C++ 程序设计

第6章 函数

内容

函数

- 6.1 函数基础
- 6.2 参数传递
- 6.3 返回类型和return语句
- 6.4 函数重载
- 6.5 特殊用途语言特性
- 6.6 函数匹配
- 6.7 函数指针

函数基础

- 函数(function)是命名了的代码块
- 函数定义包括：
  - 返回类型(return type)
  - 函数名
  - 形参(parameter)列表
    - ✓形参以逗号隔开，位于一对圆括号内
  - 函数体
    - ✓语句块

编写函数

函数基础

◆求阶乘n!的函数

```
//  val * (val - 1) * (val - 2) . . . * ((val -(val - 1)) * 1)
int fact(int val)
{
    int ret = 1;

     while (val > 1)

        ret *= val--;  // assign ret*val to ret and decrement val
    return ret;        // return the result
}
```

函数调用

函数基础

- 通过调用运算符(call operator)—圆括号来执行函数：

  函数名 (实参列表)

  - 调用运算符作用于函数或函数指针
  - 圆括号内是逗号分隔的实参(argument)列表
    - ✓实参用来初始化函数的形参
  - 调用表达式的类型就是函数的返回类型

调用函数

函数基础

- 函数的调用完成两项工作：
  - 隐式地定义并用实参初始化函数对应的形参
  - 将控制权从主调函数(calling function)转移动到被调函数(called function)
- 当遇到return语句时函数结束执行过程
- return语句完成两项工作：
  - 返回return语句中的值（如果有的话）
    - ✓函数返回值用于初始化调用表达式的结果
  - 将控制权从被调函数转移回主调函数

调用函数

函数基础

```
int fact(int val);   // factorial of  val
int main()
{
    int j = fact(5);  // j equals 120, i.e., the result of fact(5)
    cout << "5! is " << j << endl;
    return 0;
}
```

◆ int j = fact(5);等价于：

```
int val = 5;       // initialize val from the literal 5
int ret = 1;       // code from the body of fact
while (val > 1)
    ret *= val--;
int j = ret;       // initialize j as a copy of ret
```

形参和实参

函数基础

- 实参是形参的初始值
- 函数有几个形参，就必须提供相同数量的实参
  - 没有规定实参的求值顺序
- 实参的类型必须与对应的形参类型相匹配

```
int fact(int val);    // factorial of  val
fact("hello");       // error: wrong argument type
fact();              // error: too few arguments
fact(42, 10, 0);     // error: too many arguments
fact(3.14);          // ok: argument is converted to int
```

函数的形参列表

函数基础

- 形参列表可以为空，但是不能省略
  - 与C兼容，可用关键字void表示没有形参

```
void f1(){ /* ... */ }     // implicit void parameter list
void f2(void){ /* ... */ } // explicit void parameter list
```

- 形参以逗号隔开，每个形参都是含有一个声明符的声明

```
int f3(int v1, v2) { /* ... */ }     // error
int f4(int v1, int v2) { /* ... */ } // ok
```

- 任意两个形参不能同名，且不能与函数内最外层的局部变量同名
  - 形参名可选，但无法使用未命名的形参
  - 即使函数内不使用某个形参，也必须提供实参

函数返回类型

函数基础

- 大多数类型都能作为函数的返回类型
- 特殊类型void表示函数不返回任何值
- 返回类型不能是数组类型或函数类型
  - 可以是指向数组或函数的指针

练习

◆练习6.2：指出并修改下列函数的错误

```
(a) int f() {
      string s;
      // ...
      return s;
  }
(b) f2(int i) { /* ... */ }
(c) int calc(int v1, int v1) {/* ... */ }
(d) double square(double x) return x * x;
```

局部对象

函数基础

- 名字的作用域(scope)是程序文本的一部分，名字在其中可见
- 对象的生存期(lifetime)是程序执行过程中该对象存在的时间
- 形参和函数体内部定义的变量统称为局部变量(local variable)
  - 在函数作用域内可见
  - 隐藏(hide)外层作用域中相同名字的声明
  - 局部变量的生存期依赖于定义的方式
    - ✓在所有函数体外定义的全局对象存在于程序的整个执行过程中

自动对象

局部对象

- 只存在于块执行期间的对象称为自动对象(automatic object)
- 形参是自动对象
  - 函数开始时为形参分配存储空间；函数终止时，形参被销毁
  - 用传递给函数的实参初始化形参对应的自动对象
- 对于局部变量对应的自动变量，若定义时没有初始值，执行默认初始化
  - 内置类型的未初始化局部变量值未定义

局部静态对象

局部对象

- 将局部变量定义为static类型得到一个局部静态对象(local static object)
  - 执行第一次经过对象定义语句时初始化，直到程序终止才被销毁int
  - 如果没有显式初始化，执行值初始化
    - ✓ 内置类型的未初始化局部静态变量初始化为0

```
int count_calls(){

    static size_t ctr = 0;
    // value will persist across calls
    return ++ctr;
}
int main()
{
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;
    return 0;
}
```

函数声明

函数基础

- 和其他名字一样，函数名必须在使用之前声明
- 类似于变量，函数只能定义一次，但可以声明多次
- 函数声明和函数定义类似，区别是无需函数体，用一个分号代替
  - 函数声明中，经常省略形参的名字
  - 函数声明也称作函数原型(function prototype)
- 函数在头文件中声明，在源文件中定义
  - 定义函数的源文件应包含声明函数的头文件

参数传递

- 形参初始化的机理与变量初始化一样
- 当形参是引用类型时，它将绑定到对应的实参。称对应的实参被引用传递(passed by reference)或者函数被传引用调用(called by reference)
  - 引用形参是它对应实参的别名
- 当形参不是引用类型时，实参的值被拷贝给形参，我们说实参被值传递(passed by value)或函数被传值调用(called by value)
  - 形参和实参是两个相互独立的对象

传值参数

参数传递

- 对于传值参数，函数对形参做的所有操作都不会影响实参
  - 对指针形参，可通过形参指针修改它所指对象的值，但不会影响实参指针

```
// function that takes a pointer and sets the pointed-to value to zero
void reset(int *ip)
{
    *ip = 0;  // changes the value of the object to which ip points
    ip = 0;   // changes only the local copy of ip; the argument is
unchanged
}
int i = 42;
reset(&i);                  // changes i but not the address of i
cout << "i = " << i << endl;  // prints i = 0
```

传引用参数

参数传递

- 引用形参绑定初始化它的实参对象
- 通过使用引用形参，函数可以改变一个或多个实参的值

```
// function that takes a reference to an int and sets the given object to zero
void reset(int &i)  // i is just another name for the object passed to reset
{
    i = 0;  // changes the value of the object to which i refers
}
```

- 调用时，直接传入对象：

```
int j = 42;
reset(j);  // j is passed by reference; the value in j is changed
cout << "j = " << j  << endl;  // prints j = 0
```

使用引用避免拷贝

传引用参数

- 传值时拷贝大的类类型或容器对象是很低效的，还有些类类型不支持拷贝
- 此时，函数只能通过引用形参访问该类型的对象

```
// compare the length of two strings
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

使用引用参数返回额外信息

传引用参数

- 一个函数只能返回一个值，通过引用形参，函数可以同时返回多个值

```
// returns the index of the first occurrence of c in s
// the reference parameter occurs counts how often c occurs
int find_char(const string &s, char c, int  &occurs)
{

    int ret = s.size();   // position of the first occurrence, if any
    occurs = 0;           // set the occurrence count parameter for
    (int i = 0; i != s.size(); ++i) {
        if (s[i] == c) {
            if (ret == s.size())
                ret = i;   // remember the first occurrence of c
            ++occurs;      // increment the occurrence count
        }
    }
    return ret;           // count is returned implicitly in occurs
}
```

const形参和实参

参数传递

- 顶层const作用于对象本身
- 实参初始化形参时，形参的顶层const被忽略
  - 当形参有顶层const时，可传给它常量或非常量对象

```
void fcn(const int i) { /* fcn can read but not write to i */ }
void fcn(int i) { /* . . . */ } // error: redefines fcn(int)
```

指针或引用形参与const

const形参和实参

- 可用非const对象初始化一个底层const对象，但反过来不行
- 普通引用必须用同类型的对象初始化

```
void reset(int *ip);
void reset(int &i);
int i = 0;
const int ci = i;
reset(&i);   // calls the version of reset that has an int* parameter
reset(&ci);  // error: can't initialize an int* from a pointer to a const int object
reset(i);    // calls the version of reset that has an int& parameter
reset(ci);  // error: can't bind a plain reference to the const object ci
```

指针或引用形参与const

const形参和实参

```
void reset(int *ip);
void reset(int &i);
```

```
reset(42);   // error: can't bind a plain reference to a literal
```

```
reset(ctr);  // error: types don't match; ctr has an unsigned type
```

◆ 允许用字面值初始化常量引用

```
int ctr = 0;
int find_char(const string &s, char c, int &occurs);
// ok: find_char's first parameter is a reference to const
find_char("Hello World!", 'o', ctr);
```

尽量使用常量引用

const形参和实参

- 如果函数不会改变实参的值，那么形参尽量定义为常量引用类型
  - 普通引用限制了函数能接受的实参类型
    - ✓不能把const对象、字面值或需要类型转换的对象传递给普通引用形参

```
// bad design: the first parameter should be a const string&
string::size_type find_char(string &s, char c,
                            int &occurs);
```

  - 下面调用编译时报错

```
find_char("Hello World", 'o', ctr);
```

练习

const形参和实参

◆练习6.19：判断下面调用是否合法？

```
double calc(double);
int count(const string &, char);
int sum(vector<int>::iterator, vector<int>::iterator, int);
vector<int> vec(10);
(a) calc(23.4, 55.1);   (b) count("abcda", 'a');
(c) calc(66);           (d) sum(vec.begin(), vec.end(), 3.8);
```

数组形参

参数传递

- 数组有两个特殊性质
  - 不允许拷贝数组
    - ✓无法以值传递的方式使用数组参数
  - 使用数组时，通常会将其转换成指针
    - ✓传递数组实际上传递的是指向首元素的指针
- 可以把形参写成类似数组的形式

```
// despite appearances, these three declarations of print are equivalent
// each function has a single parameter of type const int*
void print(const int*);
void print(const int[]);   // shows the intent that the function takes an array
void print(const int[10]); //dimension for documentation purposes(at best)
//以上三种写法不能同时出现，编译器会都编译成一样
 int i = 0, j[2] = {0, 1};
print(&i); // ok: &i is int*
print(j);  // ok: j is converted to an int* that points to j[0]
```

使用标记指定数组长度

数组形参

- 数组是以指针的形式传递给函数的，所以函数不知道数组的确切大小
- 管理数组实参的第一种方法是要求数组本身包含一个结束标记
  - C风格字符串

```
void print(const char *cp)
{
    if (cp)          // if cp is not a null pointer
        while (*cp)

            cout << *cp++;

}
```

使用标准库规范

数组形参

- 管理数组实参的第二种技术是传递指向数组首元素和尾后元素的指针

```
void print(const int *beg, const int *end)
{
   // print every element starting at beg up to but not including end
   while (beg != end)
      cout << *beg++ << endl; // print the current element
                              // and advance the pointer
}
```

  - 调用时，传入两个指针

```
int j[2] = {0, 1};
// j is converted to a pointer to the first element in j
// the second argument is a pointer to one past the end of j
print(begin(j), end(j)); // begin and end functions
```

显式传递数组大小参数

数组形参

- 管理数组实参的第三种办法是定义一个表示数组大小的形参

```
// const int ia[] is equivalent to const int* ia
// size is passed explicitly and used to control access to elements of ia
void print(const int ia[], size_t size)
{
    for (size_t i = 0; i != size; ++i) {
        cout << ia[i] << endl;
    }
}
```

  - 调用时，传入数组大小

```
int j[] = { 0, 1 };  // int array of size 2
print(j, end(j) - begin(j));
```

数组形参和const

数组形参

- 当函数不需要对数组元素进行写操作时，数组形参定义为指向const的指针
- 只有当函数确实要改变元素值时，才把形参定义成指向非const的指针

数组引用形参（了解）

数组形参

- 形参可定义为数组的引用
  - 引用形参绑定到对应的实参，即数组上

```
//ok: parameter is a reference to an array; the dimension is part of the type
void print(int (&arr)[10])
{
    for (auto elem : arr)
        cout << elem << endl;
}
```

  - 由于数组的大小是数组类型的一部分，只能将函数作用于大小固定的数组上

```
int i = 0, j[2] = {0, 1};
int k[10] = {0,1,2,3,4,5,6,7,8,9};
print(&i);   // error: argument is not an array of ten ints
print(j);    // error: argument is not an array of ten ints
print(k);    // ok: argument is an array of ten ints
```

传递多维数组

数组形参

- 将多维数组传递给函数时，传递的是指向首元素的指针
  - 指针是一个指向数组的指针
  - 数组第二维（以及后面所有维度）的大小都是数组类型的一部分，不能省略

// matrix points to the first element in an array whose elements are arrays of ten ints（了解）

```
void print(int (*matrix)[10], int rowSize) { /* . . . */ }
```

  - 等价的数组语法形式：

```
void print(int matrix[][10], int rowSize) { /* . . . */ }
```

练习

数组形参

◆练习6.24：指出下面代码存在的问题

```
void print(const int ia[10])
{
    for (int i = 0; i != 10; ++i)
        cout << ia[i] << endl;
}
```

练习

数组形参

◆练习6.24：指出下面代码存在的问题

```
void print(const int ia[10])
{
    for (int i = 0; i != 10; ++i)
        cout << ia[i] << endl;
}
```

如果出现：

```
int i[2]={0};
print(i);
//void print(const int (&ia)[10])
```

main:处理命令行选项(不讲)

参数传递

prog -d -o ofile data0

- 可以通过可选的形参向main函数传递命令行选项

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

- 形参argv是一个数组，元素是指向C风格字符串的指针
- 形参argc表示数组中字符串数量
- 等价形式：

int main(int argc, char **argv) { ... }

返回类型和return语句

- return语句终止当前正在执行的函数并将控制权返回到调用该函数的地方
- return语句有两种形式：

  return;

  return *expression*;

无返回值函数

返回类型和return语句

- 没有返回值的return语句只能用在返回类型是void的函数中
  - 返回void的函数不要求必须有return语句
  - void函数可使用return在中间位置提前退出
- 返回类型是void的函数也能使用return语句的第二种形式
  - *expression*必须是另一个返回void的函数
  - 从void函数返回其他类型的表达式将产生编译错误

例子

无返回值函数

```
void swap(int &v1, int &v2)
{
    // if the values are already the same, no need to swap, just return
    if (v1 == v2)
        return;
    // if we're here, there's work to do
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    // no explicit return necessary
}
```

有返回值函数

返回类型和return语句

- return语句的第二种形式提供函数结果
  - return语句返回值的类型必须与函数的返回类型相同，或者能隐式转换成函数的返回类型
- 在含有return语句的循环后面应该也有一条return语句，否则会导致错误
  - 很多编译器都无法发现此类错误
  - 编译器只保证每条return语句的结果类型正确

例子

有返回值函数

```
// incorrect return values, this code will not compile
bool str_subrange(const string &str1, const string &str2)
{
    // same sizes: return normal equality test
    if (str1.size() == str2.size())
        return str1 == str2;   // ok: == returns bool
    // find the size of the smaller string
    int size = (str1.size() < str2.size()) ? str1.size() : str2.size(); //
    look at each element up to the size of the smaller string for
    (int i = 0; i != size; ++i) {
        if (str1[i] != str2[i])
            return; // error #1: no return value ; compiler should detect this error
    }
    // error #2: control might flow off the end of the function without a return
    // the compiler might not detect this error
}
```

值是如何被返回的

返回类型和return语句

- 返回值用于初始化调用点的一个临时变量，该变量就是函数调用的结果

```
// return the plural version of word if ctr is greater than 1
string make_plural(size_t ctr, const string &word,
                   const string &ending)
{
    return (ctr > 1) ? word + ending : word;
}
```

  - 如果返回引用，则该引用是它所引对象的别名

```
// return a reference to the shorter of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

不要返回局部对象的引用或指针

返回类型和return语句

- 返回局部对象的引用或指针是错误的
  - 函数完成后，局部对象被释放，引用或指针将指向一个不存在的对象

```
// disaster: this function returns a reference to a local object
const string &manip()
{
    string ret="hello";
    if (!ret.empty())
        return ret;     // WRONG: returning a reference to a local object!
    else
        return "Empty"; // WRONG: "Empty" is a local temporary string
}
    int main(){
        cout<<manip();
    }
```

返回类类型的函数和调用运算符

返回类型和return语句

- 调用运算符的优先级与点运算符和箭头运算符相同，满足左结合性
  - 如果函数返回类类型的指针、引用或对象，就能使用函数调用的结果访问对象的成员

```
const string &shorterString(const string &s1, const string &s2)
// call the size member of the string returned by shorterString
int sz = shorterString(s1, s2).size();
```

引用返回左值

返回类型和return语句

- 函数的返回类型决定函数调用是否是左值
- 调用一个返回引用的函数得到左值，其他返回类型得到右值
  - 能为返回类型是非常量引用的函数的结果赋值

```
char &get_val(string &str, string::size_type ix)
{
    return str[ix]; // get_val assumes the given index is valid
}
int main()
{
    string s("a value");
    cout << s << endl;   // prints a value
    get_val(s, 0) = 'A'; // changes s[0] to A
    cout << s << endl;   // prints A value
    return 0;
}
```

主函数main的返回值（了解）

返回类型和return语句

- main的返回类型是int，但允许main函数没有return语句直接结束
  - 编译器隐式地插入一条返回0的return语句
- main函数返回0表示执行成功，返回其他值表示执行失败
- cstdlib头文件定义了两个预处理变量分别表示成功和失败

```
int main()
{
    if (some_failure)
        return EXIT_FAILURE;  // defined in cstdlib
    else
        return EXIT_SUCCESS;  // defined in cstdlib
}
```

递归

返回类型和return语句

- 如果一个函数调用自身，不管是直接调用还是间接调用，该函数都称为递归函数(recursive function)
  - 递归函数中一定有条路径不包含递归调用

```
// calculate val!, which is 1 * 2 * 3 . . . * val
int factorial(int val)
{
    if (val > 1)
        return factorial(val-1) * val;
    return 1;
}
```

递归

返回类型和return语句

- 下面表格给出了参数为5时，factorial函数的执行轨迹

调用	返回	值
factorial(5)	factorial(4) * 5	120
factorial(4)	factorial(3) * 4	24
factorial(3)	factorial(2) * 3	6
factorial(2)	factorial(1) * 2	2
factorial(1)	1	1

练习

返回类型和return语句

- 练习6.32：下面的函数是否合法？如果合法，说明其功能

```
int &get(int *arry, int index) { return arry[index]; }
int main() {
    int ia[10];
    for (int i = 0; i != 10; ++i)
        get(ia, i) = i;
}
```

6.3.3返回数组指针（不讲，了解）

返回类型和return语句

- 函数不能返回数组，但可以返回数组的指针或引用
- 可使用类型别名来简化定义返回数组的指针或引用的函数

```
typedef int arrT[10];  // arrT is a synonym for the type array of ten ints
using arrtT = int[10]; // equivalent declaration of arrT
arrT* func(int i);     // func returns a pointer to an array of ten ints
```

6.4 函数重载

- 在同一作用域中名字相同但形参列表不同的函数称为重载(overloaded)函数

```
void print(const char *cp);
void print(const int *beg, const int *end);
void print(const int ia[], size_t size);
```

- 编译器根据传入的实参类型推断调用的是哪个函数

```
int j[2] = {0,1};
print("Hello World");        // calls print(const char*)
print(j, end(j) - begin(j)); // calls print(const int*, size_t)
print(begin(j), end(j));     // calls print(const int*, const int*)
```

定义重载函数

函数重载

- 重载的函数必须在形参数量或形参类型上有所不同
- 不允许两个函数除了返回类型其他所有的要素都匹配

```
Record lookup(const Account&);
bool lookup(const Account&);   // error: only the return type is different
```

判断形参类型是否相异

函数重载

- 有时两个形参列表看起来不一样，但实际上是相同的

```
// each pair declares the same function
Record lookup(const Account &acct);
Record lookup(const Account&); // parameter names are ignored

typedef Phone Telno;
Record lookup(const Phone&);
Record lookup(const Telno&); // Telno and Phone are the same type
```

重载和const形参（了解）

函数重载

- 顶层const形参无法和没有顶层const的形参区分开

```
Record lookup(Phone);
Record lookup(const Phone);   // redeclares Record lookup(Phone)
Record lookup(Phone*);
Record lookup(Phone* const);  // redeclares Record lookup(Phone*)
```

- 如果形参类型是指针或引用，可以通过其指向的是const对象还是非常量对象来实现函数重载，此时const是底层的

```
// functions taking const and nonconst references or pointers have different parameters
// declarations for four independent, overloaded functions
Record lookup(Account&);       // function that takes a reference to Account
Record lookup(const Account&); // new function that takes a const reference
Record lookup(Account*);       // new function, takes a pointer to Account
Record lookup(const Account*); // new function, takes a pointer to const
```

const_cast和重载（了解）

函数重载

- const_cast在重载函数时非常有用
  - const_cast可以把普通引用强制转换为const的引用，或反过来

```
// return a reference to the shorter of two strings
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);
}
```

调用重载的函数

函数重载

- 函数匹配(function matching)：编译器比较调用实参与重载集合中每一个函数的形参后，决定要调用哪个函数
- 有三种可能的结果
  - 编译器找到一个与实参最佳匹配(best match)的函数，并生成调用该函数的代码
  - 找不到任何一个函数与调用实参相匹配，编译器发出无匹配(no match)的错误信息
  - 有多于一个函数可以匹配，但每一个都不是明显的最佳选择，此时发生二义性调用(ambiguous call)错误

6.4.1 重载与作用域（了解）

函数重载

- 在不同的作用域中无法重载函数名
  - 如果在内层作用域中声明名字，它将隐藏外层作用域声明的同名实体

```
string read();
void print(const string &);
void print(double);   // overloads the print function
void fooBar(int ival)
{
   bool read = false; // new scope: hides the outer declaration of read
   string s = read(); // error: read is a bool variable, not a function
   // bad practice: usually it's a bad idea to declare functions at local scope
   void print(int);  // new scope: hides previous instances of print
   print("Value: "); // error: print(const string &) is hidden
   print(ival);      // ok: print(int) is visible
   print(3.14);      // ok: calls print(int); print(double) is hidden
}
```

练习

函数重载

- 练习6.39：说明下面的函数重载声明是否合法

(a) int calc(int, int);

int calc(const int, const int);

(b) int get();

double get();

(c) int *reset(int *);

double *reset(double *);

练习

函数重载

- 练习6.39：说明下面的函数重载声明是否合法

```
(a) int calc(int, int);                    //error
int calc(const int, const int);
(b) int get();
double get();                              //error
(c) int *reset(int *);
double *reset(double *);                   //ok
```

特殊用途语言特性

- 三种与函数相关的语言特性
  - 默认实参
  - 内联函数和constexpr函数
  - 调试辅助功能

默认实参

特殊用途语言特性

- 如果有形参在函数多次调用中都被赋以相同的值，那么可以把该值声明为函数的默认实参(default argument)
  - 默认实参作为形参的初始值出现在参数列表中
  - 可以为一个或多个形参定义默认值
  - 如果某个形参有默认实参，那么它后面的所有形参都必须有默认实参
  - 调用含有默认实参的函数时，可以包含或省略该实参

```
string screen(int ht = 24, int wid = 80, char backgrnd = ' ');
```

调用有默认实参的函数

默认实参

- 如果想使用默认实参，只需在调用函数时省略该实参

```
string screen(int ht = 24, int wid = 80, char backgrnd = ' ');
string window;
window = screen();  // equivalent to screen(24,80,' ')
window = screen(66);// equivalent to screen(66,80,' ')
window = screen(66, 256);      // screen(66,256,' ')
window = screen(66, 256, '#'); // screen(66,256,'#')
```

- 函数调用时实参按其位置解析，默认实参负责填补调用时缺少的尾部实参

```
window = screen(, , '?'); // error: can omit only trailing arguments
window = screen('?');     // calls screen('?',80,' '), int('?')=63
```

默认实参声明

默认实参

- 设计含有默认实参的函数时，要合理安排形参的顺序
  - 让经常使用默认值的形参出现在后面
- 在给定的作用域中，一个形参只能被赋予一次默认实参
  - 函数的后续声明只能为之前那些没有默认值的形参添加默认实参，且该形参右侧的所有形参必须都有默认值

```
// no default for the height or width parameters
string screen(int, int, char = ' ');
```

  - 不能修改已声明的默认值，但可以添加默认实参

```
string screen(int, int, char = '*'); // error: redeclaration string
screen(int = 24, int = 80, char); // ok: adds default arguments
```

```
 string screen(int, int, char = '*');
//string screen(int, int, char = '**');   //error
//string screen(int=3 , int=4, char='**'); //error
string screen(int=3 , int=4, char);  //ok
int main()
{
    screen();
    string screen(int , int, char);
    //string screen(int=0 , int=1, char='!!');  //ok
    screen(3,2,'b');
   // string screen(int=1 , int=3, char='!!');  //error: redefinition of default argument
}

string screen(int a, int b, char c)
{
cout<<a<<b<<c<<endl;
    return "hi";
}
```

练习

默认实参

- 练习6.40：下面的声明是否有错？

  (a) int ff(int a, int b = 0, int c = 0);

  (b) char *init(int ht = 24, int wd, char bckgrnd);

- 练习6.41：下面的调用是否非法？是否有合法但与程序员初衷不符的调用？

  char *init(int ht, int wd = 80, char bckgrnd = ' ');

  (a) init();

  (b) init(24,10);

  (c) init(14, '*');

内联函数和constexpr函数

特殊用途语言特性

- 经常把规模较小的操作定义成函数
  - 易阅读和理解，易维护，可重复利用

```
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

- 调用函数一般比等效的表达式求值慢
  - 调用前保存寄存器，返回时恢复
  - 可能需要拷贝实参

内联函数避免调用开销

内联函数和constexpr函数

- 定义为内联(inline)的函数通常可以在调用点内联地(in line)展开
  - 内联说明只是向编译器发出的一个请求

```
// inline version: find the shorter of two strings
inline const string &
shorterString(const string &s1, const string &s2)
{
        return s1.size() <= s2.size() ? s1 : s2;
}
```

  - 如下调用

```
cout << shorterString(s1, s2) << endl;
```

  - 将在编译过程中展开为

```
cout << (s1.size() < s2.size() ? s1 : s2) << endl;
```

constexpr函数

内联函数和constexpr函数

- constexpr函数(constexpr function)是指能用于常量表达式的函数
  - 函数的返回类型及所有形参类型必须是字面值类型
  - 函数体内有且只有一条return语句
    - ✓也可以包含其他运行时不执行操作的语句，如空语句、类型别名以及using声明

```
constexpr int new_sz() { return 42; }
constexpr int foo = new_sz();  // ok: foo is a constant expression
```

constexpr函数（不讲，了解）

内联函数和constexpr函数

- 执行初始化时，编译器把对constexpr函数的调用替换成其结果值
  - constexpr函数被隐式地指定为内联函数
- 允许constexpr函数返回一个非常量的值

```
constexpr int new_sz() { return 42; }
// scale(arg) is a constant expression if arg is a constant expression
constexpr size_t scale(size_t cnt) { return new_sz() * cnt; }
int arr[scale(2)]; // ok: scale(2) is a constant expression
int i = 2;         // i is not a constant expression
int a2[scale(i)]; // error: scale(i) is not a constant expression
```

6.5.3 调试辅助（了解）

特殊用途语言特性

- 程序中可以包含一些仅在开发时使用调试代码，发布时要屏蔽掉这些代码
- 使用两项预处理器功能：
  - assert
  - NDEBUG

assert预处理宏

调试辅助

- 预处理宏(preprocessor macro)的行为类似于内联函数
- assert宏使用一个表达式作为条件：

  assert(*expr*);

  - 对*expr*求值，如果为假(0)，assert输出信息并终止程序；否则，什么都不做
- assert宏定义在cassert头文件中，宏名字在程序中必须唯一
- assert宏常用于检查“不能发生”的条件

  assert(word.size() > threshold);

NDEBUG预处理变量

调试辅助

- assert宏的行为依赖于预处理变量NDEBUG的状态
  - 如果定义了NDEBUG，那么assert什么都不做
  - 默认情况下没有定义NDEBUG，此时assert将执行运行时检查
- 可以用一条#define语句定义NDEBUG来关闭调试状态
- 很多编译器都提供了命令行选项来定义预处理变量

```
$ CC -D NDEBUG main.C  # use /D with the
Microsoft compiler
```

NDEBUG预处理变量

调试辅助

- 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)
{
#ifndef NDEBUG
    // __func__ is a local static defined by the compiler that holds the function's name
    cerr << __func__ << ": array size is " << size << endl;
#endif
// ...
}
```

  - 使用变量__func__输出当前调试函数的名字
- 编译器为每个函数都定义了__func__
  - const char的局部静态数组，存放函数名

NDEBUG预处理变量

调试辅助

- 预处理器还定义了4个调试时很有用的名字
  - __FILE__ 存放文件名的字符串字面值
  - __LINE__ 存放当前行号的整型字面值
  - __TIME__ 存放文件编译时间的字符串字面值
  - __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
         << "       Compiled on " << __DATE__
         << " at " << __TIME__ << endl
         << "       Word read was \"" << word
         << "\":  Length too short" << endl;
```

6.6 函数匹配

- 许多情况下，容易确定哪个重载函数匹配给定的调用
- 但当重载函数的形参数量相等且某些形参的类型可以由类型转换得到时，函数匹配就没那么简单了

```
void f();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6);  // calls void f(double, double)
```

候选函数和可行函数

函数匹配

- 函数匹配的第一步是确定本次调用的候选函数(candidate function)集合
  - 候选函数具备两个特征：与被调用的函数同名；其声明在调用点可见
- 第二步是从候选函数中选出能被提供的实参调用的可行函数(viable function)
  - 可行函数也有两个特征：形参数量与实参数量(包含默认实参)相等；每个实参的类型与对应形参相同或能转换为形参类型
  - f(5.6)的可行函数有f(int)和f(double, double)
- 如果没找到可行函数，编译器报无匹配函数的错误

寻找最佳匹配

函数匹配

- 函数匹配的第三步是从可行函数中寻找形参类型与实参类型最匹配的函数
- 最佳匹配(best match)是指有且只有一个可行函数满足下列条件：
  - 该函数每个实参的匹配都不劣于其他可行函数需要的匹配
  - 至少有一个实参的匹配优于其他可行函数提供的匹配
- 如果没有找到最佳匹配，编译器报二义性调用的错误
  - 对于f(42, 2.56)，f(int, int)和f(double, double)中无最佳匹配

练习

函数匹配

◆练习6.50：找出下面调用的最佳匹配，如果没有，指出错误原因

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

(a) f(2.56, 42);

(b) f(42);

(c) f(42, 0);

(d) f(2.56, 3.14);

实参类型转换

函数匹配

- 为了确定最佳匹配，编译器将实参类型到形参类型的转换划分成几个等级
  - 精确匹配(exact math)
    - ✓实参类型与形参类型相同
    - ✓实参从数组类型或函数类型转换成对应的指针类型
    - ✓实参添加或去除顶层const
  - 通过const转换实现的匹配
  - 通过类型提升实现的匹配
  - 通过算术类型转换或指针转换实现的匹配
  - 通过类类型实现的匹配

类型提升和算术类型转换的匹配

实参类型转换

- 小整型总是提升到int类型或更大的整数类型

```
void ff(int);
void ff(short);
ff('a');   // char promotes to int; calls f(int)
```

- 所有算术类型转换的级别都一样
  - 从int向unsigned int的转换并不比从int向double的转换的级别高

```
double i=3.14;
void manip(long);
void manip(float);
manip(i); // error: ambiguous call
```

函数匹配与const实参

实参类型转换

- 如果重载函数的区别在于引用或指针形参是否指向const，那么编译器通过实参是否常量来决定调用哪个函数

```
Record lookup(Account&);       // function that takes a reference to Account
Record lookup(const Account&); // new function that takes a const reference
const Account a;
Account b;
lookup(a);   // calls lookup(const Account&)
lookup(b);   // calls lookup(Account&)
```

  - 第一个调用只有1个可行函数，第二个调用有2个可行函数

练习

实参类型转换

- 练习6.53：说明下列的函数重载声明是否合法？

(a) int calc(int&, int&);

int calc(const int&, const int&);

(b) int calc(char*, char*);

int calc(const char*, const char*);

(c) int calc(char*, char*);

int calc(char* const, char* const);

函数指针

- 函数指针(function pointer)指向函数
- 函数的类型由其返回类型和形参类型共同决定，与函数名无关

```
// compares lengths of two strings
bool lengthCompare(const string &, const string &);
```

  - 函数的类型为：bool(const string&, const string&)
- 声明一个指向函数的指针，只需用指针替换函数名即可：

```
// pf points to a function returning bool that takes two const string references
bool (*pf)(const string &, const string &);  // uninitialized
```

  - *pf两端的括号必不可少

使用函数指针

函数指针

- 函数名作为值使用时，自动转换成函数指针

```
bool (*pf)(const string &, const string &);  // uninitialized
bool lengthCompare(const string &, const string &);
pf = lengthCompare;  // pf now points to the function named lengthCompare
pf = &lengthCompare; // equivalent assignment: address-of operator is optional
```

- 可直接使用指向函数的指针调用该函数，而不必解引用该指针

```
bool b1 = pf("hello", "goodbye");    // calls lengthCompare
bool b2 = (*pf)("hello", "goodbye"); // equivalent call
bool b3 = lengthCompare("hello", "goodbye"); // equivalent call
```

- 不同函数类型的指针间不存在转换规则
  - 可为空函数指针赋一个nullptr或值为0的整型常量表达式

重载函数的指针

函数指针

- 可以定义一个指向重载函数的指针

```
void ff(int*);
void ff(unsigned int);
void (*pf1)(unsigned int) = ff;  // pf1 points to ff(unsigned)
```

- 编译器通过指针类型决定选用哪个函数，指针类型必须精确匹配某一个重载函数

```
void (*pf2)(int) = ff;    // error: no ff with a matching parameter list
double (*pf3)(int*) = ff; // error: return type of ff and pf3 don't match
```

函数指针形参

函数指针

- 不能定义函数类型的形参，但形参可以是指向函数的指针
  - 可以把形参写成函数类型形式，但实际上当成函数指针使用

```
// third parameter is a function type and is automatically treated as a pointer to function
void useBigger(const string &s1, const string &s2,
               bool pf(const string &, const string &));
// equivalent declaration: explicitly define the parameter as a pointer to function
void useBigger(const string &s1, const string &s2,
               bool (*pf)(const string &, const string &));
```

- 函数作为实参时，自动转换成函数指针

```
// automatically converts the function lengthCompare to a pointer to function
useBigger(s1, s2, lengthCompare);
```

函数指针形参(后面几页都为了解，不讲)

函数指针

- 可使用类型别名和decltype简化使用函数指针的代码

```
bool lengthCompare(const string &, const string &);
// Func and Func2 have function type
typedef bool Func(const string&, const string&);
typedef decltype(lengthCompare) Func2; // equivalent type
// FuncP and FuncP2 have pointer to function type
typedef bool(*FuncP)(const string&, const string&);
typedef decltype(lengthCompare) *FuncP2;  // equivalent type
```

  - decltype返回函数类型，加上*才得到指针

```
// equivalent declarations of useBigger using type aliases
void useBigger(const string&, const string&, Func);
void useBigger(const string&, const string&, FuncP2);
```

返回函数指针

函数指针

- 不能返回一个函数，但是能返回指向函数的指针
  - 必须显式地将返回类型写成指针形式
    - ✓和函数类型的形参不同，编译器不会自动将函数返回类型当成对应的指针类型处理
- 声明一个返回函数指针的函数，最简单的办法是使用类型别名

```
using F = int(int*, int);     // F is a function type, not a pointer
using PF = int(*)(int*, int); // PF is a pointer type
PF f1(int); // ok: PF is a pointer to function; f1 returns a pointer to function
F f1(int);  // error: F is a function type; f1 can't return a function
F *f1(int); // ok: explicitly specify that the return type is a pointer to function
```

返回函数指针

函数指针

- 可以使用下面的形式直接声明：

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

  - 按照由内而外的顺序解读这条声明语句
- 还可以使用尾置返回类型的方式声明返回指针类型的函数：

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

使用decltype

函数指针

- 如果明确知道返回的函数是哪一个，就能使用decltype简化书写函数指针返回类型的过程
  - decltype返回函数类型而非指针类型

```
string::size_type sumLength(const string&, const string&);
string::size_type largerLength(const string&, const string&);
// depending on the value of its string parameter,
// getFcn returns a pointer to sumLength or to largerLength
decltype(sumLength) *getFcn(const string &);
```