
中国科学技术大学
University of Science and Technology of China
程序设计 II
Programming Design II
C++的函数
主讲：吴锋

目录
CONTENTS

◎ 函数的参数传递

- 传递过程：实参 ←→ 形参
- 值传递：把实际参数的值复制给形式参数
  - 形参和实参是两个不同的内存空间。
  - 函数所处理的仅仅是实际参数的一个拷贝。
- 引用传递：把实际参数的地址传递给形式参数
  - 形参和实参实际上是同一个内存空间。
  - 通过对形参的处理来达到对实参进行加工的目的。

函数的引用传参和调用

- 函数定义：
  - 形参声明为引用量；
  - 函数体直接对引用量操作；
- 函数调用：
  - 形式上象值传递，实际上是引用传递。
  - 形参是实参的别名，没有新开辟空间。
  - 对形参的修改直接影响到实参。

```
void swap(int &v1, int &v2) {
     int temp = v2;
     v2 = v1;
     v1 = temp;
}

int main() {
    int a=5, b=9;
    swap(a, b);
    return 0;
}
```

◎ 函数的引用传参和调用

- 主要优点
  - 形式参数只是实参的别名，因此函数调用时不需要为它分配空间，尤其是传递大的对象。
  - 在函数内对参数值的改变不再作用于局部拷贝，而是直接针对实参。
  - 相对于指针传递而言，引用传递的函数体定义和调用的形式更为简洁。

```
void swap1(int *v1, int *v2){
     int temp = *v2;
     *v2 = *v1;
     *v1 = temp;
}

void swap2(int &v1, int &v2){
     int temp = v2;
     v2 =v1;
     v1 = temp;
}

int main() {
    int a=5, b=9;
    swap1(&a, &b);
    swap2(a, b);
    return 0;
}
```

函数的传数组调用

- 情形一：形参和实参都用数组

```
#include <iostream>
using namespace std;
int a[8] = {1,3,5,7,9,11,13};

// 本质上是传指针，b指针是a指针的一个拷贝
void fun(int b[], int n){
    for (int i = 0; i < n - 1; i++){
        b[7] += b[i];
    }
}

int main (){
    int m = 8;
    fun(a, m);
    cout << a[7] << endl;
    return 0;
}
```

函数的传数组调用

- 情形二：实参用数组名，形参用指针

```
#include <iostream>
using namespace std;
int a[8] = {1,3,5,7,9,11,13};

// p 指针是a指针的一个拷贝
void fun(int *p, int n){
   for (int i = 0; i < n - 1; i++){
      *(p + 7) += *(p + i);
   }
}

int main (){
    int m = 8;
    fun(a, m);
    cout << a[7] << endl;
    return 0;
}
```

函数的传数组调用

- 情形三：实参用数组名，形参用引用

```
#include <iostream>
using namespace std;
int a[8] = {1,3,5,7,9,11,13};

// b指针是a指针的一个引用，必须指定数组大小
void fun(int (&b)[8], int n){
    for (int i = 0; i < n - 1; i++){
        b[7] += b[i];
    }
}

int main (){
    int m = 8;
    fun(a, m);
    cout << a[7] << endl;
    return 0;
}
```

C++的作用域

- 作用域的屏蔽规则
  - 在某个作用范内定义的标识符在该范围内的子范围内可以定义定义新的同名标识符。
  - 这时原定义的标识符在子范围内是不可见的，但是它还是存在的，只是在子范围内由于出现了同名的标识符而被暂时隐藏起来。
  - 过了子范围后，它又是可见的。

```
int x;
void main()
{
    int a;
    {
        int a;
        a = 10;
        ::a = 20;
    }
}
```

全局作用域分辨符

```
const int max = 65000;
const int LineLength = 12;

void fibonacci(int max) {
        if (max < 2) {
          return;
        }

        for (int i =3, vl = 0, vl = 1, cur; i <= max; ++i) {
          cur = v1 + v2;
           if (cur > ::max) {
             break;
          }
           cout << cur << " ";
           v1 = v2;
           v2 = cur;
           if (i % LineLength == 0) {
             cout << "\n";
          }
       }
}
```

C++的名字空间

- 名字空间允许为自定义的作用域命名，允许嵌套定义名字空间，以及设置别名。
  - 作用：限制标识符作用域，避免重名冲突

```
namespace cplusplus_Fibonacci {
    const int max = 65000;
    const int LineLength = 12;
    void fibonacci(int max) { … }
}
cplusplus_Fibonacci::fibonacci(3);

using namespace cplusplus_Fibonacci;
fibonacci(3); // 可以直接使用名字空间中变量或函数

namespace cpp_Fib = cplusplus_Fibonacci; // 设置简短别名
using cpp_Fib::max; // 后面可以直接使用变量 max
```

C++的内联函数

- 内联函数（关键字 inline）
  - C++对宏替换的改进（类似const之于变量）
  - 告诉编译器，在编译时用函数代码的“拷贝”替换函数调用，达到宏替换相同的效果

```
inline int power_int(int x){
      return   x*x;
}

void main( ) {
       for (int i = 1; i <= 10; i++) {
            int p = power_int(i);      // 编译时换成 (i*i)
            cout << i << "*" << i << "=" << p << endl;
       }
}
```

C++的内联函数

- 内联函数的优点：
  - 提高代码效率（没有函数调用带来的消耗）
  - 保持源代码可读性及易维护性
- 内联函数的限制：
  - 在内联函数内不允许用循环语句和开关语句
  - 内联函数的定义必须出现在每一个调用该函数的源文件之中（由于要进行源代码替换）
  - 编译器不保证所有被定义为inline 的函数编译成内联函数，例如：递归函数

◎ 内联函数 vs 直接代码

- 内联函数形式上具有一般函数的特点：
  - 可读性强

```
max(a, b)        vs        (a > b ? a : b);
```

  - 复用性好

```
max(a, b)        vs        (a > b ? a : b);
max(c, d)        vs        (c > d ? c : d);
```

  - 便于修改，易于维护

◎ 内联函数 vs 宏替换

- 内联函数由编译器处理，而宏由预处理器处理
- 编译器会对内联函数调用的实参进行类型检查
- 宏可能会有副作用（只是简单的文本替换），而内联函数不会有副作用（经过了编译器的编译）

```
#define abs(n)    ((n)<0 ? -(n):(n)) ;
i = 1;
j = abs(--i);   // 实际上j= ( (--i)<0 ? -(--i) : ( -- i) ) ;

#define power(n)  (n*n)
k = power(1+2);  //    实际上k=(1+2*1+2)
```

函数的默认参数

- C++允许设置函数参数的默认值

```
typedef struct time {
      int h;                 // hour
      int m;                // minute
      int s;                 // second
} *TIME, CLOCK;

int setclock(TIME t, int hour=12, int minute=0, int second=0) {
      t->h = (hour <= 23 && hour >=0)? hour : 12;
      t->m = (minute<=59 &&minute>=0)? minute:0;
      t->s = (second<=59 && second >=0)?second:0;
}
```

函数的默认参数

- C++允许设置函数参数的默认值

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

void main() {
      CLOCK t1, t2, t3, t4, t5;

      setclock(&t1);                  // t1=12:00:00
      setclock(&t2, 7);               // t2=07:00:00
      setclock(&t3,14,20);            // t3=14:20:00
      setclock(&t4,18,30,0);          // t4=18:30:00
      setclock(&t5,19, , 0) ;         // 非法
}
```

函数的默认参数

- 若有实际参数值传入，则缺省值无效（被替代）；否则，该参数值就采用缺省值

```
setclock(&t1);              // t1=12:00:00
setclock(&t2, 7);           // t2=07:00:00
setclock(&t3,14,20);        // t3=14:20:00
```

- 带有缺省值的参数必须全部放置在参数的最后，即在带有缺省值的参数的右边不再出现无缺省值的参数（否则调用时编译器无法判断对应的参数）

```
int setclock(TIME t, int hour=12, /*非法*/ int minute, int second) {…}
```

函数的默认参数

- 函数的参数既可以在定义又可以在声明中指明，一旦定义了缺省参数，就不能再定义它，但可以添加一个或多个缺省参数。

```
void f(int a, int b, int c =0);      // 定义缺省参数c
void f(int a, int b=1, int c=0);     // 增加缺省参数b
void f(int a, int b=2, int c=1);     // 错误，企图重定义b和c的缺省参数
```

C++函数的重载

- 重载的概念：把多个功能类似的函数定义成同名函数
- 函数重载的实现
  - 核心问题： 编译器能够确定应执行哪一个函数，由于函数是静态关联，需要在编译阶段确定

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

  - 编译器根据什么来区分调用哪个函数?
    - 函数名：相同，无法区分
    - 返回值类型：语句中不出现返回值类型，也无法区分
    - 参数：参数的个数和类型不同，可以区分。

C++函数的重载

- 参数类型不同的重载函数

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

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

double add(double a, double b) {
     return a + b;
}

void main( ){
     // 调用 add(int, int )
     cout << add(5, 10) << endl;

     // 调用 add(double, double)
     cout << add(5.1, 10.5) << endl;
}
```

C++函数的重载

- 参数个数不同的重载函数

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

int min(int a, int b) {
  return a<b ? a : b;
}
int min(int a, int b, int c) {
  return min(min(a, b), c);
}
int min(int a, int b, int c, int d) {
     return min(min(a, b), min(c, d));
}

void main() {
     cout << min(13, 5, 4, 9) << endl;
     cout << min(-2, 8, 0);
}
```

C++函数的重载

- 重载函数的“二义性”错误
  - 返回值类型不同，参数相同

```
int    print(int);
double   print(int);

void main() {
      int a = 10, b = 20;

      // 调用哪个print???
      print(a);    // 二义性
      print(b);    // 二义性
}
```

C++函数的重载

- 重载函数的“二义性”错误
  - 仅用了const或引用造成的参数类型不同

```
int print(const int &);
int print(int );

void main() {
	int a = 10, b = 20;

	// 调用哪个print???
	print(a);	// 二义性
	print(b);	// 二义性
}
```

C++函数的重载

- 重载函数的“二义性”错误
  - 使用了修饰符造成的的参数类型不同的函数

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

void main() {
     // 注意：1l 既可转换成unsigned int，又可转换成int。

     // 调用哪个print???
     print(1l);  // 二义性
     print(1u) ; // OK,无二义
}
```

C++函数的重载

- 重载函数的“二义性”错误
  - 使用缺省参数引起的二义性

```
void print(int a , int b=1);
void print(int a);

void main() {
        // 到底是一个参数的print，
        // 还是取缺省值b为1的print.
        // 调用哪个print???
        print(1);
}
```

C++函数的重载

- 编译器如何处理重载函数
  - C的函数编译后通常为 _cdecl_funcname();

    如：int max(int,int)　　编译后为：

    int _cdecl_max()(0040260);
  - 为了支持重载，C++的函数编译后通常为 _cppdecl_funcname_parameter_...();

    如：int max(int,int); int max(int,int,int) 编译后为：

    int _cppdecl_max_int_int()(0080330) ;

    int _cppdecl_max_int_int_int()(0100280) ;

C++函数的重载

- 关于extern “C” 声明：
  - 强制C++编译器按C的函数命名规则去连接相应的目标代码（例如：函数用C实现且编译成了库，C++中需要调用该函数时）

```
extern “C” max(a, b);
```

编译为：

```
call _cdecl_max()(0040260);
```

  - 使得函数编译后与库中C语言编译出的函数名一致，从而实现正确调用的目的。

C++函数模板

- 函数模板实际上是定义了一组函数，也是C++的一种代码重用机制，是在强类型语言上实现的一种弱类型机制。
- 函数模板的定义：

```
template <模板参数表>
返回值类型      函数名（函数参数表）
{
        //函数模板的定义
}
```

C++函数模板

- 模板参数类型参数由关键字 class 或 typename 后加一个标识符 T 组成，可以有多个。
- 这里T可以是 int, char ,float 等内置类型 或者 任何重载了>算符的自定义对象。
- 函数模板也可以定义为内联函数。

```
template <class T>
inline T max(T a, T b){
      return    a > b ? a : b;
}
```

C++函数模板

- 模板参数必须在函数参数声明中被使用，且编译器不考虑模板实例的返回类型。

```
template <class T>
void f() {
    // error, 函数参数列表中无函数模板参数T
    T a;
    // …
}
```

C++函数模板

- 模板参数可以是类型参数（如 class T），也可以是非类型参数（如 int size）。

```
template <class T, int size>
T max(const T (&arr)[size]) {
    T max_val = arr[0];
    for (int i = 1; i < size; ++i)
        if (arr[i] > max_val)
            max_val = arr[i];
    return max_val;
}
```

C++函数模板

- 函数模板的实例化不需要用户显式进行，而是在函数调用时由编译器来处理。
- 应保证函数调用时的参数与模板参数正好匹配，因为编译器不会为函数模板的参数提供任何形式的转换，需要显式的指定模板实参才能调用。

```
void main()  {
      int a, b; char c, d;
      int m1 = max(a,b);       // 调用max(int a, int b);
      char m2 = max(c,d);      // 调用max(char c,char d);
      int m3 = max(a,c);       // error, 不能生成 max(int, char);
      m3 = max<int>(a,c);      // ok, 显式的指定了模板实参
}
```

C++函数模板

- 函数模板的重载跟普通函数重载一样，也应该保证两个重载的函数模板参数有所不同。
- 编译器在进行函数模板的推演时必须没有二义性。

```
template <class T>  T sum(T *array,int size)  {
      T total;
      for (int i = 0; i < size; i++) total += array[i];
      return total;
}
template <class T> T sum(T *array, T *array2, int size)  {
      T total;
      for (int i = 0; i < size; i++) total + = array1[i] + array2[i];
      return  total;
}
```

C++函数模板

- 当调用函数时，编译器会对于字符串类型使用这个特化的定义，而对其它类型，使用通用的定义。
- 如果模板实参可以从参数类型推导出来，则 显式的类型指示 <CString> 可省略。
- 函数模板的显式特化必须要使用这一类型前先定义或声明，否则编译会报错。

```
typedef const char* Cstring;


template <> Cstring max<Cstring>(Cstring a, Cstring b){
      return strcmp(a, b) >  0 ? a : b;
}
```

◎ C++函数小结

- 对C语言安全性的改进
  - 引用传参 代替 指针传参
  - 内联函数 代替 带参数的宏
  - 名字空间 进行作用域控制，避免重名
- 提高代码的重用性
  - 函数设置多个默认参数
  - 不同参数类型和数量的函数重载
  - 一套代码处理多种类型参数的函数模板