C++编程(5)

唐晓晟
北京邮电大学电信工程学院

第七章 函数

- 函数声明
- 参数传递
- 返回值
- 重载函数名
- 默认参数
- 未确定数目的参数
- 指向函数的指针
- 宏
- 忠告

7.1 函数声明

- 函数是编程人员为了完成某一个任务而编写的可以被重复使用的代码
- 函数只有在预先声明之后才能被调用
- 函数声明中，需要给出函数的名称、返回值类型以及调用该函数时必须提供的参数个数和类型
- 参数传递的语义等同于初始化的语义，必要时，会进行隐式的类型转换
- 函数声明中可包含参数的名字，但是编译器将忽略这样的名字

函数声明示例

```
Elem* next_elem();
char* strcpy(char* to, char* from);
void exit(int);


double sqrt(double);

double sr2 = sqrt(2);
double sr3 = sqrt("three"); // Error


double sqrt(double temperature);
```

7.1.1 函数定义

- 程序中调用的函数必须在某个地方定义(仅仅一次)，函数定义相当于给出了函数体的函数声明
- 函数的定义和对它的所有声明都必须描述了同样的类型，因为参数名字不是类型的一部分，所以参数名不必保持一致
- 函数可以被定义为inline(内联)的

函数定义示例

```
extern void swap(int*, int*); // 声明

void swap(int* p, int* q)  // 定义
{
    int t = *p;
    *p = *q;
    *q = t;
}

inline int fac(int n)
{
    return (n<2) ? 1 : n*fac(n-1);
}
```

7.1.2 静态变量

- 局部变量将在运行线程达到其定义时进行初始化，这种情形发生在函数的每次被调用时，且函数的每次调用都有自己的一份局部变量副本
- 若局部变量被声明为static，那么将只有唯一的一个被静态分配的对象，在该函数的所有调用中该对象唯一，这个对象在该函数第一次被调用时被初始化

静态变量示例

```
void f(int a)
{
    while(a--) {
        static int n = 0; //初始化仅一次
        int x = 0;  //每次f()被调用都初始化
        cout << “n=“<<n++<<“,x=“<<x++<<‘\n’;
    }
}

int main()
{
    f(3);
}
```

输出结果：
n=0,x=0
n=1,x=0
n=2,x=0

7.2 参数传递

- 当一个函数被调用时，将安排好其形式参数所需要的存储，并用实际参数对其进行初始化，必要时需要进行类型转换
- 传递一个引用参数一般意味着要修改这个参数，为了效率原因传递引用参数，可以将其声明为const类型，显式表明将不对其数值进行修改
- 文字量、常量和需要转换的参数都可以传递给const&参数，但不能传递给非const引用

参数传递示例(1)

```
void f(int val, int& ref)
{
    val ++;  ref ++;
}
```

```
void g()
{
    int i = 1;
    int j = 1;
    f(i,j);
    // i不变，j变为2
}
```

```
int strlen(const char*);
// 求C风格的字符串的长度
char* strcpy(char* to, const char* from);
// 复制C风格的字符串
int strcmp(const char*, const char*);
// 比较C风格的字符串
```

参数传递示例(2)

```
float fsqrt(const float&);  // 要求类型是常量引用类型
void g(double d)
{
    float r = fsqrt(2.0f);    // 传递的是保存2.0f临时量的引用
    r = fsqrt(r);    // 传递r的引用
    r = fsqrt(d);    // 传递的是保存float(d)的临时量的引用
}
void update(float& i);  // 要求参数为普通引用类型
void g(double d, float r)
{
    update(2.0f);    // Error: 传递的参数是const
    update(r);       // 传递r的引用
    update(d);       // Error: 要求类型转换，否则update将会
}                    // 更新临时变量float(d)的数值
```

7.2.1 数组参数

- 如果将数组作为函数的参数，传递的就是到数组的首元素的指针，即：类型T[]作为参数传递时将被转换为一个T*
- 对数组参数的某个元素的赋值，将改变实际参数数组中的那个元素的值，也就是说，数组不会(也不能)按传值的方式传递
- 传递数组时，注意数组参数的大小问题

数组参数示例

```
int strlen(const char*);
void f()
{
    char v[] = “an array”;
    int i = strlen(v);
    int j = strlen(“Nicholas”);
}

void compute1(int* vec_ptr, int vec_size); // 一种方法

struct Vec{          void compute2(const Vec& v);
    int* ptr;        // 另一种方法
    int size;
}
```

7.3 返回值

- 没有被声明为void的函数都必须返回一个值，void函数不能有返回值
- 返回值由返回语句(return)描述，一个函数中可以有多个返回语句，返回值的语义和初始化的语义相同
- 不要返回指向局部变量的指针

返回值示例

```
int f1(){ } // Error: 无返回值
void f2() { } // ok
int f3() { return 1; } // ok
void f4() { return 1; } // Error
int f5() { return; } // Error
void f6() { return; } // ok

int fac(int n)
{  return (n>1) ? n*fac(n-1):1;
}// 调用自己的函数称为递归函数

int fac2(int n){
   if(n>1) return n*fac2(n-1);
   return 1; } // 可以有多个返回语句
```

```
double f() { return 1; }
// 1被隐式转换为double(1)

int* fp() { int local = 1;
/* ... */ return &local; }
// Error

int& fr() { int local = 1;
/* ... */ return local; }
// Error

void g(int* p);
void h(int* p) { /* ... */
return g(p); } // ok
```

7.4 重载函数名

- 函数在不同类型的对象上执行概念相同的工作时，可以给多个函数起相同的名字，叫做重载
- 当被重载的函数f被调用时，编译器必须弄清楚应该调用具有名字f的哪一个函数，依据是：实际参数的类型与哪个f函数的形式参数匹配的最好，就调用哪个，如果找不到匹配最好的函数，则给出编译错误
- 重载使得程序员不必为解决某类问题记忆过多的函数名称

编译器的匹配判断原则

- 准确匹配：无需任何转换或者只需做平凡转换(数组名到指针、函数名到函数指针、T到const T等)的匹配
- 利用提升的匹配：即包含整数提升(bool到int、char到int、short到int以及相应的无符号版本)以及从float到double的提升
- 利用标准转换(int到double、double到int、double到long double、Derived*到Base*、T*到void*、int到unsigned int)的匹配
- 利用用户定义转换的匹配
- 利用在函数声明中的省略号的匹配
- 若在匹配的某个层次上同时发现两个匹配，这个调用将被作为歧义而遭拒绝

重载函数名示例

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

void f()
{
    print(1L);
    // print(long)
    print(1.0);
    // print(double)
    print(1);
    // Error:歧义
}
```

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{  print(c); print(i); // 准确匹配
   print(s); // 整数提升：print(int)
   print(f); // float到double提升
   print('a'); print(49); // 准确匹配
   print(0);  // 准确匹配：print(int)
   print("a");
   // 准确匹配：print(const char*) }
```

7.4.1 重载和返回类型

- 重载解析中不考虑返回类型，其理由是：保持对重载的解析只是针对单独的运算符或者函数调用，与调用的环境无关

```
float sqrt(float);
double sqrt(double);
void f(double d, float f)
{
    float fl = sqrt(d);  // sqrt(double)
    double db = sqrt(d); // sqrt(double)
    fl = sqrt(f);  // sqrt(float)
    db = sqrt(f);  // sqrt(float)
}
```

7.4.2 重载与作用域

- 在不同的非名字空间作用域里声明的函数不算是重载

```
void f(int);

void g()
{
    void f(double);
    f(1);  // 调用f(double)
}
```

7.4.3 手工的歧义性解析

- 对一个函数，声明的重载版本过多(或者过少)都有可能导致歧义性
- 只要可能，应该做的就是将该函数的重载版本集合作为一个整体来考虑，有关问题通常可以通过增加一个消除歧义性的版本来解决
- 此外，也可以通过增加一个显式类型转换的方式去解决某个特定调用的问题，但是这样做通常只是权益之计

手工的歧义性解析示例

```
void f1(char);
void f1(long);

void f2(char*);
void f2(int*);

void k(int i)
{
    f1(i);  // 歧义
    f2(0); // 歧义
}
```

解决方案

```
void f1(int n)
{
    f1(long(n));
}

f2(static_cast<int*>(0));
```

7.4.4 多参数的解析

- 基于上述重载解析规则，可以保证：当所涉及到的不同类型在计算效率或者精度方面存在明显差异时，被调用的将会是最简单的算法(函数)

多参数的解析示例

```
int pow(int,int);
double pow(double,double);
complex pow(double,complex);
complex pow(complex,int);
complex pow(complex,double);
complex pow(complex,complex);
void k(complex z)
{
   int i = pow(2,2);  // pow(int,int)
   double d = pow(2.0,2.0); // pow(double,double)
   complex z2 = pow(2,z);  // pow(double,complex)
   complex z3 = pow(z,2);  // pow(complex,int)
   complex z4 = pow(z,z);  // pow(complex,complex)
}
```

注意：
double d = pow(2.0,2);
将会引起歧义

7.5 默认参数

- 一个通用函数所需要的参数常常比处理简单情况时所需要的参数更多一些
- 默认参数的使用可以为编程增加灵活性
- 默认参数的类型将在函数声明时检查，在调用时求值
- 只能对排列在最后的那些参数提供默认参数

默认参数示例

```
void print(int value,
        int base = 10);
void f() {
    print(31);
    print(31,10);
    print(31,16);
    print(32,2);
}
```

输出：31 31 1f 11111

替代方案：

```
void print(int value, int base);
void print(int value){ print(value,10);}
```

但是，读者不容易看出原来的意图：一个函数加上一种简写形式

声明

```
int f(int,int =0, char* =0); // ok
int g(int =0, int =0, char*);//Error
int h(int =0,int, char* =0);//Error
int nasty(char*=0); //Syntax Error

void f(int x = 7);
void f(int = 7); // Error:参数重复
void f(int = 8); // Error:参数值改变
```

```
void g()
{
    void f(int x=9);
    // ok
}
```

7.6 未确定数目的参数

- 有些函数，无法确定在各个调用中所期望的所有参数的个数和类型，声明这种函数的方式就是在参数表的最后用省略号(...)结束，表示还可能有另外一些参数
- 对于省略号省略的参数，编译器在编译时无法对其进行参数检查，即：可能编译通过，但是运行出错，C++中可以通过重载函数和使用默认参数避免此类问题
- <cstdarg>里提供了一组标准宏，专门用于在这种函数里访问未加描述的参数

未确定数目的参数示例

```
int printf(const char* ...);

printf("Hello, world!\n");
printf("My name is %s %s\n", first_name, last_name);
printf("%d + %d = %d\n", 2,3,5);

#include <stdio.h>
int main()
{
    printf("My name is %s %s\n",2);
} // 最好情况下，会有一些很奇怪的输出

int fprintf(FILE*, const char* ...);
int execl(const char* ...);
```

请参看书中p139，error函数实现

7.7 指向函数的指针

- 对一个函数只能做两件事：调用它或者取得它的地址
- 通过取一个函数的地址而得到的指针即为函数指针，可以在后面用来调用这个函数
- 在指向函数的指针的声明中需要给出参数类型，就像函数声明一样，在指针赋值时，完整的函数类型必须完全匹配

函数指针示例

```
void error(string s) { /* ... */ }
void (*efct)(string); //函数指针
void f()
{
    efct = &error; // efct指向error
    efct("error");  // 调用error
}              // 不需要间接运算符*

void (*f1)(string) = &error;//ok
void (*f2)(string) = error; // ok

void g() {
    f1("Vasa");  // ok
    (*f2)("Mary Rose"); //ok }
```

```
void (*pf)(string);
void f1(string);
int f2(string);
void f3(int*);

void f() {
    pf = &f1; // ok
    pf = &f2;
    // Error:返回值不匹配
    pf = &f3; // Error
    pf("Hera"); // ok
    pf(1); // Error
    int i = pf("Zeus"); }
    // Error: void赋值给int
```

其他函数指针示例

```
// 摘自<signal.h>
typedef void (*SIG_TYP)(int); //对比 typedef char* PCH;
typedef void (*SIG_ARG_TYP)(int);
SIG_TYP signal(int,SIG_ARG_TYP);

typedef void (*PF)();
PF edit_ops[] = { &cut, &paste, &copy, &search };
PF file_ops[] = { &open, &append, &close, &write };

PF* button2 = edit_ops;
PF* button3 = file_ops;

button2[2]; //调用按钮2的第3个函数
```

请参看书中p141, ssort函数的实现，以及p142中对其的应用举例

函数指针示例

- 关于重载函数的指针，可以通过赋值或者初始化指向函数的指针方式，取得一个重载函数的地址
- 通过指向函数的指针调用的函数，其参数类型和返回值类型必须与指针的要求完全一致
- 当用函数对指针赋值或初始化时，没有隐含的参数或者返回值类型转换

```
void f(int);
int f(char);

void (*pf1)(int) = &f; // void f(int)
int (*pf2)(char) = &f; // int f(char)
void (*pf3)(char) = &f; // Error
```

7.8 宏

- 第一规则：绝不应该去使用它，除非你不得不这么做
- 几乎每个宏都表明了程序设计语言中、或者程序里、或者程序员的一个缺陷
- 宏使得编译器在真正处理正文之前需要进行预处理操作，编译器看到的是宏展开后的结果，可能导致非常难以理解的错误

关于宏使用中的注意事项

- 宏预处理器不能处理递归的宏
- 宏的名字不能重载
- 在宏中，引用全局名字时一定要使用作用域解析运算符::，并在所有可能的地方将出现的宏参数都用括号围起来
- 宏中的注释请使用/* */方式
- 通过##宏运算符可以拼接起两个串，构造出一个新串

宏的示例(1)

```
#define NAME rest of line
named = NAME
// named = rest of line

#define MAC(x,y) arg:x arg:y
expanded = MAC(foo bar, yuk yuk)
// expanded = arg: foo bar arg: yuk yuk

#define PRINT(a,b) cout << (a) << (b)
#define PRINT(a,b,c) cout << (a) << (b)<< (c) // Error
#define FAC(n) (n>1)?n*FAC(n-1):1 // Error

#define MIN(a,b) (((a) < (b)) ? (a) : (b) )
#define M2(a) something(a) /* 细心的注释 */
```

宏的示例(2)

```
可能有用的宏 #define FOREVER for(;;)
没有必要的宏 #define PI 3.14159
很危险的宏    #define SQUAR(a) a*a
    // int xx = 0; int y = square(xx+2);

#define NAME2(a,b) a##b
int NAME2(hack,cah)();
// 将产生 hackcah();

#undef X
//保证不再有称为X的有定义的宏
```

7.8.1 条件编译

- 根据条件决定某一段代码是否让编译器编译
- #ifdef identifier将条件性地导致随后的输入被忽略，直到遇到一个#endif指令

```
int f(int a
#ifdef arg_two
, int b
#endif
);
```

将产生 int f(int a
); //除非宏arg_two存在

7.9 忠告

- [1]质疑那些非const的引用参数；如果你想要一个函数去修改其参数，请使用指针或者返回值
- [2]当你需要尽可能减少参数复制时，应该使用const引用参数
- [3]广泛而一致地使用const
- [4]避免宏
- [5]避免不确定数目的参数
- [6]不要返回局部变量的指针或者引用

忠告

- [7]当一些函数对不同的类型执行概念上相同的工作时，请使用重载
- [8]在各种整数上重载时，通过提供函数去消除常见的歧义性
- [9]在考虑使用指向函数的指针时，请考虑虚函数或模板是不是更好的选择
- [10]如果你必须使用宏，请使用带有许多大写字母的丑陋的名字