

函数

 C++ Primer第五版

 第6章

11

函数基础

参数传递

返回类型 和 `return`

函数重载

特殊用途语言特性
`constexpr` `inline` `assert`

函数匹配

函数指针

函数基础

函数是一个命名了的代码块，通过调用函数执行相应的代码。可以有0个或多个参数，可以重载。

//编写一个求数的阶乘的程序

```
int fact(int val){
    int ret = 1; //局部变量，用于保存计算结果
    while(val>1)
        ret *= val--;
    return ret;
}
```

函数的调用：

- 用实参初始化函数对于的形参（类型、个数需要匹配）
- 主函数暂时中断，被调函数开始执行

```
int main(){
    int j = fact(5); //j等于120，即fact(5)的结果
    cout << "5! is " << j << endl;
    return 0;
}
```

局部对象

- 自动对象：生命周期从变量声明开始，到函数块末尾结束
- 局部静态对象：生命周期从变量声明开始，直到程序结束才销毁

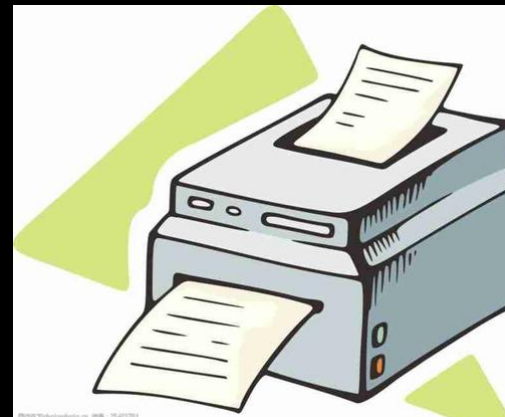
```
size_t count_calls()
{
    static size_t ctr = 0; //调用结束后，这个值仍然有效
    return ++ctr;
}
```

函数的声明 函数名 形参 ； 不需要{ }

```
int main()
{
    for ( size_t i = 0; i != 10; ++i)
        cout<<count_calls()<<endl;
    return 0;
}
```

形参不改变实参

```
//该函数接受一个指针，然后将指针所指的值置为0
void reset (int *ip) { //指针形参
    *ip = 0; //改变指针ip所指对象的值
    ip = 0; //只改变了ip的局部拷贝，实参未被改变
}
int main(){
    int i=42;
    reset(&i); //改变i的值而非i的地址
    cout<<"i="<<i<<endl; //输出i = 0
    return 0;
}
```



传引用参数

```
//该函数接受一个int对象的引用，然后将对象指的值置为0
void reset (int &i) //i是传给reset函数的对象的另一个名字{
    i = 0; //改变了i所引用对象的值
}
int main(){
    int j=42;
    reset(j); //j采用传引用方式，它的值被改变
    cout<<"j="<<j<<endl; //输出j = 0
    return 0;
}
```



使用引用避免拷贝

```
//比较两个string对象的长度
bool isShroter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

如果函数无需改变引用形参的值，最好将其声明为常量引用。

使用引用形参返回额外信息

```
//返回s中c第一次出现的位置索引，引用参数occurs赋值统计c出现的总次数
string::size_type find_char(const string&s, char c, string::size_type &occurs){
    auto ret = s.size(); //第一次出现的位置（如果有的话）
    occurs = 0;
    for(decType(ret) i=0; i!=s.size(); ++i){
        if(s[i]==c){
            if(ret == s.size()) ret = i; //记录第一次出现的位置
            ++occurs; //将出现的次数加1
        }
    }
}
```

const形参和实参

形参的顶层const会被忽略掉

```
void fcn(const int i){/*fcn能够读取i，但不能向i写值*/}
void fcn(int i){ } //错误：重复定义
```

指针或引用形参与const

```
void reset (int *ip) { *ip = 0; ip = 0; }
void reset (int &ip) { ip = 0; }

//返回s中c第一次出现的位置索引，引用参数occurs赋值统计c出现的总次数
string::size_type find_char(const string&s,char c,string::size_type &occurs){
    auto ret = s.size(); //第一次出现的位置（如果有的话）
    occurs = 0;
    for(decType(ret) i=0;i!=s.size();++i){
        if(s[i]==c){
            if(ret == s.size()) ret = i; //记录第一次出现的位置
            ++occurs; //将出现的次数加1
        }
    }
}

int main(){
    int i = 0;
    const int ci = i;
    string::size_type ctr = 0;

    reset(&i); //调用形参类型是int*的reset函数
    reset(&ci); //错误：不能用指向const int对象的指针初始化int*
    reset(i); //调用形参类型是int&的reset函数
    reset(ci); //错误：不能把普通引用绑定到const对象ci上
    reset(42); //错误：不能把普通引用绑定到字面值上
    reset(ctr); //错误：类型不匹配，ctr是无符号类型
    //正确:find_char的第一个形参是对常量的引用
    find_char("Hello World !",'o',ctr);
}
```

如果函数无需改变引用形参的值，最好将其声明为常量引用。

数组形参

```
//尽管形式不同，但这三个print函数是等价的
//每个函数都有一个const int*类型的形参
void print(const int*);
void print(const int[]);
void print(const int[10]); //这里的10只是一个期望
```

以数组作为形参的函数必须确保使用数组时不会越界

```
//1、利用数组本身的介绍符
void print(const char *cp){
    if(cp) //若cp不是一个空指针
        while(*cp) //取出来的不是空字符
            cout<<*cp++; //输出当前字符，并将指针移动到下一个位置
}
//2、使用标准库规范
void print(const int *beg, const int *end){
    //输出beg到end之间(不含end)的所有元素
    while(beg!=end)
        cout<<*beg++<<endl;
}
int j[2] = {0,1};
print(begin(j),end(j));
//3、显式传递一个表示数组大小的形参
//const int ia[]等价于const int* ia
//size 表示数组的大小
void print(const int ia[], size_t size){
    for(size_t i = 0; i!=size; ++i)
        cout<<ia[i]<<endl;
} //可以使用print(j,end(j)-begin(j));进行调用
```

C++允许将变量定义成数组的引用

```
//正确：形参是数组的引用，维度是类型的一部分
void print(int (&arr)[10]){ //（）不能少
    for(auto elem:arr)
        cout<<elem<<endl;
}
int i=0, j[2]={0,1}, k[10]={0,1,2,3,4,5,6,7,8,9};
print(&i); //错误：实参不是含有10个整数的数组
print(j); //错误：实参不是含有10个整数的数组
print(k); //正确
```

传递多维数组：在C++中没有实际的多维数组

```
void print(int (*matrix)[10], int rowSize){ };
void print(int matrix[][10], int rowSize){ }; //实际上是指向10个整数的数组的指针
```

main：处理命令行选项

```
int main(int argc,char *argv[]){ }
int main(int argc,char **argv){ }

//如果运行prog -d -o ofile data0
//argc = 5
//argv[0] = "prog"
//...
//argv[4] = "data0"
//argvp[5] = 0 //最后一个指针之后的元素值保证为0
```

含有可变形参的函数：参数个数不固定
如果所有的实参类型相同，可以传递一个名为**initializer_list**的标准库类型

Initializer_list提供的操作	
initializer_list<T> lst;	默认初始化：T类型元素的空列表
initializer_list<T> lst{a,b,c,...};	list的元素数量和初始值一样多；lst的元素是对应初始值的副本；列表中的元素是const
list2(list) lst2=lst	拷贝或赋值一个initializer_list对象不会拷贝列表中的元素；拷贝后，原始列表和副本共享元素
lst.size()	列表中的元素数量
lst.begin()	返回指向lst中首元素的指针
lst.end()	返回指向lst中尾元素下一位置的指针

```
//和vector一样，initializer_list也是一种模板类型
initializer_list<string> ls; //元素类型是string
initializer_list<int> li; //元素类型是int
//和vector不一样的是，initializer_list对象中的元素永远是常量值
void error_msg(initializer_list<string> il){
    for(auto beg = il.begin(); beg!=il.end();++beg)
        cout<<*beg<<" ";
    cout<<endl;
}

//excepted和actual是string对象
if(expected != actual)
    error_msg({"functionX",expected,actual});
else
    error_mesg({"functionX","okey"});
```



```
void swap(int &v1, int &v2)
{
    //如果两个值是相等的，则不需要交换，直接退出
    if(v1 == v2)
        return;
    //如果程序执行到这里，说明还需要继续完成某些功能
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    //此处无需显示的return语句
}
```

有返回值函数

```
//因为含有不正确的返回值，所以这段代码无法通过编译
bool str_subrange(const string &str1, const string &str2)
{
    //大小相同：此时通过普通的相等性判断结果作为返回值
    if(str1.size() == str2.size())
        return str1 == str2; //正确：==运算符返回布尔值
    //得到较短string对象的大小
    auto size = (str1.size() < str2.size()) ? str1.size() : str2.size();
    //检查两个string对象的对应字符是否相等，以较短的字符串长度为限
    for(decType(size) i=0; i!=size; ++i){
        if(str1[i] != str2[i]) return; //错误#1：没有返回值,编译器报错
    }
    //错误#2：控制流可能尚未返回任何值就结束了函数的执行
    //编译器可能检查不出来这一错误
}
```

值是如何被返回的

```
//如果ctr的值大于1，返回word的复数形式
string make_plural(size_t ctr, const string &word, const string &ending)
{
    return (ctr > 1) ? word + ending : word;
}
```

返回word对象的副本，或一个没命名的临时string对象

```
//挑出两个string对象中较短的那个，返回其引用
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

这里不会真正拷贝string对象

不要返回局部对象的引用或指针

```
//严重错误：这个函数试图返回局部对象的引用
const string &manip(){
    string ret;
    //以某种方式改变一下ret
    if(!ret.empty())
        return ret;//错误：返回局部对象的引用！
    else
        return "Empty";//错误：“Empty”是一个局部临时变量
}
```

返回类类型的函数和调用运算符

```
//调用string对象的size成员，该string对象是由函数返回的
auto sz = shorterString(s1,s2).size();
```

引用返回左值

```
char &get_val(string &str, string::size_type ix){
    return str[ix]; //get_val嘉定索引值是有效的
}
int main(){
    string s("a value");
    cout<<s<<endl; //输出a value
    get_val(s,0)='A'; //将s[0]的值改为A
    cout<<s<<endl; //输出A value
    return 0 ;
}
```

如果返回类型是常量引用，则不可以给调用的结果赋值

列表初始化返回值：C++11

```
vector<string> process(){
    //...
    //expected和actual是string对象
    if(expected.empty())
        return{};//返回一个空vector对象
    else if (expected == actual)
        return{"functionX","okay"}; //返回列表初始化的vector对象
    else
        return{"functionX",expected,actual};
}
```


主函数main的返回值

```
int main()
{
    if(some_failure)
        return EXIT_FAILURE; //定义在cstdlib头文件中
    else
        return EXIT_SUCCESS; //定义在cstdlib头文件中

    //编译器将隐式的插入一条返回0的return语句。
}
```

递归：函数调用了自己

```
//计算val的阶乘
int factorial(int val){
    if(val>1)
        return factorial(val-1)* val;

    return 1;
}
```

main函数不能调用它自己



返回数组指针：数组不能拷贝，所以函数不能直接返回数组。

```
typedef int arrT[10]; //arrT是一个类型别名,using arrT = int[10];
arrT* func(int i); //func返回一个指向含有10个整数的数组的指针
```

为了简化，可以使用类型别名

```
//声明一个返回数组指针的函数
//返回数组指针的函数形式如下所示：
//Type (*function(parameter_list))[dimension]
int (*func(int i))[10];
```

- `func(int i)`表示调用func函数时需要一个int型的实参
- `(*func(int i))`意味着我们可以对函数调用的结果执行解引用操作
- `(*func(int i))[10]`表示解引用func将得到一个大小为10的数组
- `int (*func(int i))[10]`表示数组的大小为10

//使用尾置返回类型，C++ 11

//->符号开始，在本应该出现类型的地方使用auto

//func接受一个int型的实参，返回一个指针，该指针指向含有10个整数的数组

```
auto func(int i) -> int(*) [10];
```

//使用decltype:当我们知道函数返回的数组将指向哪个数组

```
int odd[] = {1,3,5,7,9};
```

```
int even[] = {0,2,4,6,8};
```

//返回一个指针，该指针指向含有5个整数的数组

```
decltype(odd) *arrPtr(int i)
```

```
{
```

```
    return (i%2)?&odd:&even;//返回一个指向数组的指针
```

```
}
```

函数重载：函数名称相同但形参列表不同

定义重载函数

```
Record lookup(const Account&);
bool lookup(const Account&); // 错误
Record lookup(const Phone&);
Record lookup(const Name&);
Account acct;
Phone phone;
Record r1 = lookup(acct);
Record r2 = lookup(phone);
```

const_cast和重载

```
// 比较两个string对象的长度，返回较短的那个引用
const string& shorterString(const string& s1, const string& s2){
    return s1.size() <= s2.size() ? s1 : s2;
}
string& shorterString(string& s1, string& s2){
    auto& r = shorterString(<const_cast<const string&>(s1),
                           <const_cast<const string&>(s2));
    return const_cast<string&>(r);
}
```

强制转换不能省略，否则就成了递归
(最佳匹配原则)

重载与作用域

```
string read();
void print(const string&);
void print(double); // 重载print函数

void fooBar(int ival) {
    bool read = false; // 隐藏了外层的read
    string s = read(); // 错误：read是一个布尔值
    // 不好的习惯
    void print(int); // 隐藏了外层的print
    print("Value:"); // 错误
    print(ival); // 正确
    print(3.14); // 正确：等价于print(3)
}
```

一旦在当前作用域中找到了所需的名字，编译器就会忽略掉外层作用域中的同名实体

特殊用途语言特性：

默认实参

一旦某个形参被赋予了默认值，它后面的所有形参都必须有默认值

```
typedef string::size_type sz;
string screen(sz ht = 24, sz wid = 80, char backgrnd = ' ');
```

```
string window;
window = screen(); //等价于screen(24,80,' ')
window = screen(66); //等价于screen(66,80,' ')
window = screen(66, 256); //screen(66,256,' ')
window = screen(66,256, '#'); //screen(66,256, '#')
```


```
window = screen( , , '?'); //错误：只能省略尾部的实参
window = screen( '?'); //调用screen('?',80,' ')
```

```
//多次声明同一个函数也是合法的
string screen2( sz, sz, char = ' ');
string screen2( sz, sz, char = '*' );//错误：重复声明
string screen2(sz = 24, sz = 80, char );//正确
```

```
//局部变量不能作为默认实参。
//除此之外，只要表达式的类型能转换成形参所需的类型，就能作为默认实参
//wd,def和ht的声明必须出现在函数之外
sz wd = 80;
char def = ' ';
sz ht();
string screen(sz = ht(), sz = wd, char = def);
string window = screen(); //调用screen(ht(),80,' ')
```

```
void f2()
{
    def = '*'; //改变默认实参的值
    sz wd = 100; //隐藏那个了外层定义的wd，但是没有改变默认值
    window = screen(); //调用screen(ht(), 80, '*')
}
```

内联函数：

@阿西拜-南昌 

在每个调用点上“内联地”展开，避免函数调用的开销

```
//比较两个string对象的长度，返回较短的那个引用
inline const string& shorterString(const string& s1, const string& s2){
    return s1.size() <= s2.size() ? s1 : s2;
}

cout<<shorterString(s1,s2)<<endl;
//在编译过程中展开成类似于下面的形式
cout<<( s1.size()<s2.size()?s1:s2 )<<endl;
//一般来说，内联机制用于优化规模较小，流程直接、频繁调用的函数
```

内联说明只是向编译器发出的一个请求，编译器可以选择忽略这个请求

constexpr函数：

能用于常量表达式的函数:函数的返回类型以及所有的形参都是字面值类型

```
constexpr int new_sz() { return 42;}
constexpr int foo = new_sz(); //正确：foo是一个常量表达式


//如果arg是常量表达式，则scale(arg)也是常量表达式
constexpr size_t scale(size_t cnt) {return new_sz() * cnt;}

int arr[scale(2)]; //正确：scale(2)是常量表达式
int i = 2;
int a2[scale(i)]; //错误：scale(i)不是常量表达式
```

函数体中必须有且只有一条return语句

constexpr函数被隐式地指定为内联函数

调试帮助：只在开发过程中使用的代码，发布时屏蔽掉

@阿西拜-南昌 

assert 预处理宏：cassert头文件中

```
//如果表达式为假，assert输出信息并终止程序的执行
//如果表达式为真，assert什么也不做
assert(word.size()>threshold);
```

用于检测“不能发生”的条件

NDEBUG预处理变量：

assert的行为依赖NDEBUG预处理变量的状态，如果定义了NDEBUG,则assert无效

```
#define NDEBUG//关闭调试状态，必须在cassert头文件上面
#include <cassert>

int main( void )
{
    int x = 0;
    assert(x);
}
```

除了用于assert外，也可以使用NDEBUG编写自己的条件调试代码

```
void print( const int ia[], size_t size)
{
#ifdef NDEBUG
    //__func__是编译器定义的一个局部静态变量，用于存放函数的名字
    cerr<< __func__ >><<":array size is "<<size <<endl;
#endif
    //...
    __func__为const char 的一个静态数组，即“print”
}
```

```
//除了C++编译器定义的__func__之外
//预处理器还定义了另外4个对于程序调试很有用的名字
//1、__FILE__：存放文件名的字符串字面值。
//2、__LINE__：存放当前行号的整形字面值。
//3、__TIME__：存放文件编译时间的字符串字面值。
//4、__DATE__：存放文件编译日期的字符串字面值。
```

```
if(word.size()<threshold)
    cerr<<"Error:"<<__FILE__
    <<":in function"<<__func__
    <<" at line "<<__LINE__<<endl
    <<"    Comiled on "<<__DATE__
    <<"    Word read was \""<<word
    <<"\":Length too short"<<endl;
```

```
Error:wdebug.cc : in function main at line 27
  Compiled on Jul 11 2012 at 20:50:03
  Word read was "foo": Length too short
```




```
void f();  
void f(int);  
void f(int, int);  
void f(double, double=3.14);
```

```
f(5.6); //调用 void f(double, double)  
f(42,2.56); //错误，具有二义性
```

- 1、先确定“候选人”
- 2、寻找最佳匹配
- 3、如果有二义性编译器将拒绝请求

编译器将实参类型到形参类型的转换划分成几个等级：

1. 精确匹配
2. 通过const转换实现的匹配
3. 通过类型提升实现的匹配
4. 通过算术类型转换实现的匹配
5. 通过类类型转换实现的匹配

```
void ff(int);  
void ff(short);  
ff('a'); //char 提升成int;调用f(int)
```

```
void manip(long);  
void manip(float);  
manip(3.14); //错误：二义性
```

所有算术类型转换的级别都一样

```
Record lookup(Account&);  
Record lookup(const Account&);  
const Account a;  
Account b;
```

```
lookup(a); //调用 lookup(const Account&);  
lookup(b); //调用 lookup(Account&);
```



```
//比较两个string对象的长度
bool lengthCompare(const string &, const string &);
//该函数的类型是bool (const string &, const string &)
//声明一个可以指向该类型函数的指针，只要用指针替换函数名即可
bool (*pf)(const string &, const string &); //括号不能少

pf = lengthCompare;
pf = &lengthCompare; //等价的赋值语句：取地址符是可选的
//可以直接使用指针函数的指针调用该函数，无需提前解引用
bool b1 = pf("hello", "goodbye");
bool b2 = (*pf)("hello", "goodbye"); //等价的调用
bool b3 = lengthCompare("hello", "goodbye");

string::size_type sumLength(const string&, const string&);
bool cstringCompare(const char*, const char*);
pf = 0; //正确：pf不指向任何函数
pf = sumLength; //错误：返回类型不匹配
pf = cstringCompare; //错误：形参类型不匹配
pf = lengthCompare; //正确：函数和指针的类型精确匹配
```

在指向不同函数类型的
指针间不存在转换规则

重载函数的指针

```
void ff(int *);
void ff(unsigned int);

void (*pf1)(unsigned int) = ff; //pf1指向ff(unsigned)
void (*pf2)(int) = ff; //错误：没有任何一个ff与该形参列表匹配
double (*pf3)(int *) = ff; //错误：ff和pf3的返回类型不匹配
```

必须精确匹配



//第三个形参是函数类型，它会自动地转换成指向函数的指针

```
void useBigger(const string &s1, const string &s2,
              bool pf(const string &, const string &));
```

//等价的声明：显式地将形参定义成指向函数的指针

```
void useBigger(const string &s1, const string &s2,
              bool (*pf)(const string &, const string &));
```

//可以直接把函数作为实参使用，会制动转换成指针
useBigger(s1,s2,lengthCompare);

不能定义函数类型的形参，但是形参可以是指向函数的指针

//通过使用类型别名，简化使用函数指针

//Func和Func2是函数类型

```
typedef bool Func(const string&, const string&);
```

```
typedef decltype(lengthCompare) Func2; //等价的类型
```

//FuncP和FuncP2是指向函数的指针

```
typedef bool(*FuncP)(const string&,const string&);
```

```
typedef decltype(lengthCompare) *FuncP2; //等价的类型
```

//useBigger的等价声明，其中使用了类型别名

```
void useBigger(const string&, const string&,Func);
```

```
void useBigger(const string&, const string&,FuncP2);
```

返回指向函数的指针

```
using F = int(int*, int); //F是函数类型，不是指针
```

```
using PF = int (*)(int*, int); //PF是指针类型
```

```
PF f1(int); //正确：PF是指向函数的指针，f1返回指向函数的指针
```

```
F f1(int); //错误：F是函数类型，f1不能返回一个函数
```

```
F *f1(int); //正确：显示地指定返回类型是指向函数的指针
```

//当然，我们也能用心的形式直接声明

```
int (*f1(int))(int*, int);
```

不能返回函数，但可以返回指向函数的指针（和函数类型的形参不一样，返回类型必须写成指针形式）

//使用尾置返回的方式

```
auto f1(int) -> int (*)(int*,int);
```

用auto和decltype用于函数指针类型

```
string::size_type sumLength(const string&, const string&);
```

```
string::size_type largeLength(const string&, const string&);
```

//根据getFcn形参的取值，返回sumLength或largeLength

```
decltype(sumLength) *getFcn(const string &);
```