C++ 语言基础

北京邮电大学

网络与交换技术国家重点实验室

宽带网研究中心

内容

- 基本数据类型
- 表达式
- 基本控制结构
- 函数

C++字符集

- 大小写的英文字母

  A~Z，a~z

- 数字字符

  0~9

- 符号

  ```
  !   #   %     ^     &     *     :
  _   +   =     -     ~     <     >     /     \
  '   "   ;     .     ,     ()    []    {}
  ```

- 其他符号

  〈空格符〉 〈换行符〉 〈TAB〉

词法记号

- 关键字　C++预定义的单词
- 标识符　程序员声明的单词，它命名程序正文中的一些实体
- 文字　在程序中直接使用符号表示的数据
- 操作符　用于实现各种运算的符号：+ - * / ...
- 分隔符　用于分隔各个词法记号或程序正文
- 空白符
  - 空格
  - 制表符
  - 换行符
  - 注释

关键字

- 关键字是C++保留的，作为专用定义符的单词，在程序中不允许另作它用

auto	break	case	char	class
const	continue	default	do	default
delete	double	else	enum	explicit
extern	false	float	for	friend
Goto	if	inline	int	long
Mutable	new	operator	private	protected
public	register	return	short	signed
sizeof	static	static_cast	struct	switch
this	ture	typedef	union	unsigned
Virtual	void	while		

标识符

- 标识符是对实体定义的一种定义符
  - 由字母或下划线开头、后面跟字母或数字或下划线组成的字符序列
  - 具有特定的有效长度
    - ANSI C 标准规定 31 个字符
  - 用来标识用户定义的常量名、变量名、函数名、文件名、数组名、和数据类型名和程序等。
  - 大写字母和小写字母代表不同的标识符

文字常量

- 数字常量

  1　　100　　200U　　10000L

  0x100　　3.14　　1.25e+10

- 字符常量

  ‘a’　　‘b’

- 字符串

  - 字符串是由双引号括起来的字符串常量

    “China”、“C++ Program”

分隔符与空白符

- 分隔符
  - 用于分隔各个词法记号或程序正文

    { } ( ) 和空白符
- 空白符
  - 空格、换行、制表符
  - 注释
    - 是用来帮助阅读、理解及维护程序
    - 注释部分被忽略，不产生目标代码
    - C++语言提供两种注释方式：

      一种是与C兼容的多行注释，用 /* 和 */ 分界

      另一种是单行注释，以“//”开头，表明本行中“//”符号后的内容是注释

C++ 简单实例

```
//2_1.cpp
#include <iostream>
using namespace std;

int main()
{
    cout<<"Hello!\n";
    cout<<"Welcome to c++!\n";
}
```

运行结果：

```
Hello!
Welcome to c++!
```

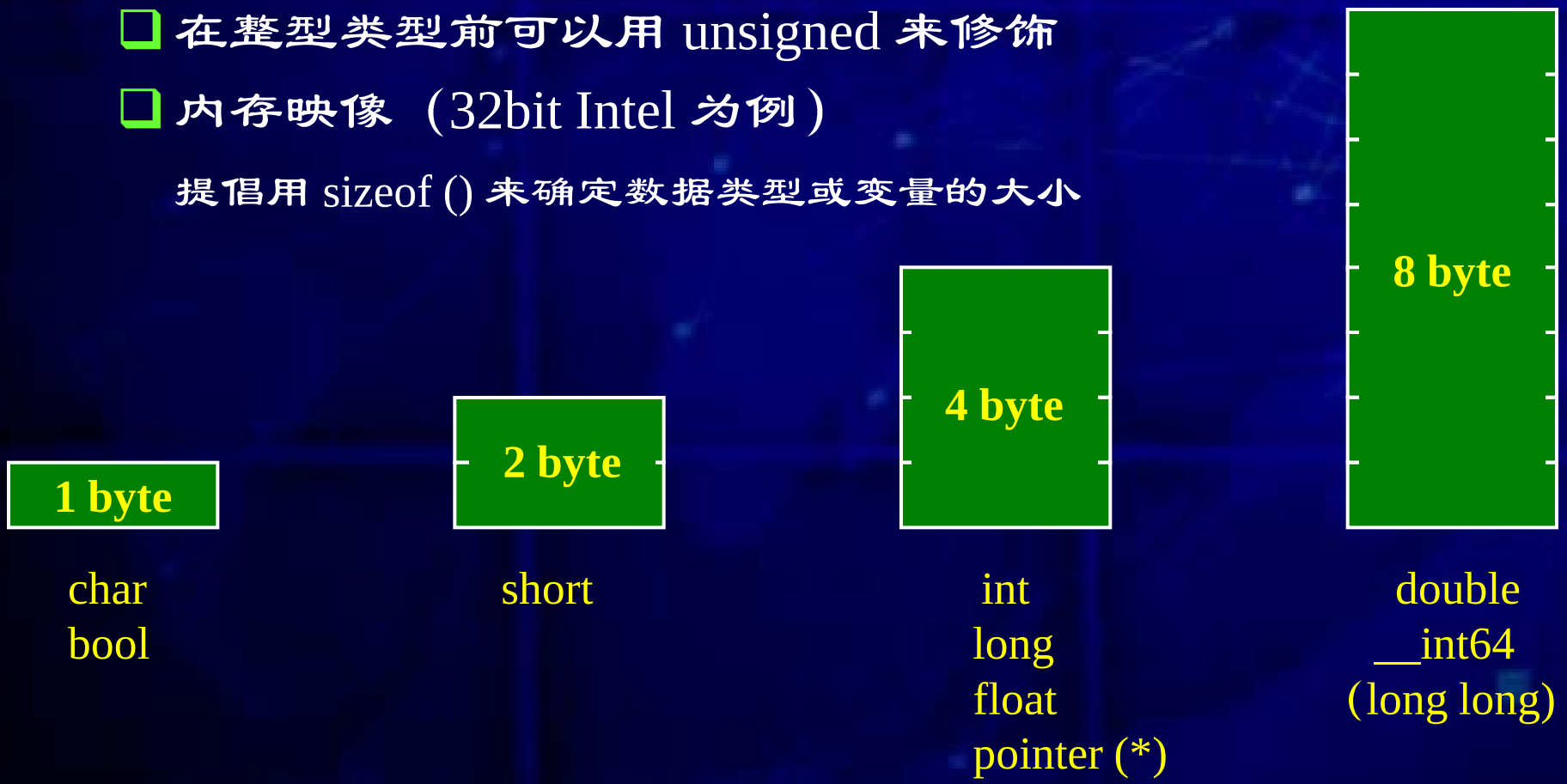
C++ 简单实例

```
#include <iostream>
using namespace std;

int main(void)
{
   const int PRICE = 30;                    // 符号常量

   int num, total;                          // 变量
   num = 10;
   total = num * PRICE;
   cout << "Total:" << total <<endl;    // 字符串常量

   float v,r,h;
   r = 2.5;                                 // 文字常量
   h = 3.2;
   v = 3.14159 * r * r * h;

   cout << v << endl;
}
```

数据类型

- 内部数据类型
  - char int short long __int64 float double bool 指针 (*)
  - 在整型类型前可以用 unsigned 来修饰
  - 内存映像 (32bit Intel 为例)

    提倡用 sizeof () 来确定数据类型或变量的大小



枚举类型—enum

- 只要将需要的变量值一一列举出来，便构成了一个枚举类型。
- 枚举类型的声明形式如下：

  enum 枚举类型名 {变量值列表};
- 例如：

  enum weekday {sun,mon,tue,wed,thu,fri,sat};
- 枚举元素具有缺省整型值，它们依次为： 0,1,2,.....。
  - 也可以在声明时另行指定枚举元素的值，如：

    enum weekday {sun=7,mon=1,tue,wed,thu,fri,sat};

数据类型

- 用户定义数据类型/抽象数据类型（UDT/ADT）
  - 使用 struct / class 定义

```
struct  Sample_Struct
{
    int    iCount;
    char * pName;
}   sample;

class   CComputer
{
private:
    char * m_pComputerName;
    ...
public:
    void  PowerOn();
    void  Startup();
    ...
};
```

联合

- 声明形式：

```
union 联合名
{
    数据类型   成员名 1;
    数据类型   成员名 2;
        :
    数据类型   成员名 n;
};
```

- 联合的成员共享相同的地址空间
- 联合体类型变量说明的语法形式

  联合名 联合变量名;

- 引用形式：

  联合变量.成员名　　联合指针->成员名

typedef 语句

- 为一个已有的数据类型另外命名
- 语法形式

  typedef 已有类型名 新类型名表;
- 例如

```
typedef double area,volume;
typedef int    natural;
natural i1,i2;
area    a;
volume  v;
```

变量初始化

例：

```
int       a = 3;
double    f = 3.56;
char      c = 'a';
int       i(5);
```

变量的存储类型

- auto （变量的缺省类型）
  - 临时存储变量，具有生命周期，生命期结束后存储空间可以被其他自动变量覆盖使用
- register
  - 建议使用通用寄存器
- extern
  - 声明该变量的定义不一定在本源文件中
- static
  - 静态，在程序的整个生命周期（执行期间）有效

```
extern int e;
void func(int a, int b)
{
   int           i;
   auto      int j;
   static    int x;
   register int y;

}
```

```
void func(int a, int b)
{
   int i;
   ...
   {
       int i = 1;
       i++;
       ...
   }
}
```

类型转换

- 不同类型数据进行混合运算时，C++编译器会自动进行类型转换。
- 为了避免不同的数据类型在运算中出现类型问题，应尽量使用同种类型数据。
- 可以采用强制类型转换

强制类型转换

- 语法形式：

  类型说明符(表达式)

  或

  (类型说明符)表达式

- 强制类型转换的作用是将表达式的结果类型转换为类型说明符所指定的类型

```
float c;
int a , b;
c = float(a) / float(b);
c = (float)a / (float)b;
```

运算符

Operator	Name or Meaning	Associativity
::	Scope resolution	None
.	Member selection (object)	Left to right
->	Member selection (pointer)	Left to right
[ ]	Array subscript	Left to right
( )	Function call	Left to right
( )	member initialization	Left to right
++	Postfix increment	Left to right
--	Postfix decrement	Left to right
typeid( )	type name	Left to right
const_cast	Type cast (conversion)	Left to right
dynamic_cast	Type cast (conversion)	Left to right
reinterpret_cast	Type cast (conversion)	Left to right
static_cast	Type cast (conversion)	Left to right

```
CComputer::PowerOn()
Comp.Mouse
pComp->Mouse
int array[10]
printf("%d\n", a);
CComputer Comp(10);
a++
a--
typeid(Comp)
```

运算符

sizeof	Size of object or type	Right to left	sizeof(CComputer)
++	Prefix increment	Right to left	++a
- -	Prefix decrement	Right to left	--a
~	One's complement	Right to left	~a
!	Logical not	Right to left	!a
-	Unary minus	Right to left	-a
+	Unary plus	Right to left	+a
&	Address-of	Right to left	&Comp
*	Indirection	Right to left	*pComp
new	Create object	Right to left	pComp = new Computer;
delete	Destroy object	Right to left	delete pComp;
()	Cast	Right to left	(unsigned)a
.*	Pointer-to-member (objects)	Left to right	Comp.*Startup()
->*	Pointer-to-member (pointers)	Left to right	pComp->*Startup()

运算符

*	Multiplication	Left to right
/	Division	Left to right
%	Modulus	Left to right
+	Addition	Left to right
-	Subtraction	Left to right
<<	Left shift	Left to right
>>	Right shift	Left to right
<	Less than	Left to right
>	Greater than	Left to right
<=	Less than or equal to	Left to right
>=	Greater than or equal to	Left to right
==	Equality	Left to right
!=	Inequality	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise inclusive OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
e1?*e2*:*e3*	Conditional	Right to left
=	Assignment	Right to left
*=	Multiplication assignment	Right to left
/=	Division assignment	Right to left
%=	Modulus assignment	Right to left
+=	Addition assignment	Right to left
-=	Subtraction assignment	Right to left
<<=	Left-shift assignment	Right to left
>>=	Right-shift assignment	Right to left
&=	Bitwise AND assignment	Right to left
|=	Bitwise inclusive OR assignment	Right to left
^=	Bitwise exclusive OR assignment	Right to left
throw *exp*	throw expression	Right to left
,	Comma	Left to right

表达式

- 算术表达式
  - 基本算术运算符：+　　-　　*　　/ (若整数相除，结果取整)
- 赋值表达式：= +=　-=　*=　/=　%=　<<=　>>=　&=　^=　|=
- 逗号表达式：表达式1，表达式2
- 关系表达式：<　　<=　　>　　>=　　==　　!=
- 逻辑表达式：!　　&&　　||
- 条件表达式：表达式1 ？表达式2 ： 表达式3

sizeof 运算符

- 语法形式

  sizeof (类型名)

  sizeof (变量名)

  sizeof (表达式)

- 结果值：

  “类型名”所指定的类型或“表达式”的结果类型所占的字节数。

- 例：

  sizeof (short)

  sizeof (x)

位运算

- 按位与 (&)
- 按位或 (|)
- 按位异或 （^）,即半加
- 按位取反 （~）
- 左移运算（<<）
- 右移运算（>>）

C ++ 语句

- 声明语句
- 表达式语句
- 选择语句
- 循环语句
- 跳转语句
- 复合语句
- 标号语句

语句之间需要使用分隔符；

```
int a = -1;
if (a < 0) a = 0;
for (int i=0; i<10; i++)
{
        a ++;
        a += i;
}
if (a < 0) a = 0;
```

C++ 语句

- 声明语句：声明或定义变量、函数等

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

- 表达式语句：

```
a = 100;
```

- 将多个语句用大括号包围，便构成一个复合语句：

```
{
   sum = sum + i;
   i ++;
}
```

简单的输入、输出

- 向标准输出设备（显示器）输出

例：

```
int x;
cout << “x=“ << x;      // 表达式语句
```

- 从标准输入设备（键盘）输入

例：

```
int x;
cin >> x;                // 表达式语句
```

其中，cin、cout 是 C++ 流式 IO 库中定义的

所以使用时一般会有如下语句：

```
#include <iostream>
using namespace std;
```

算法的基本控制结构

- 顺序结构
- 分支结构： if， switch
- 循环结构： while， do-while， for，break 和 continue

函数

函数的声明

- 函数声明的语法形式

```
类型标识符 函数名（形式参数表）
{
  语句序列
}
```

是被初始化的内部变量，寿命和可见性仅限于函数内部

若无返回值，应写void

函数的声明

- 形式参数表

  $<type_1>\ name_1, <type_2>\ name_2, ..., <type_n>\ name_n$

- 函数的返回值
  - 由 return 语句给出，例如：

    ```
    return  0;
    ```

  - 无返回值的函数（void类型），不必写return语句

函数的调用

- 调用必须前先定义该函数，或声明函数原型：
  - 函数原型说明：

    类型标识符 被调用函数名 （含类型说明的形参表）；

    例如：

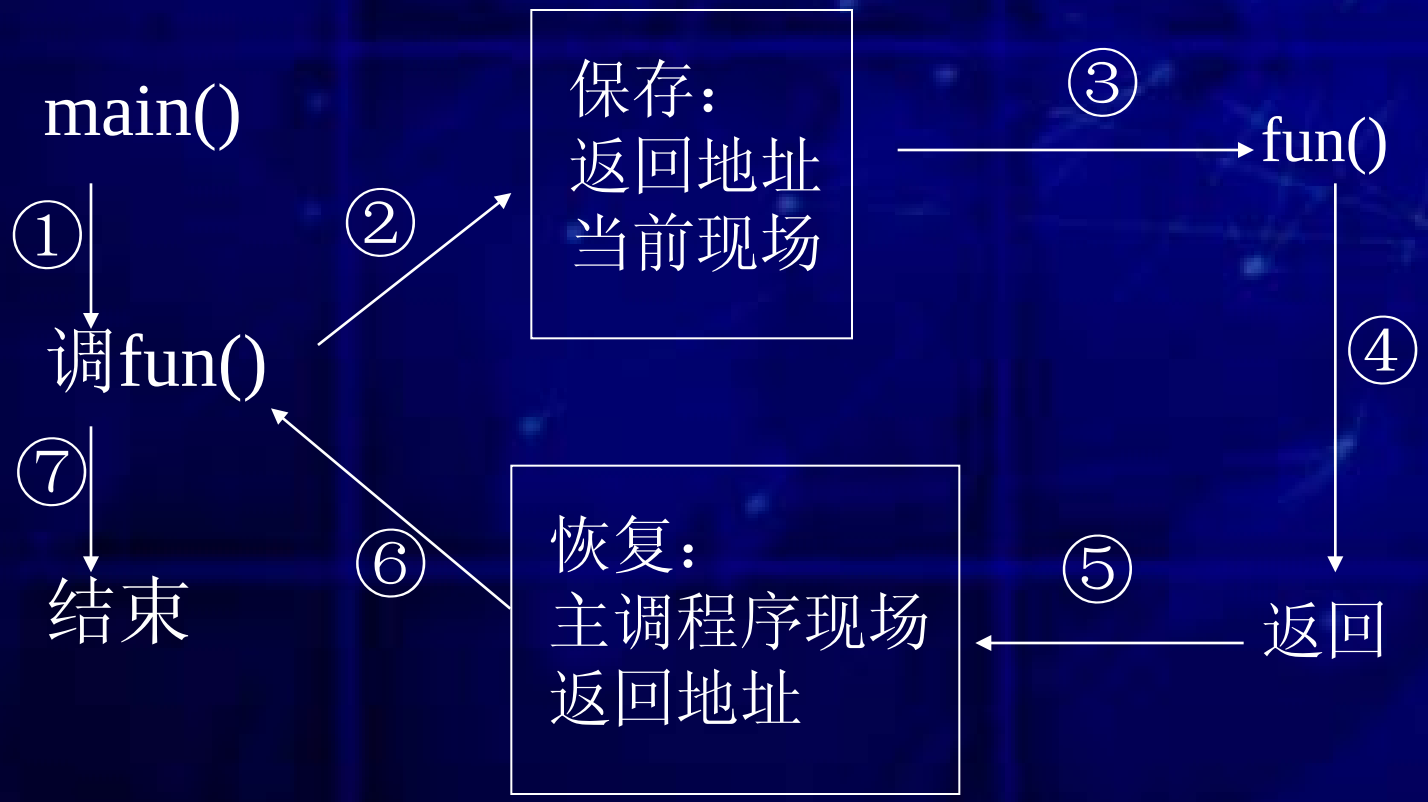
```
void func(int a, int b);
float foo(float a);
```

- 调用形式

  函数名（实参列表）

- 嵌套调用
  - 函数可以嵌套调用，但不允许嵌套定义。
- 递归调用
  - 函数直接或间接调用自身。

函数调用的执行过程



嵌套调用

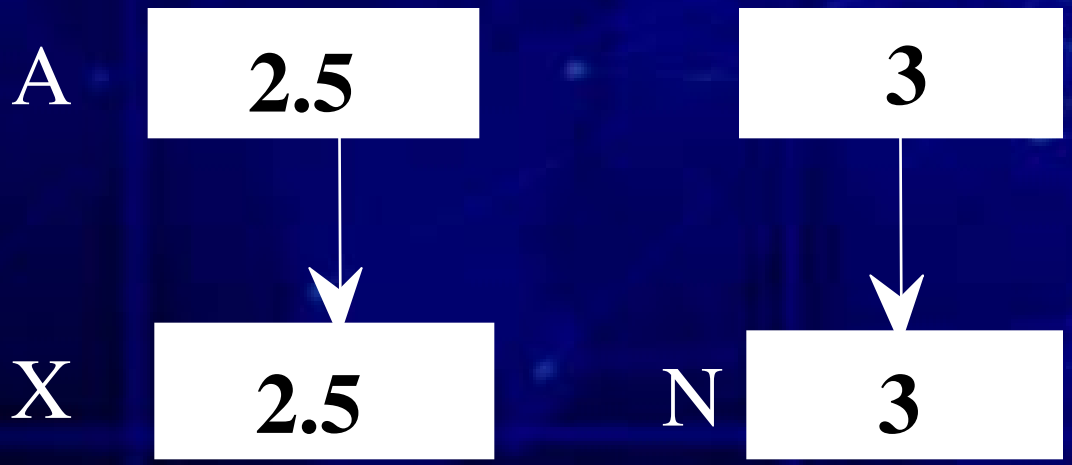
函数的声明与使用

函数的参数传递机制

- 在函数被调用时才分配形参的存储单元。
- 实参可以是常量、变量或表达式。
- 实参类型必须与形参相符。
- 传递时是传递参数值，即单向传递。

函数的参数传递机制

调用者：　　D = power(A,3)



被调函数：

double power(double X, int N)

例：参数传递

```
#include<iostream>
using namespace std;

void Swap(int a, int b);

int main()
{
   int a(5), b(10);

   cout << “(a,b)=“ << a << “,” << b << endl;
   Swap(a,b);
   cout << “(a,b)=“ << a << “,” << b << endl;
   return 0;
}

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

输出结果：

```
(a,b)=5,10
(a,b)=5,10
```

例：参数传递（使用指针）

```
#include<iostream>
using namespace std;

void Swap(int *a, int *b);

int main()
{
   int a(5), b(10);

   cout << “(a,b)=“ << a << “,” << b << endl;
   Swap(&a,&b);
   cout << “(a,b)=“ << a << “,” << b << endl;
   return 0;
}

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

输出结果：

```
(a,b)=5,10
(a,b)=10,5
```

数据类型：引用

- 引用(&)是标识符的别名,例如:

```
int  i, j;
int& ri=i;          //建立int型的引用ri,初始化为变量i的引用
                    //
j  = 10;
ri = j;             //相当于 i=j;

double  f;
double &rf = f;
rf = 100.5;         //相当于 f=100.5;
```

- 声明一个引用时，必须同时对它进行初始化，使它指向一个已存在的变量（对象）。
- 一旦一个引用被初始化后，就不能改为指向其它对象。
- 引用可以作为函数的参数进行传递，也可以作为返回值

```
void swap(int& a, int& b) {...}
```

例：参数传递（使用引用）

```
#include <iostream>
using namespace std;

void Swap(int& a, int& b);

int main()
{
   int a(5), b(10);

   cout << "(a,b)=" << a << "," << b << endl;
   Swap(a,b);
   cout << "(a,b)=" << a << "," << b << endl;
   return 0;
}

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

输出结果：

```
(a,b)=5,10
(a,b)=10,5
```

内联函数

- 声明时使用关键字 inline。

```
inline void Swap(int& a, int& b)
{
   int t;
   t=a; a=b; b=t;
}
int main()
{
   int a(5), b(10);
   Swap(a,b);
   return 0;
}
```

- 编译时在调用处用函数体进行替换,节省了参数传递、控制转移等开销。
  - 内联函数的声明必须出现在内联函数第一次被调用之前

默认参数

- 函数在声明时可以预先给出默认的形参值，调用时如给出实参，则采用实参值，否则采用预先给出的默认形参值。
- 例如：

```
int add(int x=5, int y=6)
{
   return  x+y;
}

int main()
{
   add(10,20);      // 10+20
   add(10);         // 10+6
   add();           // 5+6
   return 0;
}
```

默认参数

- 默认形参值必须从右向左顺序声明，并且在默认形参值的右面不能有非默认形参值的参数。因为调用时实参取代形参是从左向右的顺序。
- 例：

```
int add(int x,   int y=5, int z=6); //正确
int add(int x=1, int y=5, int z);   //错误
int add(int x=1, int y,   int z=6); //错误
```

默认参数

- 若调用出现在函数体实现之后，默认参数可在函数实现时给出
- 默认参数值也可以函数原型中给出，但默认参数只能定义一次

```
int add(int x, int y);
void func()
{
   add(10);
}
 int add(int x=5, int y=6)
 {
   return  x+y;
 }
```

```
int add(int x=5, int y=6);
int main()
{
   add(10);          // 10+6
   add();            // 5+6
   return 0;
}
 int add(int x, int y =6 )
 {  return  x+y;
 }
```

```
int add(int x, int y);
void func()
{
   add(10,20);
   add(10);      //不能使用默认参数
}

int add(int x=5, int y=6);
int main()
{
   add(10);          // 10+6
   add();            // 5+6
   return 0;
}

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

默认参数

- 在相同的作用域内，默认形参值的说明应保持唯一，但如果在不同的作用域内，允许说明不同的默认形参。
- 例：

```
int add(int x=1,int y=2);

int main()
{
    int add(int x=3,int y=4);
    add();              //使用局部默认形参值（实现3+4）
    retrun 0
}

void fun(void)
{   ...
     add();             //使用全局默认形参值（实现1+2）
}
```

函数重载（function overloading）

- C++允许功能相近的函数在相同的作用域内以相同函数名声明，从而形成重载
- 实际意义和功能相同或相近，方便使用，便于记忆
- 编译器自动根据参数类型、个数等调用相应的函数

```
int   add(int x, int y);
float add(float x, float y);

int add(int x, int y);
int add(int x, int y, int z);


int   x, y, z;
float e, f;

add(x,y);
add(x,y,z);
add(e,f);
```

形参类型不同

形参个数不同

函数重载

- 重载函数的形参必须不同：个数不同或类型不同。
- 编译程序将根据实参和形参的类型及个数的最佳匹配来选择调用哪一个函数。

```
int add(int x,int y);
int add(int a,int b);
```

编译器不以形参名来区分重载

```
int add(int x,int y);
void add(int x,int y);
```

编译器不以返回值来区分重载

- 不要将不同功能的函数声明为重载函数，以免出现调用结果的误解、混淆