

第5章 函数

目录

- ▶ 5.1 函数概述
- ▶ 5.2 函数原型
- ▶ 5.3 全局变量与局部变量
- ▶ 5.4 函数调用机制
- ▶ 5.5 静态局部变量
- ▶ 5.6 递归函数
- ▶ 5.7 内联函数
- ▶ 5.8 重载函数
- ▶ 5.9 默认参数的函数

本章目标

- ▶ 理解什么是函数：过程模块
- ▶ 区分函数声明与定义
- ▶ 了解数学函数：数学库函数
- ▶ 掌握全局和局部变量的用法
- ▶ 了解递归函数

学习函数必须要理解的几个问题

主要问题： ★★★★★

- 1、要传递什么样的数据给函数？**
- 2、数据怎样传递给函数？**
- 3、要从函数获得什么结果？**
- 4、怎样从函数获得处理结果？**

5.1 函数概述

▶ 问题

- 程序功能模块的分解
- 程序中某些功能的反复执行

▶ 方法：用模块化程序设计的思路

- 采用“组装”的办法简化程序设计的过程
- 事先编好一批实现各种不同功能的函数
- 把它们保存在函数库中，需要时直接用

5.1 函数的概述

$$\frac{k!}{n! + m!}$$

```
cin>>k>>n>>m;
```

```
fk=1;  
for(i=1; i<=k; i++)  
    fk = fk*i;
```

```
fm=1;  
for(i=1; i<=m; i++)  
    fm = fm*i;
```

```
fn=1;  
for(i=1; i<=n; i++)  
    fn = fn*i;
```

```
cout<<fm/(fn+fk);
```

```
f=1;  
for(i=1; i<=n; i++)  
    f=f*i;
```

反复使用的代码段

```

# include <iostream>
using namespace std;
int main(){
    int k, m, n;
    float fk, fm, fn;
    cin>>k>>m>>n;
    fk=1;
    for(i=1; i<=k; i++)
        fk = fk*i;
    fm=1;
    for(i=1; i<=m; i++)
        fm = fm*i;
    fn=1;
    for(i=1; i<=n; i++)
        fn = fn*i;
    cout<<fm/(fn+fk));
}

```

```

# include <iostream>
using namespace std;
float fact(int n);
int main(){
    int k, m, n;
    float fk, fm, fn;
    cin>>k>>m>>n;
    fm = fact(m);
    fn = fact(n);
    fk = fact(k);
    cout<<fm/(fn+fk));
}
float fact(int n)
{
    int i;
    float f=1;
    for(i=1; i<=n; i++)
        f=f*i;
    return f;
}

```

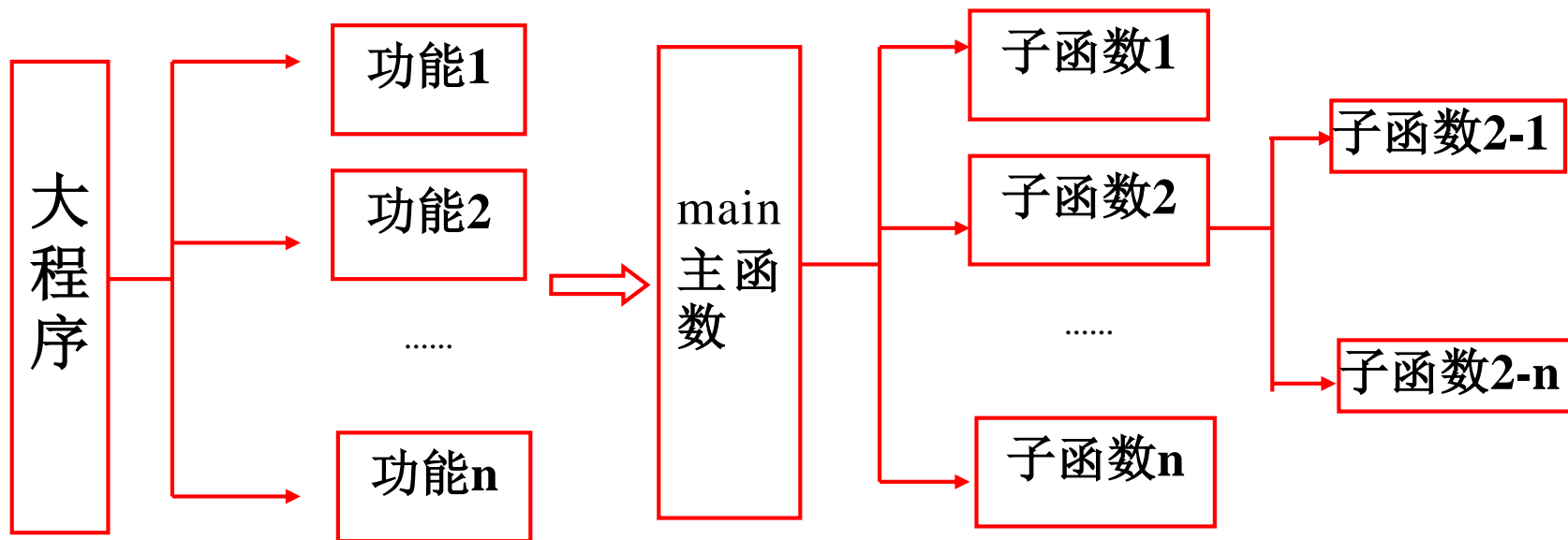
函数声明

函数定义

5.1 函数的概述

▶ 模块化程序设计

将大程序拆分成若干个**独立的小功能块**，每个独立的功能块利用函数完成，在组装成一个整体。



5.1 函数的概述

▶ 关于函数的几点说明：

- 一个源程序文件：一个或多个函数，C++语言的一个编译单位
- 一个C++程序：一个或多个源程序文件。
 - 优点：分别编写，分别编译，提高调试效率
- C++程序：
 - 起点：main函数
 - 终点：main函数
 - C++程序有且仅有一个main函数
- 函数：相互独立，平行，相互调用，多次调用
- 在程序设计中要善于利用函数，可以减少重复编写程序段的工作量，同时可以方便地实现模块化的程序设计。
- 函数可以有参数也可以无参数

函数类型

- ▶ 有参数有返回（函数） //完成数学计算

```
int bigger(int a, int b){ return a>b ? a : b; }
```

- ▶ 有参数无返回（过程） //履行指定过程

```
void delay(int a)
{
    for(int i=1; i<=a; i++); //延迟某个片刻
}
```

- ▶ 无参数有返回（函数） //完成任务

```
int geti()
{
    int x;
    cout<<"please input a integer:\n";
    cin>>x;
    return x;
}
```

- ▶ 无参数无返回（过程） //履行固有过程

```
void message(){ cout<<"This is a message.\n"; }
```

函数

▶ 功能

- 对程序进行模块化处理
- 函数内部定义的变量都是局部变量
- 参数是一个函数调用另一个函数时所传递的数据信息
- 参数是函数的局部数据，也是局部变量

▶ 函数的好处

- 分而治之，程序易管理
- 作为模块，软件可重用
- 功能独立，代码易维护

函数的声明

- ▶ C++语言要求，在程序中用到的所有函数，必须“**先定义，后使用**”
 - 指定**函数名字**、**函数返回值类型**、函数实现的**功能**以及**参数的个数与类型**，将这些信息通知编译系统。
- ▶ 作用：
仅告诉系统有这样一个用户自定义函数
- ▶ 声明格式：
 <返回类型> 函数名称(<参数列表>);



注意：分号不能少，否则编译错误

函数定义

定义的一般形式：

<类型标识符> 函数名称(<参数列表>)

{

变量声明; //C语言有此要求

功能实现的执行语句组;

}

函数体

说明

- 类型标识符：函数值的类型，定义了函数的运算结果类型，也就是函数返回值的类型。如果没有返回值，则返回类型为void。int可以省略，但不建议省略。
- 函数名称：与标识符的命名规则相同；不能与变量重复
- 参数信息：以逗号间隔的“<类型> 变量名”组，指明函数需要输入的信息

函数的返回值

▶ 返回语句:

`return [<表达式>];`

- 将表达式的值计算出来，并返回到上一层(函数之外)

▶ 说明:

- 如果函数不需要返回值，即为void类型，则return后面不需要<表达式>，甚至可以省略该return语句。
- 如果函数值类型与return语句中表达式值不一致，则以函数类型为准
- return语句可以用于从函数的任意位置返回到主调函数。

▶ 例如:

```
int ReadData(string filename, vector<int>& vec_int)    {  
    ifstream in(filename.c_str());  
    if (!in)  
        { return -1;}  
    //.....  
}
```

栈机制

▶ 程序运行时的内存布局

代码区：存放程序的执行代码

数据区：存放全局数据、常量、文字量、静态全局变量和静态局部变量

堆区：存放动态内存，供随机申请使用

栈区：存放函数数据区，动态地反映了程序运行中的函数状态；**遵循先进后去的原则**

进程空间



C++的函数调用过程：

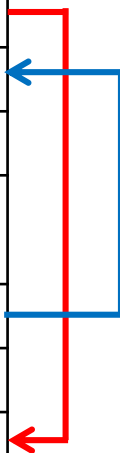
- ◆ 建立被调函数的栈空间，栈空间大小由函数定义体中的数据量决定
- ◆ 保护调用函数的运行状态和返回地址
- ◆ 传递参数
- ◆ 将控制权交给被调函数
- ◆ 函数运行完成后，复制返回值到函数数据块底部
- ◆ 恢复调用函数的运行状态
- ◆ 返回调用函数

```

//=====
// f0501.cpp
// 栈区运行的演示程序
//=====
int funcA(int x, int y);
int funcB(int s);
//-----
int main(){
    int a=6, b=12;
    a = funcA(a,b);
}//-----
int funcA(int x, int y){
    int n=5;
    x = funcB(n);
    return n;
}//-----
int funcB(int s){
    int x=8;
    return x;
}//=====

```

funcB函数		栈区.....
	局部变量x	8
	形参s	
		返回funcA地址
		funcA状态保护例程指针
funcA函数	局部变量n	5
	形参y	12
	形参x	6/8
		返回main地址
		main状态保护例程指针
	返回值	8
main函数	b	12
	a	6
		参数
		返回操作系统地址
		操作系统状态保护例程指针
		操作系统返回值0




```

//=====
// f0501.cpp
// 栈区运行的演示程序
//=====
int funcA(int x, int y);
int funcB(int s);
//-----
int main(){
    int a=6, b=12;
    a = funcA(a,b);
}//-----
int funcA(int x, int y){
    int n=5;
    x = funcB(n);
    return n;
}//-----
int funcB(int s){
    int x=8;
    return x;
}//=====

```

		栈区.....
		8
		返回funcA地址
		funcA状态保护例程指针
funcA函数	局部变量n	5
	形参y	12
	形参x	8
		返回main地址
		main状态保护例程指针
	返回值	8
main函数	b	12
	a	6
		参数
		返回操作系统地址
		操作系统状态保护例程指针
		操作系统返回值0

函数的参数

- ▶ 形式参数和实际参数
 - 形式参数：定义或者声明时出现的参数
 - 如： `int swap(int x, int y);`
 - 实际参数：函数调用时出现的参数
 - 如： `for(i=0; i<10;i++) sum += fun(i);`
 - 形式参数和实际参数具有各自独立的内存空间
 - 实参必须有确定的值，可以是常量、变量或表达式

■ C++语言中函数参数传值是单向的：

实际参数——> 形式参数

函数参数

▶ 函数调用时：

- 实参与形参：个数相同、类型一致、按顺序传递
- 实参 -> 形参，值传递是单向的，因此形参值的变化不会影响实参的值
- 实参和形参可以同名，也可以不同名

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

```
void main( )
{
    int x, y;

    cin>>x>>y;
    swap(x, y);
    cout<<x<<" "<<y<<endl;
}
```

swap	t= 3
	y=5/3
	x=3 /5
	返回地址等
main	y=5
	x=3

函数调用的说明

- ▶ 被调用函数必须是已经存在的函数
- ▶ 调用库函数时，一般应该在文件开头用#include命令将用到的库函数信息包含到本文件中。
- ▶ 调用自定义函数时，应该在调用函数前对函数进行声明
- ▶ 当主调函数和被调函数在同一个文件中，在主调函数前声明或者定义被调函数(此时定义和声明合并一起)
- ▶ 当主调函数和被调函数在不同文件中，在文件开头用#include命令包含被调函数的信息。

函数使用

返回一个int值

-函数声明，函数调用，函数定义是函数使用三部曲

```
int maximum(int x,int y,int z); //函数声明
```

```
int main()
```

```
{  
    int a=100, b=150,c=80;  
    int d = maximum(a,b,c); //函数调用：之前必须出现过函数声明  
    printf("%d\n",d);  
}
```

定义了3个参数

将a,b,c传给对应参数x,y,z

```
int maximum(int x,int y,int z) //函数定义
```

```
{  
    if(x>y) return x>z?x:z;  
    else    return y>z?y:z;  
}
```

运行结果：

150

函数使用

-函数声明，函数调用，函数定义是函数使用三部曲

```
int maximum(int x,int y,int z) //函数声明和定义合并
{
    if(x>y) return x>z?x:z;
    else    return y>z?y:z;
}
int main()
{
    int a=100, b=150,c=80;
    int d = maximum(a,b,c); //函数调用：之前必须出现过函数声明
    printf("%d\n",d);
}
```

运行结果：

150

使用库函数

- 使用库函数，则包含其头文件，等于声明且定义了该函数，例如

```
#include<cmath>    //math.h  
#include<cstring>  //string.h
```

说明：C++将C头文件改造(文件名前加c,且去掉.h)，纳入C++头文件

- 每个头文件代表一类库函数，例如cmath代表数学计算，cstring代表C字符串处理

```
#include<cmath>    //for sqrt  
#include<cstdlib>  //for strlen  
#include<iostream>  
using namespace std;  
int main()  
{  
    cout<<sqrt(5.8)<<"\n";  
    cout<<strlen("hello")<<"\n";  
}
```

- **注意：**C++标注库有自己的字符串处理类：string,头文件：
#include <string>

5.3 全局变量与局部变量

- ▶ 全局变量：变量在整个程序中都可读写，在函数的外部声明或定义。**可用于函数之间进行数据传递，但不建议这样使用。**程序退出才失效。
- ▶ 局部变量：变量只在一个函数内可读写，随着函数调用而产生，也随着函数的退出而失效。

```
int main()
{
    int m=n; //错:n无定义
    //...
}
int n;      //全局变量
void func()
{
    int s;   //局部变量
    n=s;     //ok:访问全局变量
    //...
}
```


5.5 静态局部变量

- ▶ 概念：static关键字标记的局部变量，**不随着函数退出而失效**，可用于累计自计算结果，自跟踪调用次数等。
- ▶ 存放：在内存分布的**全局数据区**
- ▶ 特点：
 - 只能被定义静态变量的函数使用
 - 在函数第一次调用时初始化，以后的函数调用不再进行初始化。

```
#include<iostream>
using namespace std;
int f()
{
    static int x=0;
    return ++x;
}
int main(){
    cout<<f()<<"\n"; //1
    cout<<f()<<"\n"; //2
}
```

5.6 递归函数

▶ 递归函数概述

- 递归函数只针对于子函数，不用于主函数
- 所谓递归：定义函数时，自己调用自己

▶ 递归函数的形式：直接调用；间接调用

直接递归调用：

```
functionA(parameters)
{
    .....
    functionA(realParameters);
    .....
}
```

间接递归调用：

```
functionA(parameters)
{
    .....
    functionB(realParameters);
    .....
}
functionB(parameters)
{
    .....
    functionA(realParameters);
    .....
}
```

5.6 递归函数

直接递归调用:

```
f1(parameters)
{
    .....
    f1(realParameters);
    .....
}
```

直接调用
本函数

f函数

调用f函数

间接递归调用:

```
f1(parameters)
{
    .....
    f2(realParameters);
    .....
}
f2(parameters)
{
    .....
    f1(realParameters);
    .....
}
```

间接调用
本函数

f1函数

f2函数

调用f2函数

调用f1函数

递归的前提条件

► 问题可以划分成

1) 简单的不需递归即可结束的: **结束递归的条件**

2) 递推形式, 即类似情况: **问题类似**

例如: 求 $f=n!$; 分成两种情况:

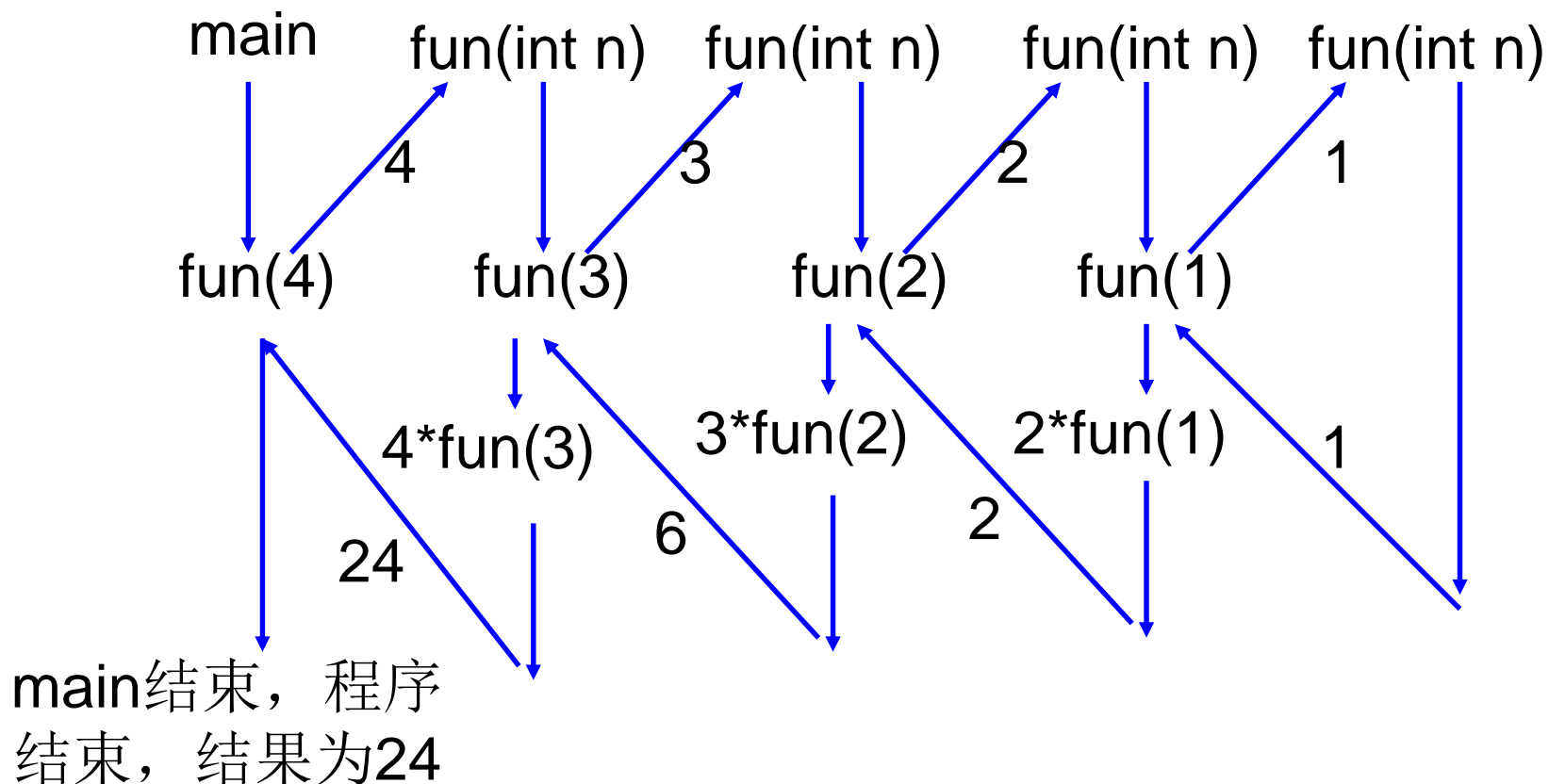
1) $n \leq 1$ 时, $f=1$;

2) $n > 1$ 时, $f=n \cdot [(n-1)!]$; **注意:** $(n-1)!$ 的求解与 $n!$ 的求解类似

这样得到程序为:

```
long fun(int n)
{
    if (n<=1)
        return 1;
    else
        return (n*fun(n-1));
}
```


递归函数的执行过程



递归函数举例

- ▶ 编写一个递归函数，判断一个任意整数是几位数？
 - 如何判断整数是几位数？
 - 提出整数的数字个数，每次提取整数的个位数，然后对整数缩小10。
 - 递归停止的条件
 - 当整数为个位数时，则停止。

```
int GetBitsOfNum(int num)
{
    if (num/10 == 0)
        return 1;
    else
        return 1+GetBitsOfNum(num/10);
}
```

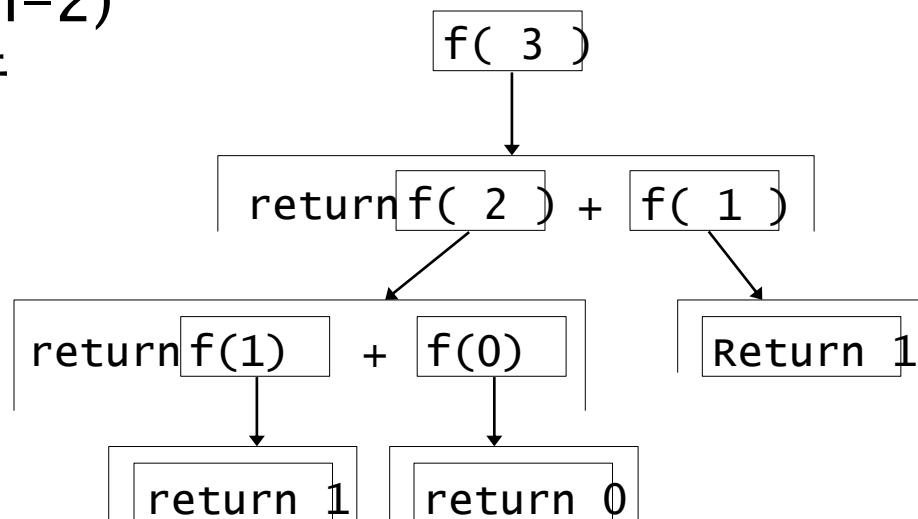


if (num==0)
return 0

递归函数举例

- ▶ Fibonacci序列：0,1,1,2,3,5,8
 - 每个数是前两个数之和
 - $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$
 - Fibonacci停止迭代的条件

```
long fibonacci(long n)
{
    if(n == 0 || n == 1)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```



内联函数

- ▶ 内联函数的需要性：保留函数调用形式，却消匿调用开销，从而提升频繁被调的小函数性能。将一些反复被执行的简单语句序列做成小函数
- ▶ 用法：在函数声明前加上inline关键字
 - 例如：`inline func(){.....};`
- ▶ 作用：不损害可读性又能提高性能
- ▶ 说明：内联函数能代替define定义的宏，但使用更安全。

频繁调用的函数：用昂贵的开销换取可读性

```
#include<iostream>
using namespace std;
bool isDigit(char); // 小函数
int main( )
{
    for(char c; cin>>c && c!='\n'; )
        if(isDigit(c))
            std::cout<<"Digit.\n";
        else std::cout<<"NonDigit.\n";
}
bool isDigit(char ch)
{
    return ch>='0' && ch<='9' ? 1 : 0;
}
```

问题：为什么会有存在昂贵的开销？

内嵌代码：开销虽少，但可读性差！

```
#include<iostream>
int main( )
{
    for(char c; cin>>c && c!='\n'; )
        if(ch>='0' && ch<='9' ? 1 : 0)
            std::cout<<"Digit.\n";
        else
            std::cout<<"NonDigit.\n";
}
```

内联方式：开销少，可读性也佳

```
#include<iostream>
using namespace std;
inline bool isDigit(char); // 小函数
int main( )
{
    for(char c; cin>>c && c!='\n'; )
    {
        if(isDigit(c))
            std::cout<<"Digit.\n";
        else std::cout<<"NonDigit.\n";
    }
    bool isDigit(char ch){
        return ch>='0' && ch<='9' ? 1 : 0;
    }
}
```

内联函数的几点说明

- ▶ 内联函数声明的位置：在函数调用前
- ▶ 构成合格的内联函数的规定：
 - 函数体尽可能要小，一般只有1~5行语句比较合适
 - 不能使用switch和while等控制语句
- ▶ 一般使用内联函数的场合
 - 函数体适当小
 - 程序中特别是在循环中反复执行该函数
 - 程序中并没有太多地方调用该函数，这样可以使得嵌入工作量少，最终程序代码量不会剧增。

5.8 函数重载

▶ 重载的需要性

- `int abs(int);`
- `long labs(long);`
- `double fabs(double);`

▶ 重载

- 定义:多个功能不同的函数共用1个函数名, 或者是对于在不同类型上作不同运算而又用同样名字的情况称为重载。
- eg:
 - `int abs(int);`
 - `long abs(long);`
 - `double abs(double);`

函数重载

▶ 匹配重载函数的顺序

- 通过1个严格的匹配，如果找到了，就使用那个函数
- 通过内部转换寻求一个匹配，只要找到了就用那个函数
- 通过用户定义的转换寻求一个匹配，若能查出有唯一的一组转换，就用那个函数

```
void print(double);
```

```
void print(int);
```

```
void func()
```

```
{
```

```
    print(1);           // 匹配void print(int); 规则1
```

```
    print(1.0);         // 匹配void print(double);规则1
```

```
    print('a');         // 匹配void print(int); 规则2
```

```
    print(3.1415f);     // 匹配void print(double); 规则2
```

```
}
```

函数重载

▶ 匹配重载函数的顺序

▶ **注意：** C++ 自动类型转换的特殊性：

int→long 和 int→double(error_overload)

```
void print(double);
```

```
void print(long);
```

```
void func(int a)
```

```
{
```

```
    print(a);    //错： 存在二义性
```

```
}
```

函数重载

▶ 使用重载的注意事项

- 重载函数的定义应至少在参数个数、参数类型或参数顺序上不同，若仅在返回类型上不同不能构成合法的函数重载；
- typedef不能用于建立新的类型，因此不能使用typedef来区分重载函数定义中的参数类型
- 重载函数的功能应该相同；若功能的意义截然不同则将极大地破坏程序风格；
- eg:将abs定义为求平方根的函数

函数重载

▶ 重载技术

- ▶ C++使用一种称为名称压轧(name mangling)技术来自动改变函数名，从而能够准确地判断出应该使用哪函数。
- ▶ 名称压轧：实际就是编译器在编译同名函数时，将函数的参数类型附加到函数名里，从而能达到区分不同参数的同名函数。
- ▶ 名称压轧技术在编译器内部实现，编程人员是不可见的。
- ▶ eg:
 - ▶ `int func(char a); => int func_c(char a)`
 - ▶ `int func(char a, int b, double c)`
`=> int func_cid(char a, int b, double c)`

5.9 函数参数的默认值

▶ 函数参数默认值的概念

- 在函数声明（或定义）时按照一定的规则为某些形式参数指定的缺省值称为**函数参数的默认值**。在函数调用时指定了默认值的参数可以不提供实参而直接采用默认值。

▶ 函数参数默认值的目的

- 为了使编程过程更为简单

▶ 默认参数的声明

- 当既有声明又有定义时，定义中不允许出现默认参数

▶ 默认参数的顺序规定

- 在指定默认值时从右往左，而调用时参数从左往右匹配

▶ 默认参数的指定不能出现多处

```
//=====
// 默认参数
//=====
#include<iostream>
using namespace std;
//-----
void delay(int a = 2);    // 函数声明时
//-----
int main(){
    cout<<"delay 2 sec.....";
    delay();              // 等于调用delay(2)
    cout<<"ended.\n";
    cout<<"delay 5 sec.....";
    delay(5);
    cout<<"ended.\n";
}//-----
```

```
void delay(int a)    // 函数定义时
```

```
{
```

```
    int sum=0;
```

```
    for(int i=1; i<=a; ++i)
```

```
        for(int j=1; j<1000; ++j)
```

```
            for(int k=1; k<100000; ++k)
```

```
                sum++;
```

```
}//=====
```

```
//=====
// 默认参数
//=====
#include<iostream>
using namespace std;
//-----
void delay(int a=2)    // 函数定义时
{
    int sum=0;
    for(int i=1; i<=a; ++i)
        for(int j=1; j<1000; ++j)
            for(int k=1; k<100000; ++k)
                sum++;
}
//=====
```

```
//-----
int main()
{
    cout<<"delay 2 sec.....";
    delay(); // 等于调用delay(2)
    cout<<"ended.\n";
    cout<<"delay 5 sec.....";
    delay(5);
    cout<<"ended.\n";
}
//-----
```

```
//=====
// many_defaults_function.cpp
// 多个默认参数值的函数匹配机制
//=====
```

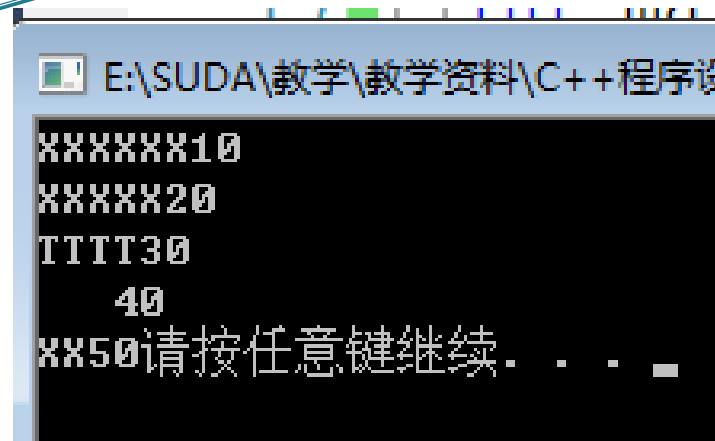
注意：

指定默认参数值只能按从右向左的顺序排列

```
void print(int num,int width=8,char fillchar='X',bool nextrow=true){
    cout<<setw(width)<<setfill(fillchar)<<num;
    if (nextrow)
        cout<<endl;
}

int main(){
    print(10);
    print(20,7);
    print(30,6,'T');
    print(40,5,' ',true);
    print(50,4,'X',false);
    return 0;
}
```

函数调用时，实参是按从左向右指定的



```
E:\SUDA\教学\教学资料\C++程序设计
XXXXXX10
XXXXXX20
TTTT30
      40
XX50请按任意键继续. . .
```

error_multi_default

```
void func(int a=1);
```

.....

```
void func(int a);
```

```
int main()
```

```
{
```

```
    int a=10;
```

```
    func();
```

```
    func(a);
```

```
    return 0;
```

```
}
```

```
void func(int a)
```

```
{
```

```
    cout<<a<<endl;
```

```
}
```

默认参数不能多处指定，否则存在函数调用时的二义性

默认参数

- ▶ 默认值指定过程中的规定
 - 默认值可以是字面常量、全局变量、全局常量和函数，但决不允许是局部变量
 - 一般使用常量。

默认参数与函数重载

▶ 默认参数和函数重载

- 默认参数的使用可以将一系列的功能相同但参数个数不同而已的若干个重载函数合并为1个函数
- 如果用1个参数的值来决定完全不同的操作，则使用重载函数较好

//使用不同进制输出整数，默认使用十进制

```
void print(int num, int base=10);
```

//比较两个对象的大小

```
bool compare(int x, int y);
```

```
bool compare(double, double);
```

```
bool compare(string, string);
```

```
bool compare(char*, char*);
```