of π using the following formula:

$$\pi = 16\arctan\left(\frac{1}{5}\right) - 4\arctan\left(\frac{1}{239}\right)$$

Use the following series to calculate the arctangent of a number:

Continue accumulating until the absolute value of one item in the series is less than 10^{-15}. The type of π and x are both double.

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

Source code:

```
//3_3.cpp
#include<iostream>
```

```
using namespace std;
int main()
{
  double a, b;
  double arctan(double x)  ;
  a=16.0*arctan(1/5.0)  ;
  b=4.0*arctan(1/239.0)  ;
  //Note: Since the division of integers is to be rounded off, the value of 1/5 and
  //1/239 are both 0.
  cout<<"PI="<<a-b<<endl;
  return 0;
}
double arctan(double x)
{
  int i;
  double r, e, f, sqr;
  sqr=x*x;
  r=0;
  e=x;
  i=1;
  while(e/i>1e-15)
  {
    f=e/i;
    r=(i%4==1)?r+f : r-f   ;
    e=e*sqr;
    i+=2;
  }
  return  r ;
}
```

Running result:

```
PI=3.14159
```

Example 3-4 Find the number m between 11 and 999 that m, m^2 and m^3 are all palindromes, and then output m.

Palindromes are numbers which have symmetrical number digits. For example: 121, 676 and 94249 are all palindromes. One instance that satisfies this subject's requirement is: $m=11$, $m^2=121$, $m^3=1331$.

Analysis: To check whether a number is a palindrome or not, we can get every digit of the number by continuously dividing it by 10 and get the remaining. After getting all the digits, we reverse the digit order to get a new number, and compare the new number with the original one. The original number is a palindrome if and only if it is the same as the new number.

Source code:

```
//3_4.cpp
#include<iostream>
using namespace std;
int main()
{
  bool symm(long n);
  long m;
  for(m=11; m<1000; m++)
      if (symm(m)&&symm(m*m)&&symm(m*m*m))
          cout<<"m="<<m<<"  m*m="<<m*m<<"  m*m*m="<<m*m*m<<endl;
  return 0;
}

bool symm(long n)
{
  long i, m;
  i=n ;  m=0 ;
  while(i)
  {
   m=m*10+i%10;
   i=i/10   ;
  }
  return (m==n);
}
```

Running result:

```
m=11  m*m=121  m*m*m=1331
m=101  m*m=10201  m*m*m=1030301
m=111  m*m=12321  m*m*m=1367631
```

Example 3-5 Compute the value of the following formula and output the result:

$$k = \begin{cases} \sqrt{\sin^2(r) + \sin^2(s)} & \text{when } r^2 \leqslant s^2 \\ \frac{1}{2}\sin(r \times s) & \text{when } r^2 > s^2 \end{cases}$$

Here the value of r and s is input from the keyboard. The approximate value of $\sin x$ is calculated using following formula:

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{x^{2n-1}}{(2n-1)!}$$

The precision of the calculation is 10^{-6}. Stop accumulating when the absolute value of one item is less than the precision, and the accumulated value is the approximate value of $\sin x$.

Source code:

```
//3_5.cpp
#include<iostream>
#include<cmath>        //Header file cmath has declaration of mathematic functions
```

```
using namespace std;
int main()
{
  double k,r,s;
  double tsin(double x);
  cout<<"r=";
  cin>>r;
  cout<<"s=";
  cin>>s;
  if (r*r<=s*s)
    k=sqrt(tsin(r)*tsin(r)+tsin(s)*tsin(s))  ;
  else
    k=tsin(r*s)/2;
  cout<<k<<endl;
  return 0;
}

double tsin(double x)
{
  double p=0.000001,g=0,t=x;
  int n=1;
  do {
     g=g+t;
     n++;
     t=-t*x*x/(2*n-1)/(2*n-2);
  }while(fabs(t)>=p);
  return g;
}
```

Running result:

```
r=5
s=8
1.37781
```

Example 3-6 Game of casting dice.

Game rules: Dice has 6 faces—counting by the points they are 1, 2, 3, 4, 5 and 6. The player inputs an unsigned integer which is used as the seed to generate a random number at the beginning of the program.

In each turn the dice is casted twice, and we can get the total point. In the first turn, if the total point is 7 or 11, the player wins and the game is over; if the total point is 2, 3 or 12, the player loses and the game is over; otherwise the player records the total point as his point. In the following turns, if the total point is equal to the player's point, the player wins and the game is over; if the total point is 7, the player loses and the game is over; otherwise, it goes on to the next round.

The function *rolldice* is used to simulate rolling the dice, get the total point and output it.

Note: The system function *int rand*(*void*) is to generate a pseudo random number. The pseudo random number is not really random. When we call this function continuously in a program, in the hope that it will generate a sequence of random numbers, we may discover that it will generate a same sequence each time we run the program. This sequence is called pseudo random number sequence. It is because that *rand* needs an initial number called "seed", and different seeds will generate different sequences. Thus, if we give the program a different seed in each run, continuously calling *rand* will generate a different random number sequence. If the seed is not set, *rand* will use the default value 1 as the seed. Note that the method to set seed is somewhat special: it is not through the parameters of *rand*, but by calling another function *void srand*(*unsigned int seed*) to set the seed before calling *rand*, and the parameter *seed* in function *srand* is the seed of *rand*.

Source code:

```
//3_6.cpp
#include<iostream>
#include<cstdlib>
using namespace std;
int rolldice(void);
int main()
{
  int gamestatus,sum,mypoint;
  unsigned seed;
  cout<<"Please enter an unsigned integer:";
  cin>>seed;          //Input the seed for the random number
  srand(seed);        //Pass the seed to rand()
  sum=rolldice();     //The first round, roll the dice and get the total point
  switch(sum)
  {
    case 7:           //Win if the total point is 7 or 11, status=1
    case 11:
           gamestatus=1;
            break;
    case 2:           //Lose if the total point is 2, 3 or 12, status=2
    case 3:
    case 12:
           gamestatus=2;
           break;
    default:          //Otherwise, continue the game, record the player's point,
                      //and set status=0
           gamestatus=0;
           mypoint=sum  ;
           cout<<"point is "<<mypoint<<endl;
```

```
      break;
    }
    while (gamestatus==0)     //Go to the next round if status=0
    {
      sum=rolldice();
      if(sum==mypoint)        //Win if the total point is equal to the player's point,
                              //set status=1
        gamestatus=1  ;
      else
        if (  sum==7  )       //Lose if the total point is 7, set status=2
          gamestatus=2;
    }
//When the status is not 0, the loop above ends, and the following code outputs the
//result
  if(  gamestatus==1  )
    cout<<"player wins\n";
  else
    cout<<"player loses\n";
  return 0;
}
int rolldice(void)
{ //Roll the dice, get the total point, and output it
  int die1,die2,worksum;
  die1=1+rand()%6;
  die2=1+rand()%6;
  worksum=die1+die2;
  cout<<"player rolled "<<die1<<'+'<<die2<<'='<<worksum<<endl;
  return worksum;
}
```

Running result 1:

```
Please enter an unsigned integer:8
player rolled 5+1=6
point is 6
player rolled 6+6=12
player rolled 6+4=10
player rolled 6+6=12
player rolled 6+6=12
player rolled 3+2=5
player rolled 2+2=4
player rolled 3+4=7
player loses
```

Running result 2:

```
Please enter an unsigned integer:23
player rolled 6+3=9
point is 9
player rolled 5+4=9
player wins
```

2. Procedure of Calling a Function

The compiler compiles a C++ program and outputs a piece of executable code, which is then stored as a file suffixed with *exe* in the external storage. When the program is started, computer firstly loads the executable code from the external storage into the code area of the memory, and then executes the code from the entry address (the beginning of the *main* function). During the execution, the computer will stop executing the current function when a function call occurs. It will then save the address of the next instruction (return address, used as the entry point of execution when returned from the called function), save the execution scene, jump to the entry address of the called function and execute the called function. When meeting a *return* statement or reaching the end of the called function, the computer will restore the scene previously stored, jump back to the return address and continue the execution. Figure 3-1 shows the procedure of calling a function and returning from the call. The labels in the figure indicate the order of execution.

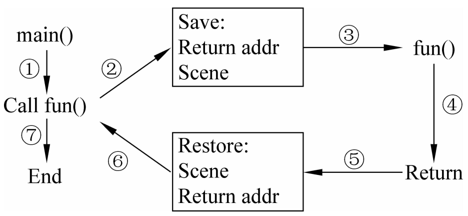


Figure 3-1 Procedure of Function Call and Function Return

3. Nested Function Call

Nested call is allowed in a function. For example, function *A* calls function *B*, then function *B* calls function *C*, and this forms a nested call.

Example 3-7 Input two integers and compute sum of squares of them.

Analysis: Although the problem is easy, we design two functions to show how nested call works: The function named fun1 is used to compute the sum of squares, and the function named fun2 is used to compute the square of an integer. The main function calls fun1, and fun1 calls fun2.

Source code:

```
//3_7.cpp
#include<iostream>
using namespace std;
int main()
{
    int a,b;
    int fun1(int x,int y);
    cin>>a>>b;
    cout<<"Sum of squares of a and b:"<<fun1(a,b)<<endl;
    return 0;
}

int fun1(int x,int y)
{
    int fun2(int m);
    return (fun2(x)+fun2(y));
}

int fun2(int m)
{
    return (m*m);
}
```

Running result:

```
3 4
Sum of squares of a and b:25
```

Figure 3-2 shows the order of function calls in Example 3-7. The labels in the figure indicate the executing order.

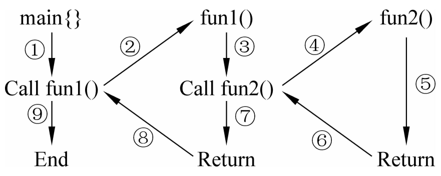


Figure 3-2 The order of function calls in Example 3-7

4. Recursive Call

A function can call itself directly or indirectly. This kind of function call is called recursive call.

Calling oneself directly means that the body of a function contains a function call to itself, for example:

```
void fun1(void)
{
   ⋮
   fun1();                    //A function call in fun1 to itself
   ⋮
}
And here is another example of function indirectly calling itself:
void fun1(void)
{
   …
   fun2();
   …
}
void fun2(void)
{
   …
```

```
    fun1();
    …
}
```

Here $fun1$ calls $fun2$ and $fun2$ in turn calls $fun1$. These two calls constitute an indirect recursive call.

The essence of recursion is that it decomposes the original problem into a new problem which may use the solution of the original problem. Continue the decomposition according to this principle, and each new problem emerged is a simplified subset of the original problem. The ultimately decomposed problem should be one whose solution is already known. The procedure above is a finite recursive call. Only finite recursive call makes sense; infinite recursive call will not get any result and it makes no sense.

The procedure of recursive call consists of 2 parts:

First stage: Recurrence. Decompose the original problem continuously into new subproblems until we reach a known situation, at which point the recurrence ends.

For example, to calculate 5!, we can make a decomposition as follows:

$$5!=5\times4!\rightarrow4!=4\times3!\rightarrow3!=3\times2!\rightarrow2!=2\times1!\rightarrow1!=1\times0!\rightarrow0!=1$$
$$\text{Unknown}\xrightarrow{\qquad\qquad\qquad\qquad\qquad\qquad}\text{Known}$$

Second stage: Regression. Starting from the known situation, use the result of the decomposed problem to solve the previous (more complex) problem. Repeat this process regressively according to the reversed order of the recurrence stage, until we reach the start of the recurrence. The regression then ends and the whole recursion finishes.

The regression procedure of calculating 5! is:

$$5!=5\times4!=120\leftarrow4!=4\times3!=24\leftarrow3!=3\times2!=6\leftarrow2!=2\times1!=2\leftarrow1!=1\times0!=1\leftarrow0!=1$$
$$\text{Unknown}\xleftarrow{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\text{Known}$$

Example 3-8 Compute n!.

Analysis: The formula of calculating n! is:

$$n! = \begin{cases} 1 & (n=0) \\ n(n-1)! & (n>0) \end{cases}$$

This formula is recursive, since it calculates a factorial by using another factorial. Thus the program uses recursive call. The ending condition of the recursion is n=0.

Source code:

```
//3_8.cpp
#include<iostream>
using namespace std;
long fac(int n)
{
    long f;
    if (n<0) cout<<"n<0,data error!"<<endl;
    else if (n==0) f=1;
    else f=fac(n-1) * n;
    return(f);
}

int main()
{
    long fac(int n);
    int n;
    long y;
    cout<<"Enter a positive integer:";
    cin>>n;
    y=fac(n);
    cout<<n<<"!="<<y<<endl;
    return 0;
}
```

Running result:

```
Enter a positive integer:8
8!=40320
```

Example 3-9 Calculate the number of possible combinations (i. e. the combinatorial number) of selecting k person(s) out of n person(s) to form a committee.

Analysis: The combinatorial number of selecting k person(s) out of n person(s)
=The combinatorial number of selecting k person(s) out of $n-1$ person(s)
+The combinatorial number of selecting $k-1$ person(s) out of $n-1$ person(s)

Since the formula is recursive, it is easy to write a recursive function to implement the calculation. The ending condition of the recursion is $n==k||k==0$, at which time the combinatorial number is 1. Then the regression may start.

Source code:

```
//3_9.cpp
#include<iostream>
using namespace std;
int main()
{
   int n,k;
   int comm(int n, int k);
   cin>>n>>k;
   cout<<comm(n,k)<<endl;
   return 0;
}
int comm(int n, int k)
{
   if (  k>n  )
     return 0;
   else if(  n==k||k==0  )
     return 1;
```

```
  else
    return  comm(n-1,k)+comm(n-1,k-1);
}
```

Running result:

```
18 5
8568
```

Example 3-10 Game of Hanoi Tower.

There are 3 pillars A, B and C. *N* plates of different sizes has been piled on pillar A, with larger plates being under smaller plates, as shown in Figure 3-3. The procedure of the game is to move the plates from pillar A to pillar C. The player can use pillar B during the game, while he can only move 1 plate once, and larger plates should always be under smaller plates during the movement.

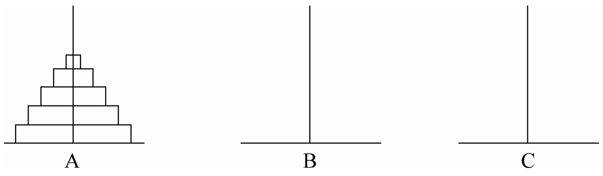


Figure 3-3 Game of Hanoi Tower

Analysis:

We could decompose the procedure of moving *n* plates from pillar A to pillar C to 3 steps:

(1) Move $n-1$ plates from pillar A to pillar B;

(2) Move the last plate on pillar A to pillar C;

(3) Move $n-1$ plates from pillar B to pillar C。

Actually, the 3 steps contain 2 kinds of operations:

(1) Move multiple plates from one pillar to another. It is a recursive operation.

(2) Move one plate from one pillar to another.

The following program uses two functions to implement the two kinds of operations above—function *hanoi* for the first operation and function *move* for the second.

Source code:

```
//3_10.cpp
#include<iostream>
using namespace std;
void move(char getone,char putone)
{
    cout<<getone<<"-->"<<putone<<endl;
}
void hanoi(int n,char one,char two,char three)
{
    void move(char getone,char putone);
    if (n==1) move (one,three);
    else
    {
        hanoi (n-1,one,three,two);
        move(one,three);
        hanoi(n-1,two,one,three);
    }
}

int main()
{
    void hanoi(int n,char one,char two,char three);
    int m;
    cout<<"Enter the number of diskes:";
    cin>>m;
    cout<<"the steps to moving "<<m<<" diskes:"<<endl;
    hanoi(m,'A','B','C');
    return 0;
}
```

Running result:

```
Enter the number of diskes:3
the steps to moving 3 diskes:
A-->C
A-->B
C-->B
A-->C
B-->A
B-->C
A-->C
```

3.1.3 Passing Parameters between Functions

Before a function is called, the formal parameters of this function neither take up any real memory space nor have real values. When a function call is made, computer allocates memory for the formal parameters and assigns the actual parameters to the formal parameters. An actual parameter could be constant, variable or expression, which should match the type of the corresponding formal parameter (the parameter in the same position in its parameter list). Passing parameters between functions is the process of assigning formal parameters according to actual parameters. C++ has two ways to pass parameters: Call-by-Value and Call-by-Reference.

1. Call-by-Value

The procedure of Call-by-Value consists of two steps: allocating memory space for a formal parameter, and using the actual parameter to initialize the formal parameter (assign the actual parameter to the formal parameter). This procedure just passes the value of the actual parameter to the formal parameter. The formal parameter does not have any relation with the actual parameter once it has been initialized, and any change of the formal parameters afterwards can not affect the actual parameter.

Example 3-11 Swap and output two integers.

```
//3_11.cpp
#include<iostream>
using namespace std;
void Swap(int a, int b);

int main()
{
    int x(5), y(10);
    cout<<"x="<<x<<"    y="<<y<<endl;
    Swap(x,y);
    cout<<"x="<<x<<"    y="<<y<<endl;
    return 0;
}

void Swap(int a, int b)
{
    int t;
    t=a;
    a=b;
    b=t;
}
```

Running result:

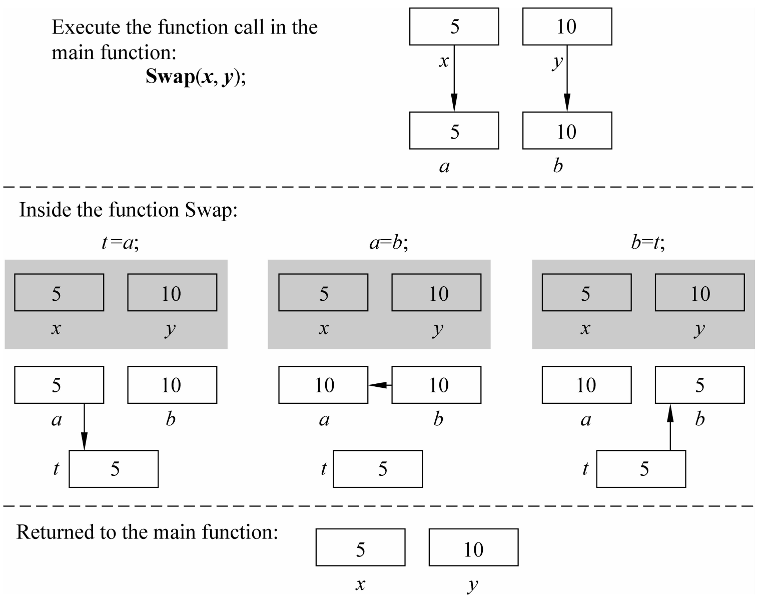
```
x=5      y=10
x=5      y=10
```

Analysis: From the running result we can see that the values of variable x and y have not been swapped. It's because that in the function call above we use Call-by-Value to pass parameters, where only the values of the actual parameters are passed to the formal parameters. Thus the change of the formal parameters afterwards will not affect the actual parameters. Figure 3-4 shows the status of variables when the program is running.

2. Call-by-Reference

We have seen that passing parameters through Call-by-Value is unidirectional. So how can changes made in the called function on formal parameters affect the actual parameters in the calling function? We can use Call-by-Reference to achieve this.

Reference is a special type of variable; it can be viewed as an alias of another variable. Accessing the reference of a variable is the same as accessing the variable itself. Here is an example:



Execute the function call in the main function:
Swap(x, y);
5
10
x
y
5
10
a
b
Inside the function Swap:
t=a;
a=b;
b=t;
5
10
x
y
5
10
a
b
t
5
10
10
a
b
t
5
10
5
a
b
t
5
Returned to the main function:
5
10
x
y

Figure 3-4 The Status of Variables When the Program in Example 3-11 is Running

```
int i,j;
int &ri=i;     //Make an int type reference of ri, initialize it to an alias of i
j=10;
ri=j;          //Same as i=j;
```

The following rules must be followed when using references:

(1) A reference must be initialized to refer to an existing object when it is declared.

(2) Once a reference is initialized, it cannot be changed to refer to other objects.

The rules above indicate that, a reference should be fixed to refer to an object in its whole life, from its definition to its end.

Formal parameters can also be references. When a formal parameter is a reference, the situation is a bit different: the formal parameter of the reference type is not initialized during its type declaration, and it is only when the function is called does the computer allocate memory space for the formal parameter and initialize it to the actual parameter. In this way, the formal parameter of the reference type becomes an alias of the actual parameter, and every operation on the formal parameter would directly affect the actual parameters.

The function call which uses references as formal parameters is called Call-by-Reference.

Example 3-12 Rewrite the program of Example 3-11, using Call-by-Reference to make the two integers swap correctly.

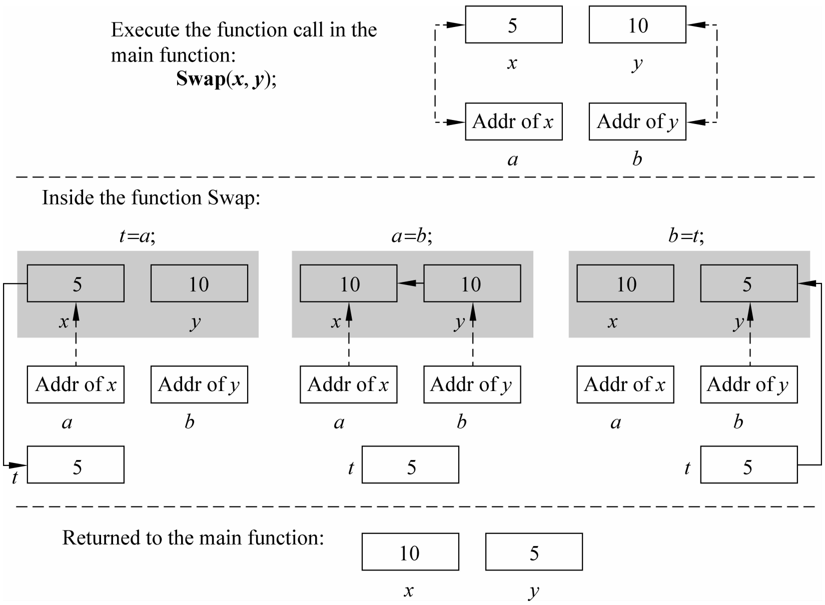
```
//3_12.cpp
#include<iostream>
using namespace std;
void Swap(int& a, int& b);
int main()
{
    int x(5), y(10);
    cout<<"x="<<x<<"    y="<<y<<endl;
    Swap(x,y);
    cout<<"x="<<x<<"    y="<<y<<endl;
    return 0;
}

void Swap(int& a, int& b)
{
    int t;
    t=a;
    a=b;
    b=t;
}
```

Running result:

```
x=5      y=10
x=10      y=5
```

Analysis: We can see from the running result that the swap is successful when the program uses Call-by-Reference to pass parameters. The only difference between Call-by-Value and Call-by-Reference lies in the declaration of the formal parameters, while the function call statements in the calling function are the same. Figure 3-5 shows the status of variables when the program is running.



Execute the function call in the main function:
Swap(x, y);
5
10
x
y
Addr of x
Addr of y
a
b
Inside the function Swap:
t=a;
a=b;
b=t;
5
10
10
10
10
5
x
y
Addr of x
Addr of y
a
b
t
5
Returned to the main function:
10
5
x
y

Figure 3- 5　The Status of Variables when the Program in Example 3-12 is Running

Example 3-13　An example of Call-by-Reference.

```
//3_13.cpp
#include<iostream>
#include<iomanip>
using namespace std;
void fiddle(int in1, int &in2);
int main()
{
    int count=7, index=12;
    cout<<"The values are ";
    cout<<setw(5)<<count;
    cout<<setw(5)<<index<<endl;
    fiddle(count, index);
    cout<<"The values are ";
    cout<<setw(5)<<count;
    cout<<setw(5)<<index<<endl;
    return 0;
}

void fiddle(int in1, int &in2)
{
    in1=in1+100;
    in2=in2+100;
    cout<<"The values are ";
    cout<<setw(5)<<in1;
    cout<<setw(5)<<in2<<endl;
}
```

Running result:

```
The values are    7   12
The values are  107  112
The values are    7  112
```

Analysis: The first parameter *in*1 of function *fiddle* has type *int*, and is assigned the value of the actual parameter *count* when the function is called. The second parameter *in*2 is a reference, and is initialized by the actual parameter *index* to an alias of *index*. Thus, the change on *in*1 in the called function has no effect on the actual parameter *count*, while the change on *in*2 in the called function is in fact the change on the variable *index* in the *main* function. When returned back to the *main* function, the value of *count* has not been changed, while the value of *index* has been changed.

3.2 Inline Functions

At the beginning of this chapter, we mentioned that using functions help developers to reuse codes, improve the development efficiency and the reliability of the program, and facilitate the collaboration and modification of the program. But function calls can also reduce the execution efficiency of programs. When a function call is made, computer needs to save the execution scene and return address before jump to the entry address of the called function and start execution; when returned from the called function, computer needs to restore the scene and return address previously saved before continuing the execution. These procedures take time and memory space. For some simple, small, but frequently used functions, we can use inline functions. **Inline function does not cause control transfer when it is called, but make itself embedded in every place it is called during the compilation.** In this way, the cost of passing parameters and control transfer can be saved.

Inline functions use keyword "inline" in the function definition. The form is like this:

```
inline Type identifier Function name (Parameters) { Function body; }
```

Several attentions when using inline functions:

(1) Generally, loop statements and *switch* statements should not appear in an inline function.

(2) Inline function must be defined before its first call.

(3) Inline function does not support abnormal interface statements. (Abnormal interface statements will be discussed in Chapter 12.)

Generally, inline functions should be simple functions, with simple structures and few statements. Defining a complex function as an inline function may lead to code bloat and also increase the cost. In this case, most compilers will automatically convert the inline function into a common function before processing. What kind of functions should be counted as complex? The answer depends on compilers. Generally, functions that have loop statements cannot be processed as inline functions.

Therefore, the keyword *inline* is just a request. The compiler does not promise that every function with keyword *inline* will be processed as an inline function. Moreover, functions without the *inline* keyword can possibly be compiled as inline functions.

Example 3-14 An example of inline function.

```
//3_14.cpp
#include<iostream>
using namespace std;
inline double CalArea(double radius)
                                //Inline function, to calculate the area of a circle
{
    return 3.14 * radius * radius;
}

int main()
{
    double r(3.0);              //r is the radius of the circle
```