6.1 函数基础

- 通过调用运算符 () 来调用函数
- 函数的形参列表：形参用，隔开，每个形参都必须在有一个声明符的声明（即使类型都相同）
- 函数的返回类型不能是数组类型或函数类型，但可以是指向他们的指针

6.1.1 局部对象

- 如果想令局部对象的生命周期贯穿函数调用之后的时间，需要将基变成局部静态对象local static object
  - 注意局部静态对象仍只可在函数内部可见和访问。只是在下一次调用该函数时会使用之间的内存空间
  - 局部静态对象的声明在前面加上 `static`
  - 内置类型的局部静态对象初始化为0（常规）

6.1.2 函数声明

- 在头文件中进行函数声明即可

6.1.3 分离式编译

6.2 参数传递

6.2.1 传值参数（拷贝会增加内存开销）

6.2.2 传引用参数（避免拷贝，推荐使用）

- 如果函数无须改变形参的值，最好将其声明成常量引用
- 使用引用形参返回额外信息，如：

```
// 返回字符串中字符出现的首位置，及其出现的总次数
string::size_type find_char(const string &s, char c, string::size_type &occurs){
    auto index = s.size();
    occurs = 0;
    for(decltype(index) i = 0; i != s.size(); ++i){
        if(s[i] == c) {
            if(index == s.size()){
                index = i; // 因为要记录下标，所以没用范围for循环
            }
            ++occurs;
        }
    }
    return index;
}
```

6.2.3 const形参和实参

- 指针或引用参与const：形参是引用，实参不能是字面值；除非是常量引用，这时c++会把字面值初始化为常量引用

- 形参尽量使用常量引用

6.2.4 数组形参（数组不能被拷贝，且通常会被转换成指针）

- 数组是以指针的形式传递给函数的，函数对于数组的信息并不明确，为此有管理数组指针形参的三种常用技术

1. 作用标记指定数组长度，如c风格字符串的空字符
2. 使用标准库规范：传递首元素和尾后元素

```
void print(const int*beg,const int *end);
print(begin(j),end(j));
```

3. 显式传递一个表示数组大小的形参

```
void print(const int ia[],size_t size);
print(j,end(j)-begin(j));
```

- 数组引用形参

```
f(int &arr[10]); // 错误：arr是一个数组，存放10个整型引用
f(int (&arr)[10]); // 正确，arr是整型数组的引用

// 数组引用形参会限制函数只能作用于大小为10的数组， 如：
int i = 0, j[2] = {0,1};
int k[10] = {0,1,2,3,4,5,6,7,8,9};
print(&i); // 错误
print(j); // 错误
print(k); // 正确

// 传递多维数组：多维数组的首元素本身就是一个数组
void print(int (*matrix)[10], int rowSize);
```

6.2.5 main:处理命令行选项

```
// 下面两个main函数声明是相同的
int main(int argc,char *argv[]);
int main(int argc,char **argv);
// argv[0]是程序名，可选参数从argv[1]开始
```

6.2.6 含有可变形参函数

1. initializer_list类型的形参：适用于实参数量未知，但类型都相同。initializer_list表示某种特定类型的数组

```
// initializer_list相关操作,基本与vector类似
initializer_list<T> lst;
initializer_list<T> lst{a,b,c};
initializer_list<T> lst2(lst);
lst2 = lst;
lst.size();
lst.begin();
lst.end();

// 与vector不同的是initializer_list对象中的元素是常量值，不可变更
// 利用initializer_list将形参设为可变
void error_msg(initializer_list<string>);
error_msg({"a","b","c"});
error_msg({"s","t"});
```

2. 省略符形参：主要是用于c++程序访问c代码的接口（不是很懂）

6.3 返回类型的return语句

`return`：终止当前执行的函数，将控制权返回到调用该函数的地方

6.3.1 无返回值函数

6.3.2 有返回值函数

- 在含有return语句的循环体后也应该有return语句
- 返回的值用于初始化调用点的一个临时量，该临时量就是调用函数的结果
- 不要返回局部对象的指针或引用
- 返回类类型的函数和调用运算符 `auto sz = shorterString(s1,s2).size();`
- 引用返回左值，其他类型返回右值；
- 特别的，也可以为返回类型是非常量引用的函数的结果赋值，如： `get_val(s,0) = 'A';`
- 列表初始化返回值：

```
vector<string> process(){
    return {"A","B","C"};
}
```

- 主函数main的返回值
  1. main函数结尾处有隐式的 `return 0`
  2. 0表示执行成功，其他表示失败，非0值的具体含义与机器有关
  3. 为使返回值与机器无关，可使用 `cstdlib` 定义两个预处理变量 `EXIT_FAILURE`，`EXIT_SOCCESS` （预处理变量不能加 `std::`，也不用在using声明中出现）
  4. main函数无法调用自己，即不能对main函数使用递归

6.3.3 返回数组指针

1. 数组不能被拷贝所以不能被返回，只能返回数组的指针或引用，可以配合类型别名使用

```
typedef int arrT[10]; // using arrT = int[10];
arrT *func(int i); // int (*func(int i))[10];
```

2. 尾置返回类型：适用于返回类型是数组引用或指针等复杂情况

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

3. 使用decltype,decltype不会把数组类型转换为指针

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

6.4 函数重载（只许形参列表不同，变量名不同不会触发重载）

1. 无法区分顶层const和非顶层const

```
Record lookup(phone); // phone 是自定义类型
Record lookup(const phone); // 重复声明

Record lookup(phone *);
Record lookup(const phone *); // 重复声明
```

2. 对于指针和引用，可以用底层const实现重载

```
Record lookup(Account &);
Record lookup(const Account &); // 实现重载

Record lookup(Account *);
Record lookup(const Account *); // 实现重载
```

3. const_cast 和重载

```
const string &shorterString(const string &s1,const string &s2){
    return s1.size()<=s2.size()?s1.size():s2.size();
} // 当实参是非const string时，返回结果仍是const string的引用

string &shorterString(string &s1,string &s2){
    auto &r = shorterString(const_cast<const string&>(s1),const_cast<const string&
>(s2));
    return const_cast<string&>(r);
} // 重载版本，当实参不是常量时，得到的结果是一个普通的引用
```

4. 调用有重载的函数：最佳匹配，无匹配，二义性匹配。只有最佳匹配可以让程序正常运行
5. 重载与作用域

```
// 内层作用域声明变量名会隐藏外层作用域声明的同名实体，所以不同作用域的两个函数无法重载
// c++中名字查找发生在类型检查之前
```

6.5 特殊用途语言特性：默认实参，内联函数，constexpr函数

6.5.1 默认实参

```
typedef string::size_type sz;
// 为每一个形参赋予默认值，一旦某个形参有默认值，此后都必须有
string screen(sz ht = 24, sz wid = 80, char backgrnd = '');

// 调用时按位置解析，默认实参负责填补缺失的尾部实参
string window;
window = screen(); // window = screen(24,80,'');
window = screen(66); // window = screen(66,80,'');
window = screen(66,256); // window = screen(66,256,'');
window = screen(66,256,'#');


// 通常在函数声明时指定默认实参，并将该声明放在合适头文件中
```

6.5.2 内联函数和constexpr函数

1. （inline）内联函数可避免函数调用的开销，用于优化规模小、流程直接、频繁调用的函数

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

// 内联声明只是向编译器发出的一个请求，可能会被忽略
```

2. constexpr函数：可以用于常量表达式的函数
   A. 要求：返回值及形参类型必须是字面值类型，函数中有且只有一个return语句
   B. constexpr函数被隐式指定为内联函数
   C. 内联函数和constexpr函数定义在头文件中

6.5.3 调试帮助：assert，NDEBUG？？？

6.6 函数匹配

6.6.1 实参类型转换：天书，不会，再见

6.7 函数指针

```
// 声明函数指针只要把函数名替换为指针（前面加个*）即可
bool (*pf)(const string&, const string&);

// 使用1：把函数名当成值来使用，会自动转换成指针
pf2 = lengthCompare; // pf2是一个类型匹配的指针，lengthCompare是一个函数
```

In [ ]:

```

```