USTC

Chapter 8
Technicalities: Functions, etc.

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

Overview

- 本章和下一章将对C++编程语言的一些技术细节进行探讨，并进行系统化说明
- 本章概览
  - 编程语言技术细节
  - 声明 Declarations
    - 定义
    - 头文件与预处理
    - 作用域
  - 函数 Functions
    - 声明和定义
    - 参数
  - 调用：传值、传引用、传常量(const)引用
  - 命名空间
    - “using”声明和指令

编程语言技术细节

- 这是必要的
  - 编程语言是一门外语
  - 当你学习一门外语时，需要研究它的语法和词汇
  - 这就是本章和下一章要做的事情
- 因为：
  - 程序必须被精确和完整地描述
    - 计算机是一个非常“愚蠢”的机器(尽管它很快)
    - 计算机不能猜测出你“说的真正意思”(也不应该这样做)
  - 因此，我们必须学习规则
    - 实际上仅学习其中的一部分(C++ 标准有782 页之多)
- 然而，永远不要忘记
  - 我们真正学习的是编程
  - 我们要产出的是程序和系统
  - 编程语言仅仅是一个工具

关于技术细节

- 不要花费太多的时间在语法和语义细节问题上
  - 表达和解决一个问题总是有多种形式
  - 就像我们的文字语言一样
- 多数的设计和编程概念是通用的
  - 至少是被几种流行的编程语言广泛支持的
  - 因此，我们在C++中学习到的东西也可以用在其他语言上
- 编程语言技术细节是针对给定语言特有的
  - 不过，C++语言提供的绝大部分技术特征在其他语言 (如 C, Java, C#, etc) 中都有明显对应的内容

声明 Declarations

- 声明将一个名称引入到作用域中
  - 声明为命名对象指定了一个类型
  - 有时，声明还进行初始化
- 在C++中，名称只有声明以后才能够使用
- 示例：
  - **int a = 7;** *// an int variable named 'a' is declared*
  - **const double cd = 8.7;** *// a double-precision floating-point constant*
  - **double sqrt(double);** *// a function taking a double argument and returning a double result*
  - **vector<Token> v;** *// a vector variable of **Token**s (variable)*

声明 Declarations

- 声明通常是通过一个“头(文件)”引入到一个程序中
  - 一个“头”是一个包含声明的文件，它提供了程序中其它部分的访问接口

- 这为“抽象”提供了可能
  - 你不必知道函数的细节，而只需像cout 那样使用即可
  - 如果添加

```
#include "../../std_lib_facilities.h"
```

到你的代码中，文件std_lib_facilities.h 中的声明对你就是可用的(包括 cout 等)

定义 Definitions

- 如果声明还给出了声明实体(实现)的完整描述，则称之为定义
  - 定义是一种特殊的声明，但声明不一定是定义
  - 定义示例：

```
int a = 7;
int b;                                  // an int with the default value (0)
vector<double> v;     // an empty vector of doubles
double sqrt(double) { … };  // i.e. a function with a body
struct Point { int x; int y; };
```

  - 是声明而不是定义的示例：

```
double sqrt(double);     // function body missing
struct Point;                           // class members specified elsewhere
extern int a;// extern means “not definition”
                                        // “extern” is archaic; we will hardly use it
```

过时的，基本不用了！

声明与定义

- 你不能定义一个对象两次
  - 定义说明了它是什么(会分配内存进行存储)
  - 例如：

```
int a;        // definition
int a;        // error: double definition
double sqrt(double d) { … }      // definition
double sqrt(double d) { … }      // error: double definition
```

- 但可以声明一个对象多次
  - 声明仅说明了它如何被使用(非定义不分配内存)

```
int a = 7;                  // definition (also a declaration)
extern int a;               // declaration
double sqrt(double);        // declaration
double sqrt(double d) { … } // definition (also a declaration)
```

为什么声明和定义都需要呢?

- 为了引用一个对象，只需要包括它的声明就可以了
- 大部分情况下，我们想在其他地方对它进行定义
  - 一个文件的后面部分
  - 在另一个文件里
    - 甚至可能是其他人编写的
- 声明用于指定接口
  - 你自己的代码
  - 程序库
    - 对程序库是关键的：我们不能也不愿意自己写全部代码
- 在较大的程序中
  - 将所有相关的声明放在头文件中，以便于共享

头文件和预处理

- “头”是包含函数、类型、常量或其他程序组件声明的一个文件
- 下面的结构

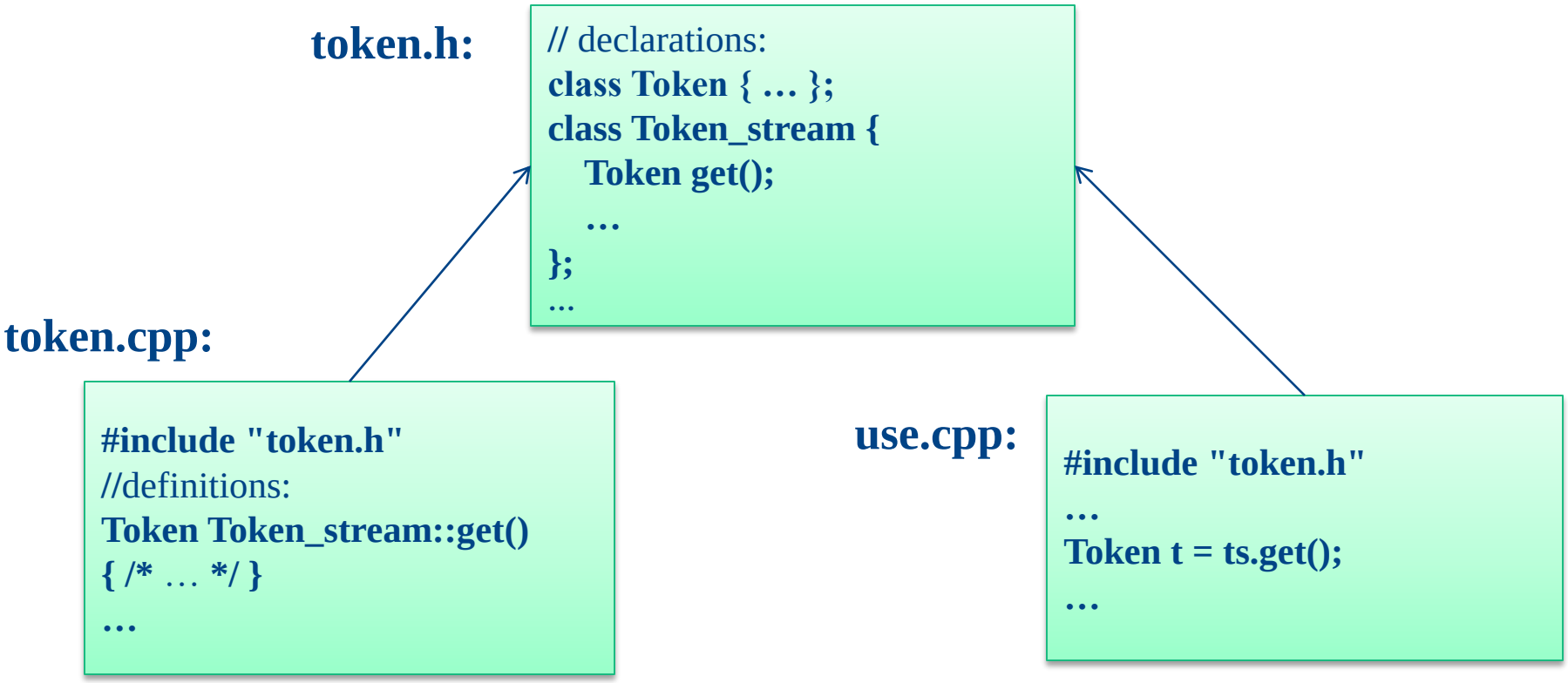
```
#include "../../std_lib_facilities.h"
```

是一个“预处理”指令

  - 简单地将头文件中的声明复制到文件的#include 指令处
  - 头文件是简单的文本文件
- “头”提供了函数、类型等的访问接口，你可以在自己的程序中使用它们
  - 通常，你不需要关心它们是如何实现的
  - 实际的函数、类型等在其他源文件中进行定义
    - 通常是程序库的一部分
  - 头文件只能包含那些可以在多个文件中重复多次的声明(函数声明、类定义和数据常量定义)

考虑一下，若头文件中有定义，如何在多个文件#include时避免重复定义错误？

源文件



- 头文件 (此处的 token.h) 定义了用户代码和实现代码之间的接口 — 通常是在一个库中
- 在所有.cpp文件 (定义和使用) 中使用同样的#include 声明 有利于一致性检查

作用域 Scope

- 作用域是一个程序文本的区域 — 编译器特性
  - 例如：
    - 全局作用域 (任何其他作用域之外的区域)
    - 类作用域 (在一个类内)
    - 局部作用域 (在花括号 { … } 之间)
    - 语句作用域 (e.g. **for** 语句)
- 一个作用域的名称对该作用域和嵌套内的作用域是可见的
  - 当然，必须在名称声明之后 ("不能先用" 规则)
- 作用域使 "实体 (things)" 是局部的
  - 避免我的变量、函数等实体与你的发生冲突
  - 谨记：实际的程序中有成千上万个实体
  - 局部性是最好的！
    - 尽量使名称局部化

作用域

```
#include "std_lib_facilities.h"          // get max and abs from here
// no r, i, or v here
class My_vector {
    vector<int> v;                       // v is in class scope
public:
    int largest()                                   // largest is in class scope
    {
        int r = 0;                       // r is local
        for (int i = 0; i<v.size(); ++i) // i is in statement scope
                r = max(r,abs(v[i]));
        // no i here
        return r;
    }
    // no r here
};
// no v here
```

作用域嵌套

```
int x;      // global variable – avoid those where you can
int y;      // another global variable   "0"

int f()
{
    int x;          // local variable (Note – now there are two x's)
    x = 7;          // local x, not the global x
    {
        int x = y; // another local x, initialized by the global y
                   // (Now there are three x's)   "0"
        x++;       // increment the local x in this scope   "1"
    }
    // x = 7;
}
// x = 0;
```

存在初始化顺序不确定、修改定位困难等问题

本质上，编译器对不同作用域有不同的命名

尽可能避免这么复杂的嵌套：keep it simple!

函数

- 一般形式：
  - **return_type *name* (*formal arguments*);** *// a declaration*
  - **return_type *name* (*formal arguments*) *body*** *// a definition*
  - 例如：

    double f(int a, double d) { return a*d; }
- 形参通常也称作参数
- 如果你不想返回任何值，使用**void** 作为返回类型

  void increase_power(int level);
  - 这里，**void** 意思是“不返回一个值”
- 函数体是一个程序块或try块
  - 例如：

    { */* code */* **}** *// a block*

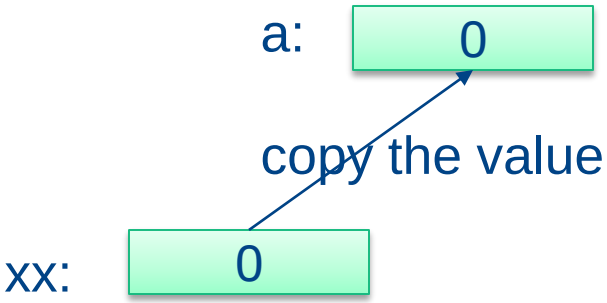
    try { */* code */* **} catch(exception& e) {** */* code */* **}** *// a try block*
- 函数实现(表示)了实际的计算功能

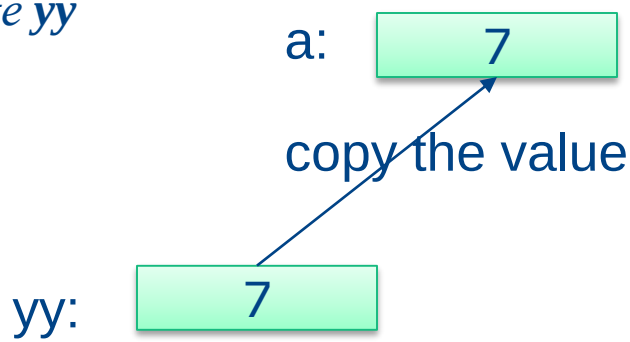
返回值可以看做为初始化的另一种形式

函数：传值

```
// call-by-value (send the function a copy of the argument's value)
int f(int a) { a = a+1; return a; }


int main()
{
    int xx = 0;
    cout << f(xx) << endl;     // writes 1
    cout << xx << endl;        // writes 0; f() doesn't change xx
    int yy = 7;
    cout << f(yy) << endl; // writes 8;  f() doesn't change yy
    cout << yy << endl;        // writes 7
}
```



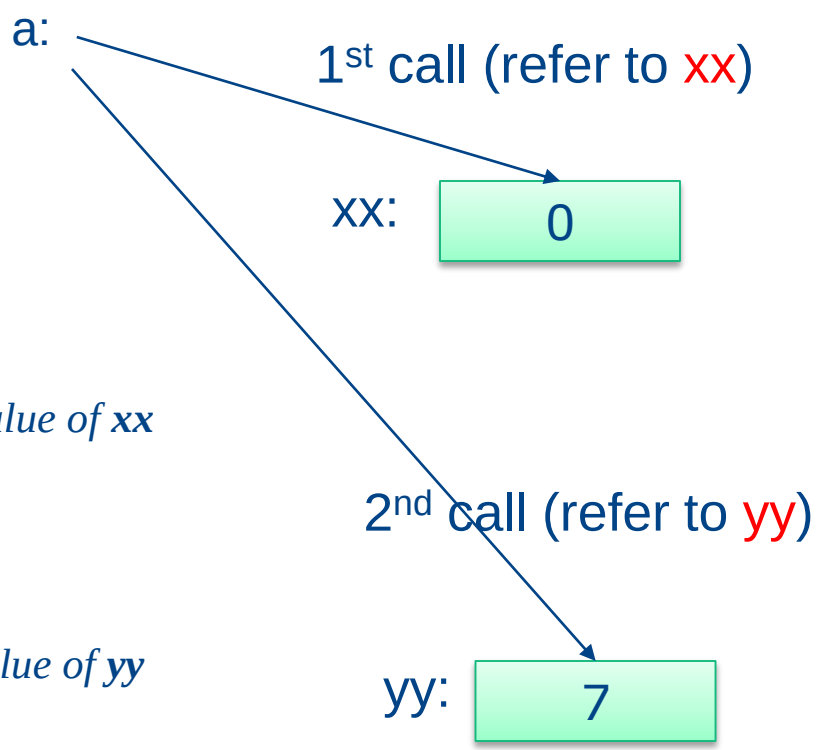


函数：传引用

```
// call-by-reference (pass a reference to the argument)
int f(int& a) { a = a+1; return a; }


int main()
{
    int xx = 0;
    cout << f(xx) << endl;        // writes 1
                                  // f() changed the value of xx
    cout << xx << endl;           // writes 1
    int yy = 7;
    cout << f(yy) << endl; // writes 8
                                  // f() changes the value of yy
    cout << yy << endl;           // writes 8

}
```



引用拷贝代价低，只需一个地址引用即可！

函数

- 尽量避免使用传引用参数 (non-const)
  - 当你忘记那些参数能够修改时，它会导致一些莫名的bugs

```
int incr1(int a) { return a+1; }
void incr2(int& a) { ++a; }
int x = 7;
x = incr1(x);   // pretty obvious
incr2(x);       // pretty obscure
```

- 那为什么还需要引用参数呢?
  - 在一些情况下，它是重要的
    - *e.g.*, 改变多个参数的值
    - 操作容器 (*e.g.*, **vector**)
  - **const** 引用更常用

传值、传引用和传常量引用

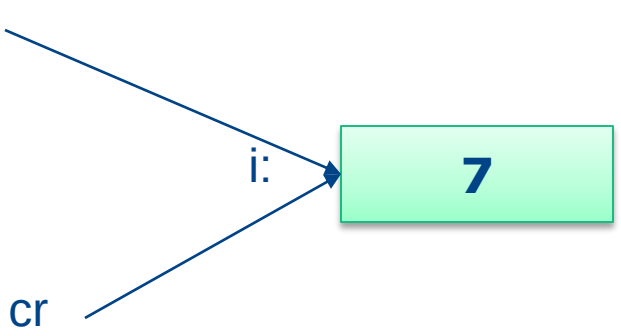
```
void f(int a, int& r, const int& cr) { ++a; ++r; ++cr; } // error: cr is const
void g(int a, int& r, const int& cr) { ++a; ++r; int x = cr; ++x; } // ok

int main()
{
    int x = 0;
    int y = 0;
    int z = 0;
    g(x,y,z);       // x==0; y==1; z==0
    g(1,2,3);       // error: reference argument r needs a variable to refer to
    g(1,y,3);       // ok: since cr is const we can pass "a temporary"
}
```

- const引用更常用，主要用于传递较大的对象

引用

- “引用”是个一般性概念
  - 不仅仅用于函数的传引用




```
int i = 7;
int& r = i;
r = 9;              // i becomes 9
const int& cr = i;
// cr = 7;          // error: cr refers to const
i = 8;
cout << cr << endl;      // write out the value of i (that's 8)
```

- 可以认为引用是一个对象的别名！
- 但你不能
  - 通过一个const 引用来修改一个对象
  - 初始化后再将引用指向另一个对象(赋值操作)

参数传递指导

- 使用传值方式传递较小的对象
- 使用传const引用方式传递你不需要修改的大对象
- 让函数返回一个结果值，而不是修改通过引用传递来的对象(尽量)
- 只在必要的时候才使用传引用调用(非const)

- 例如：

```
class Image { /* objects are potentially huge */ };
void f(Image i);  … f(my_image);   // oops: this could be s-l-o-o-o-w
void f(Image& i); … f(my_image); // no copy, but f() can modify my_image
void f(const Image&); … f(my_image);  // f() won't mess with my_image
```

→ 更多函数参数和实现的内容参看课本的8.5节

名字空间

- 考虑下面来自两个不同程序员 (Jack 和Jill) 的代码

```
class Glob { /*...*/ };          // in Jack's header file jack.h
class Widget { /*...*/ };        // also in jack.h

class Blob { /*...*/ };          // in Jill's header file  jill.h
class Widget { /*...*/ };        // also in jill.h


#include "jack.h";   // this is in your code
#include "jill.h";              // so is this

void my_func(Widget p)           // oops! – error: multiple definitions of Widget
{
  // ...
}
```

变量的重定义

名字空间

- 编译器不能编译通过多重定义的变量
  - 在使用多个头文件时，这种冲突可能会发生
- 避免这种情况的一种有效方式是使用名字空间

```
namespace Jack {                    //  in Jack's header file
    class Glob{ /*...*/ };
    class Widget{ /*...*/ };
}

#include "jack.h";   // this is in your code
#include "jill.h";                  // so is this

void my_func(Jack::Widget p)        // OK, Jack's Widget class will not
{                                   // clash with a different Widget
  // ...
}
```

名字空间

- 名字空间是一个命名的作用域
  - 不需要定义类型就可以拥有一个可访问的作用域
- “::”语法用于指定使用的名字空间名称和对象名称
- 例如：
  - cout 在名字空间 std 中，你可以写成下面的形式：

```
std::cout << "Please enter stuff… \n“;
```

- 本质上
  - 名字空间是让编译器将对象编译为一个不同的id (id依赖于作用域和对象名字)

using 声明和指令

- 为了避免下面的冗长
  - **std::cout << "Please enter stuff… \n";**

  你可以使用一个“using 声明”
  - **using std::cout;** // *when I say **cout**, I mean **std::cout**”*
  - **cout << "Please enter stuff… \n";** // *ok: std::cout*
  - **cin >> x;** // *error: cin not in scope*

- 或者，直接使用“using 指令”
  - **using namespace std;** // *“make all names from namespace **std** available”*
  - **cout << "Please enter stuff… \n";** // *ok: std::cout*
  - **cin >> x;** // *ok: std::cin*

  - 将一个using指令放在头文件中是一个非常坏的习惯！

Next

- 关于类的更多技术细节……