

基于过程抽象的程序设计

- ▶ 人们在设计一个复杂的程序时，经常会用到功能分解和复合两种手段：
 - ▶ 功能分解：在进行程序设计时，首先把程序的功能分解成若干子功能，每个子功能又可以继续分解...，从而形成了一种自顶向下（top-down）、逐步精化（step-wise）的设计过程。
 - ▶ 功能复合：把已有的（子）功能逐步组合成更大的（子）功能，从而形成一种自底向上（bottom-up）的设计过程，实现复杂功能。
-

函数

做甜点

内容回顾

和面

输入：面粉、水、鸡蛋

输出：面团



造型

输入：面包胚子

输出：造型的生面包



配料

输入：面团、配料

输出：面包胚子



烘焙

输入：造型的生面包

输出：香喷喷的面包



C/C++函数

- ▶ 函数是C/C++提供的用于实现子程序的语言成分。

- ▶ 函数的定义：

<返回值类型> **<函数名>** (**<形式参数表>**)

{

<函数体>

}

- ▶ **<函数名>**：用于标识函数的名字，用标识符表示
- ▶ (**<形式参数表>**)：描述函数的形式参数，由0,1或多个形参说明（用逗号隔开）构成，形参说明的格式为：

 <类型> <形参名>

- ▶ **<返回值类型>**：描述了函数返回值的类型

- ▶ 可以为任意的C++数据类型

- ▶ 当返回值类型为void时，它表示函数没有返回值

函数调用的执行过程

- ▶ 计算实参的值；
- ▶ 把实参分别传递给被调用函数的形参；
- ▶ 执行函数体；
- ▶ 函数体中执行return语句返回函数调用点，调用点获得返回值（如果有返回值）并执行调用之后的操作。



函数声明

- ▶ 程序中调用的所有函数都要有定义；如果函数定义在本源文件中使用点之后或在其它文件（如：C++的标准库）中，则在调用前需要对被调用的函数进行声明。

- ▶ 函数声明的格式如下：

`extern <返回值类型> <函数名>(<形式参数表>; //函数原型`



函数的参数传递

- ▶ C++提供了两种参数传递机制：
 - ▶ 值传递
 - ▶ 把实参的值赋值给形参。
 - ▶ 地址或引用传递
 - ▶ 把实参的地址赋值给形参。
- ▶ C++默认的参数传递方式是值传递。



希望做更多的课外习题？

OJ 系统刷题

- ▶ <http://poj.org/>
- ▶ <http://www.nowcoder.com/ta/acm-training>
- ▶ <http://acm.njupt.edu.cn/welcome.do;jsessionid=1002F0F56453CF96A926534F6780B65D?method=index> (中文)

前题：又好又快完成

课程作业和课本所有练习题的基础上！



例程：求1~10自然数的平方和

分析下面的程序是否有语法错误

... ..

```
int main()
```

```
{
```

```
    int i, num=0;
```

```
    int temp;
```

//有效期：从定义至main函数结束

```
    for (i=1; i<11; i++)
```

```
    {
```

```
        // int temp;
```

//有效期：从定义至本次循环结束

```
        temp = i*i;
```

```
        num += temp;
```

```
    }
```

```
    cout<< "The square of 10 is " <<temp<< ".\n" ; // Error
```

```
    cout<< "The sum is" <<num<<endl ;
```

```
    ... ..
```

```
}
```


4.3 标示符作用域和 变量生存期

郭 延 文

2019级计算机科学与技术系

变量的局部性

- 在C++中，根据变量的定义位置，把变量分成：局部变量和全局变量：
 - ▶ **局部变量**是指在复合语句中定义的变量，它们只能在定义它们的复合语句（包括内层的复合语句）中使用
例如上述例子循环内和外定义的temp
 - ▶ **全局变量**是指在函数外部定义的变量，它们一般能被程序中的所有函数使用（静态的全局变量除外）



程序中的变量

....

全局变量 //一般能被所有函数使用

....

函数 (局部变量) // 复合语句

{

局部变量

}

....

函数 (局部变量) // 复合语句

{

局部变量


}

....

局部变量和全局变量的例子

```
int x=0; //全局变量
void f()
{
    int y=0; //局部变量
    x++;
    y++;
    a++; // Error
}
```

```
int main()
{
    int a=0; //局部变量
    f();
    a++;
    x++;
    y++; // Error
    while (x<10)
    {
        int b=0; //局部变量
        a++;
        b++;
        x++;
    }
    b++; // Error
    return 0;
}
```



变量的生存期（存储分配）

- 把程序运行时一个变量占有内存空间的时间段称为该变量的生存期。
 - ▶ **静态**：从程序开始执行时就进行内存空间分配，直到程序结束才收回它们的空间；**全局变量具有静态生存期**
 - ▶ 具有静态生存期的变量，如果没有显式初始化，系统将把它们初始化成0
 - ▶ **自动**：内存空间在程序执行到定义它们的复合语句（包括函数体）时才分配，当定义它们的复合语句执行结束时，它们的空间将被收回。**局部变量和函数的参数一般具有自动生存期**
 - ▶ **动态**：内存空间在程序中显式地用**new(c++)**操作或**malloc(c)**库函数分配、用**delete(c++)**操作或**free(c)**库函数收回。**动态变量具有动态生存期**

存储类修饰符

- ▶ 在定义局部变量时，可以为它们加上**存储类修饰符**(**auto**, **static**, **register**)来显式地指出其生存期
 - ▶ **auto**：使局部变量具有自动生存期。局部变量的默认存储类为 **auto**。
 - ▶ **static**：使局部变量具有静态生存期。它只在函数第一次调用时进行初始化，以后调用中不再进行初始化，**它的值为上一次函数调用结束时的值**。
 - ▶ **register**：局部变量仍是自动生存期，但由编译程序根据CPU寄存器的使用情况来决定是否存放在寄存器中（未必放于内存）
-

举 例

```
void f()
{   auto int x=0;    //auto一般不写
    static int y=1;
    register int z=0;
    x++; y++; z++;
    cout << x << y << z << endl;
}
// 在main函数中调用f();
```

* 第一次调用f时, 输出: 1 2 1

* 第二次调用f时, 输出: 1 3 1



程序实体在内存中的安排

静态数据区
代码区
栈区
堆区

- ▶ **静态数据区**用于全局变量、static存储类的局部变量以及常量的内存分配；
- ▶ **代码区**用于存放程序的指令，对C++程序而言，代码区存放的是所有二进制代码；
- ▶ **栈区**用于auto存储类的局部变量、函数的形式参数以及函数调用时有关信息（如：函数返回地址等）的内存分配；
- ▶ **堆区**用于动态变量的内存分配
- ▶ **文字常量区**：存放常量字符串，程序结束后由系统释放

栈区和堆区

▶ 栈区 (stack)

由编译器自动分配释放，存放函数的参数值，局部变量的值等，内存的分配是连续的，类似于平时我们所说的栈(可以把它想成数组)，它的内存分配是连续分配的，即:所分配的内存是在一块连续的内存区域内。当我们声明变量时，那么编译器会自动接着当前栈区的结尾来分配内存

▶ 堆区 (heap)

一般由程序员分配释放，若程序员不释放，程序结束时可能由操作系统回收。类似于链表，在内存中的分布不是连续的，它们是不同的内存块通过指针链接起来的

例程：求1~10自然数的平方和

分析下面的程序是否有语法错误

... ..

```
int main()
```

```
{
```

```
    int i, num=0;
```

```
    int temp;
```

//有效期：从定义至main函数结束

```
    for (i=1; i<11; i++)
```

```
    {
```

```
        // int temp;
```

//有效期：从定义至本次循环结束

```
        temp = i*i;
```

```
        num += temp;
```

```
    }
```

```
    cout<< "The square of 10 is " <<temp<< ".\n" ; // Error
```

```
    cout<< "The sum is" <<num<<endl ;
```

```
    ... ..
```

```
}
```

与生存期“对应”的
一个概念是作用域



标识符的作用域

- ▶ 标识符的作用域:
 - ▶ 标识符的有效范围（能被访问的程序段）称为该标识符的作用域。
- ▶ 在不同的作用域中，可以用相同的标识符来标识不同的程序实体。



生存期 V. S. 作用域

属性

生存期

变量占有内存空间的时间段

“时间”上的概念

作用域

标识符的有效范围

从“代码空间（篇幅）”上讲



... ..

```
int main()
{
    int i, num=0;
    int temp;
    for (i=1; i<11; i++)
    {
        // int temp;
        temp = i*i;
        num += temp;
    }
}
...
```

C++标识符的作用域

▶ C++把标识符的作用域分成若干类，其中包括：

- ▶ 局部作用域
- ▶ 全局作用域
- ▶ 文件作用域
- ▶ 函数作用域
- ▶ 函数原型作用域
- ▶ 类作用域
- ▶ 名空间作用域



局部作用域

- ▶ 在函数定义或复合语句中、从标识符的定义点开始到函数定义或复合语句结束之间的程序段
- ▶ C++中的局部常量名、局部变量名/对象名以及函数的形参名具有局部作用域。

读程序

分析涉及变量x, n的语句
是否有效?

```
void f(int n)
{  x++;          // Error
  int x=0;
  x++;
  n++;
  .....
}
void g()
{  x++;          // Error
  n++;          // Error
}
int main()
{  int x=0;
  int n;
  cin >> n;
  f(n);
  .....
}
```

- ▶ 如果在一个标识符的局部作用域中包含内层复合语句，并且在该内层复合语句中定义了一个同名的不同实体，则外层定义的标识符的作用域应该是从其潜在作用域中扣除内层同名标识符的作用域之后所得到的作用域

```
void f()
```

```
{  int x;  //外层x的定义
```

```
    ... x ... //外层的x
```

```
    while ( ... x ... ) //外层的x
```

这样的写法并不推荐!!!

```
    { ... x ...          //外层的x
```

```
        double x;      //内层x的定义
```

```
        ... x ...      //内层的x
```

```
    }
```

```
    ... x ...          //外层的x
```

```
}
```


全局作用域

- ▶ 在函数级定义的标识符具有全局作用域。
- ▶ 全局变量名/对象名、全局函数名和全局类名的作用域一般具有全局作用域，它们在整个程序中可用。
- ▶ 如果在某个局部作用域中定义了与某个全局标识符同名的标识符，则该全局标识符的作用域应扣掉与之同名的局部标识符的作用域。
- ▶ 在局部标识符的作用域中若要使用与其同名的全局标识符，则需要用全局域选择符 (::) 对全局标识符进行修饰（受限）。

```
double x; //外层x的定义
void f()
{
    int x;    //内层x的定义
    ... x ... //内层的x
    ... ::x ... //外层的全局x
}
```



文件作用域

- ▶ 在全局标识符的定义中加上 **static** 修饰符，则该全局标识符就成了具有文件作用域的标识符，它们只能在定义它们的源文件（模块）中使用。
- ▶ C++ 中的关键词 **static** 有两个不同的含义。
 - ▶ 在局部变量的定义中，static 修饰符用于指定局部变量采用静态存储分配；
 - ▶ 而在全局标识符的定义中，static 修饰符用于把全局标识符的作用域改变为文件作用域。
- ▶ 一般情况下，具有全局作用域的标识符主要用于标识被程序各个模块共享的程序实体，而具有文件作用域的标识符用于标识在一个模块内部使用的程序实体。

//file1.cpp

static int y; //文件作用域!!!

static void f() //文件作用域!!!

{
}

//file2.cpp

extern int y;

extern void f();

void g()

{

... y ... // Error

f(); // Error

}




函数作用域

- ▶ **语句标号**是唯一具有函数作用域的标识符，在定义它的函数体中的任何地方都可以访问。
- ▶ 函数作用域与局部作用域的区别是：
 - ▶ 函数作用域包括整个函数，而局部作用域是从定义点开始到函数定义或复合语句结束。
 - ▶ 在函数体中，一个语句标号只能定义一次，即使是在内层的复合语句中，也不能再定义与外层相同的语句标号。



```
void f()
{ .....
goto L; // OK
.....
L: ...
.....
{ .....
  L: ... // Error
  .....
}
.....
goto L; // OK
.....
}
void g()
{ .....
  goto L; // Error
  .....
}
```



例 程

```
#include<iostream>
```

```
using namespace std;
```

//名空间

```
int main()
```

```
{
```

```
    double x, y;
```

```
    cout<<"Enter two numbers:";
```

```
    cin>>x>>y;
```

```
    ... ..
```

```
}
```



开学第一天



山东::小明

江苏::小明

名空间作用域

- ▶ 对于一个由多个文件构成的程序，有时会面临一个问题：
在一个源文件中要用到两个分别在另外两个源文件中定义的不同全局程序实体（如：全局函数），而这两个全局程序实体的名字相同
- ▶ C++提供了名空间（**namespace**）设施来解决上述的名冲突问题
 - ▶ 在一个名空间中定义的全局标识符，其作用域为该名空间
 - ▶ 当在一个名空间外部需要使用该名空间中定义的全局标识符时，可用该名空间的名字来修饰或受限

// 模块1

namespace A

{ int x=1;

void f()

{

}

}

// 模块2

namespace B

{ int x=0;

void f()

{

}

}

// 模块3

1、

... A::x ... //A 中的x

A::f(); //A 中的f

... B::x ... //B 中的x

B::f(); //B 中的f

2、

using namespace A;

... x ... //A 中的x

f(); //A 中的f

... B::x ... //B 中的x

B::f(); //B 中的f

3、

using A::f;

... A::x ... //A 中的x

f(); //A 中的f

... B::x ... //B 中的x

B::f(); //B 中的f

再回顾: `using namespace std;`

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    double x, y;
```

```
    cout << "Enter two numbers:";
```

```
    cin >> x >> y;
```

```
    ... ..
```

```
}
```

如果delete “`using namespace std`”，编译会报什么错？

// `cin`, `cout` 没有定义

命名空间std: c++标准库中的类和函数是在命名空间std中声明的，因此程序要用到c++标准库（此时就需#include命令），就需要用“`using name space std;`”作声明，表示要用到命名空间std中的内容。



C++程序的多模块结构

- ▶ **逻辑上**，一个C++程序由一些全局函数（区别于类定义中的成员函数）、全局常量、全局变量/对象以及类的定义构成，其中必须有且仅有一个名字为main的全局函数。函数内部可以包含形参、局部常量、局部变量/对象的定义以及语句。
- ▶ **物理上**，可以按某种规则（对象及实现的功能）对构成C++程序的各个逻辑单位（全局函数、全局常量、全局变量/对象、类等）的定义进行分组，分别把它们放在若干个源文件中，构成**程序模块**。
- ▶ **程序模块**(可单独编译的程序单位)是为了便于从物理上对程序进行组织、管理和理解，便于多人合作开发一个程序。

C++模块的构成

▶ 一个C++模块一般包含两个部分：

▶ 接口（.h文件）：

▶ 给出在本模块中定义的、提供给其它模块使用的一些程序实体（如：函数、全局变量等）的声明；

▶ 实现（.cpp文件）：

▶ 模块的实现给出了模块中的程序实体的定义。



✱ 在模块A中要用到模块B中定义的程序实体时，可以在A的.cpp文件中用文件包含命令（#include）把B的.h文件包含进来。

✱ 文件包含命令是一种编译预处理命令，其格式为：

#include <文件名> 或 #include "文件名"

✱ 含义：在编译前，用命令中的文件名所指定的文件内容替换该命令。

Tips: 前者从编译器的自带的头文件库查找；后者编译时从你当前创建的工程寻找文件，找不到时，再查找编译器的自带的头文件库。



//file1.h

```
extern int x;           //全局变量x的声明
extern double y;        //全局变量y的声明
int f();                //全局函数f的声明
```

//file1.cpp

```
int x=1;                //全局变量x的定义
double y=2.0;           //全局变量y的定义
int f()                 //全局函数f的定义
{ int m;                //局部变量m的定义
    .....
    m += x;             //语句
    .....
    return m;
}
```



//file2.h

void g(); //全局函数g的声明

//file2.cpp

#include "file1.h" //把文件file1.h中的内容包含进来

void g() //全局函数g的定义

{ double z; //局部变量z的定义

.....

z = y+10; //语句

.....

}




```
//main.cpp
```

```
#include "file1.h" //把文件file1.h中的内容包含进来
```

```
#include "file2.h" //把文件file2.h中的内容包含进来
```

```
int main() //全局函数main的定义
```

```
{ double r; //局部变量r的定义
```

```
.....
```

```
r = x+y*f(); //语句
```

```
.....
```

```
g(); //语句
```

```
.....
```

```
}
```



本节内容回顾

变量的

局部性

局部变量 全局变量

生存期

静态 动态 自动

`static`, `auto`, `register`
`using namespace std;`
`::`
`#include "a.h"` 或
`#include <a>`

标识符作用域

局部 全局 文件 函数名空间作用域

C++程序的多模块结构

`.h`, `.cpp`



Q & A



上节内容回顾

➤ 变量的

局部性

全局变量 局部变量

生存期

静态 自动 动态

存储类修饰符

static, auto, register



上节内容回顾： static的作用

- ▶ 使局部变量具有静态生存期
 - ▶ 它只在函数第一次调用时进行初始化，以后调用中不再进行初始化，它的值为上一次函数调用结束时的值
- ▶ 使全局标识符具有文件作用域
 - ▶ 在全局标识符的定义中加上static修饰符，则该全局标识符就成了具有文件作用域的标识符，它们只能在定义它们的源文件（模块）中使用

上节内容回顾：举 例

```
void f()
{  auto int x=0; //auto一般不写
   static int y=1;
   register int z=0;
   x++; y++; z++;
   cout << x << y << z << endl;
}
// 在main函数中调用f();
```

- 第一次调用f时，输出：1 2 1
- 第二次调用f时，输出：1 3 1

