



# 第七章 程序设计内存模型

主讲教师：同济大学计算机科学与技术学院 陈宇飞  
同济大学计算机科学与技术学院 龚晓亮



# 目录

- 变量的存储类别
- 函数和链接性
- 多源程序
- 命名空间



# 目录

- 变量的存储类别

- 基本概念

- 变量属性

- 变量分类

- 变量存储

- 自动变量、寄存器变量、静态局部变量、静态全局变量、外部全局变量

- 变量小结



# 1.1 基本概念

✓ 应用程序执行时的内存分布

程序(代码)区	存放程序的执行代码
静态存储区	程序执行中，变量占固定的存储空间
动态存储区	程序执行中，变量根据需要分配不同位置的存储空间

静态存储区	动态存储区	CPU寄存器
<ul style="list-style-type: none"><li>➤ 外部全局变量</li><li>➤ 静态全局变量</li><li>➤ 静态局部变量</li><li>➤ 常量/常变量</li></ul>	<ul style="list-style-type: none"><li>➤ 自动变量</li><li>➤ 形参</li><li>➤ 堆 (heap)</li></ul>	<ul style="list-style-type: none"><li>➤ 寄存器变量</li></ul>



## 1.2 变量属性

- ✓ 持续性：在什么时间存在，也叫生存期（时间概念）
  - 自动（动态存储）、静态（静态存储）
- ✓ 作用域：在什么范围内可以访问（空间概念）
  - 局部（代码块）、全局（定义位置到文件结尾）
- ✓ 链接性：如何在不同单元间共享
  - 外部（文件间共享）、内部（一个文件中的函数共享）



# 1.3 变量分类

- ✓ 按类型：字符型、整型、浮点型等
- ✓ 按作用域：局部变量、全局变量
- ✓ 按持续性（生存期）：动态存储变量、静态存储变量
- ✓ 按存储位置：内存变量、寄存器变量



# 1.4 变量存储

## ✓ 自动变量:

- ❖ 进入函数（代码块）后，分配空间，结束函数（代码块）后，释放空间
- ❖ 自动变量占动态存储区
- ❖ 若定义时赋初值，自动变量在函数调用时执行，每次调用均重复赋初值
- ❖ 若定义时不赋初值，则自动变量的值不确定
- ❖ 函数的形参同自动变量



# 1.4 变量存储

... //自动变量

```
int main()
```

```
{
```

```
    int texas=31;
```

```
    int year=2011;
```

```
    cout<<"In main(), ...";
```

```
    oil(texas);
```

```
    cout<<"In main(), ...";
```

```
    return 0;
```

```
}
```

texas  
存在

```
void oil(int x)
```

```
{
```

```
    int texas=5;
```

```
    cout<<"In oil(), ...";
```

```
    { //start a block
```

```
        int texas=113;
```

```
        cout<<"In block, ...";
```

```
    }
```

```
    cout<<"Post-block...";
```

```
}
```

texas  
存在

若全局变量与局部变量同名，按“低层屏蔽高层”的原则处理





# 1.4 变量存储

## ✓ 寄存器变量:

- ❖ 对一些频繁使用的变量，可放入CPU的寄存器中，提高访问速度（CPU访问寄存器比内存快一个数量级）

```
register int a;
```

- ❖ 仅对自动变量和形参有效
- ❖ 编译系统会自动判断(即使定义了register，最终是否放入寄存器中，仍需要编译系统决定)



# 1.4 变量存储

□静态局部变量

□静态全局变量

□外部全局变量

```
int global = 1000;           //静态持续性, 外部链接性
static int one_file = 50;    //静态持续性, 内部链接性
int main()
{
    ...
}
void funct1(int n)
{
    static int count = 0;    //静态持续性, 无链接性
    int llama = 0;
    ...
}
void funct2(int q)
{
    ...
}
```



# 1.4 变量存储

## ✓ 静态局部变量

- ❖ 变量所占存储单元在程序的执行过程中均不释放
- ❖ 静态局部变量占静态存储区
- ❖ 静态局部变量在第一次调用时执行，以后每次调用不再赋初值，保留上次调用结束时的值
- ❖ 若定义时不赋初值，则静态局部变量的值为0（'\0'）
- ❖ 无链接性



# 1.4 变量存储

```
#include <iostream>
using namespace std;
int f(int n)
{
    int fac=1;
    return fac*=n;
}
int main()
{
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n",i, f(i));
    return 0;
}
```

```
1!=1
2!=2
3!=3
4!=4
5!=5
```

## 自动变量

- 1、fac的分配/释放重复了5次
- 2、5次的fac不保证分配同一内存空间
- 3、fac只在f()内部可被访问

```
#include <iostream>
using namespace std;
int f(int n)
{
    static int fac=1;
    return fac*=n;
}
int main()
{
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n",i, f(i));
    return 0;
}
```

```
1!=1
2!=2
3!=6
4!=24
5!=120
```

## 静态局部变量

- 1、fac在编译时已分配了空间，程序结束时释放
- 2、每次进入f()，fac都是同一空间
- 3、fac在f()内部可被访问，在f()外不能访问(但存在)

希望将一个函数前一次调用的结果带到下一次调用中时，可用静态局部变量



# 1.4 变量存储

## ✓（外部/静态）全局变量

- ❖ 从定义点到源文件结束之间的所有函数均可使用，并可以通过extern扩展作用范围
- ❖ 外部全局变量：所有源程序文件中的函数均可使用（外部链接性）
- ❖ 静态全局变量：只限本源程序文件的定义范围内使用（内部链接性）（static）
- ❖ 两者均在静态数据区中分配，不赋初值则自动为0（'\0'）



# 1.4 变量存储

## ✓ 单定义规则（ODR）

❖ 变量只能有一次定义

## ✓ C++提供两种变量声明

❖ 定义：给变量分配存储空间；也叫定义声明

❖ 声明：不给变量分配存储空间，因为它引用已有的变量，也

叫引用声明；使用关键词extern，且不进行初始化

```
double up;           //definition, up is 0
extern int blem;      //blem defined elsewhere
extern char gr = 'z';
                    //definition because initialized
```



# 1.4 变量存储

## ✓ extern不分配存储空间

变量process\_status为外部链接性  
//file1.cpp  
...  
//defining an external variable  
int process\_status=0; //分配4字节  
  
void promise()  
{  
 ...//process\_status可访问  
}  
  
int main()  
{  
 ...//process\_status可访问  
}

//file2.cpp  
...  
//referencing an external variable  
extern int process\_status; //不分配空间  
  
int manipulate(int n)  
{  
 ... //process\_status可访问  
}  
  
char \* remark(char \* str)  
{  
 ... //process\_status可访问  
}

删除此句: extern int process\_status  
则文件file2内均不可访问process\_status



# 1.4 变量存储

✓ extern不分配存储空间

```
变量process_status为外部链接性
//file1.cpp
...
//defining an external variable
int process_status=0;          //分配4字节

void promise()
{
    ...//process_status可访问
}

int main()
{
    ...//process_status可访问
}
```

```
//file2.cpp
...
//referencing an external variable

int manipulate(int n)
{
    extern int process_status;
    ... //process_status可访问
}

char * remark(char * str)
{
    ... //process_status不可访问
}
```





# 1.4 变量存储

✓ 局部变量可隐藏同名的全局变量

```
//external.cpp
...
double warming = 0.3;
...

int main()
{
    ...
    update(0.1);
    ...
    local();
    ...
}
```

```
//support.cpp
...
extern double warming;
void update(double dt)
{
    extern double warming; //optional redeclaration
    ...
}
void local()
{
    double warming = 0.8; //new variable hides external one
    ...
    ::warming    /作用域解析运算符，只有C++方式有效，C不可用
}
```



# 1.4 变量存储

## ✓ 静态全局变量隐藏常规全局变量

```
//file1.cpp  
int errors = 20; //external declaration  
...
```

```
//file2.cpp  
int errors = 5; //know to file2 only?  
void froobish()  
{  
    cout << errors; //fails  
    ...  
}
```

违反了单定义规则



```
//file1.cpp  
int errors = 20; //external declaration  
...  
//file2.cpp  
static int errors = 5; //know to file2 only  
void froobish()  
{  
    cout << errors;  
    //uses errors defined in file2  
    ...  
}
```

static errors的链接性为内部



# 1.4 变量存储

✓ 链接性为外部和内部的变量

```
//twofile1.cpp
...
int tom = 3;      //external linkage
int dick = 30;    //external linkage
static int harry = 300; //internal linkage
...

int main()
{
    ...
    remote_access();
    ...
}
```

```
//twofile2.cpp
...
extern int tom;      //tom defined elsewhere
static int dick = 10; //overrides external dick
int harry = 200; //external variable definition,
                //no conflict with twofile1 harry

void remote_access()
{
    ...
}
```

这两个文件使用了同一个tom变量，不同的dick和harry变量  
Tips: 可以从变量地址进行区分



# 1.5 变量小结

类型	持续性（生存期）	作用域	链接性	存储区	初始化
自动变量	自动（本函数）	代码块	无	动态数据区	不确定
		（本函数）			
形参	自动（本函数）	代码块	无	动态数据区	不确定
		（本函数）			
寄存器	自动（本函数）	代码块	无	CPU的寄存器	不确定
		（本函数）			
静态局部	静态（程序执行中）	代码块	无	静态数据区	0（'\0'）
		（本函数）			
静态全局	静态（程序执行中）	文件	内部	静态数据区	0（'\0'）
		（本源程序文件）			
外部全局	静态（程序执行中）	文件	外部	静态数据区	0（'\0'）
		（全部源程序文件）			



# 目录

- 函数和链接性
  - 内部函数和外部函数
  - C++查找函数



## 2.1 内部函数和外部函数

- ✓ 内部函数：函数的链接性为内部，**仅在本文件中可见**

不同的文件中可以定义同名的函数

**static** 返回类型 函数名（形参表） //静态函数

- ✓ 外部函数：默认情况下，函数链接性为外部，即可以在文件间共享

其它文件中加函数说明（可以加extern，也可以不加）

//非静态函数



## 2.2 C++查找函数

- ✓ 函数是静态的：编译器只在该文件中查找函数
- ✓ 函数非静态的：编译器（包括链接程序）在所有程序文件中查找

❖ 找到两个定义：编译器报错

❖ 没有找到：编译器在库中搜索 →

如果定义了一个与库函数同名的函数：

编译器使用程序员定义  
的版本！



## 2.2 C++查找函数

```
//file1.cpp
```

```
...
```

```
int main()
```

```
{
```

```
    extern int max(int, int); //int max(int, int);
```

```
    ...
```

```
    cout<< max(a,b) <<endl;
```

```
    ...
```

```
}
```

```
//file2.cpp
```

```
int max (int x, int y)
```

```
{          //外部链接性
```

```
    ...
```

```
}
```

外部函数可以在文件间共享，其它文件中加  
函数说明（可以加extern，也可以不加）

```
//file1.cpp
```

```
...
```

```
static int max (int x, int y)
```

```
{
```

```
    ...
```

```
}
```

不同的源程序文件中的内部函数可以同名

```
//file2.cpp
```

```
...
```

不可调用/声明  
任何max函数

```
//file3.cpp
```

```
...
```

```
static void max (int x, int y)
```

```
{
```

```
    ...
```

```
}
```





## 2.2 C++查找函数

```
//file1.cpp
```

```
float f2();
```

```
main()
```

```
{ f2();
```

```
}
```

```
float f2()
```

```
{...}
```

```
//file2.cpp
```

```
static float f2();
```

```
int f3()
```

```
{ f2();
```

```
}
```

```
float f2()
```

```
{...}
```

```
//file3.cpp
```

```
extern float f2();
```

```
int f4()
```

```
{ f2();
```

```
}
```

//正确



## 2.2 C++查找函数

```
//file1.cpp
```

```
float f2();
```

```
main()
```

```
{ f2();
```

```
}
```

```
float f2()
```

```
{...}
```

```
//file2.cpp
```

```
float f2();
```

```
int f3()
```

```
{ f2();
```

```
}
```

```
float f2()
```

```
{...}
```

```
//file3.cpp
```

```
extern float f2();
```

```
int f4()
```

```
{ f2();
```

```
}
```

//错误：非静态，找到两个定义

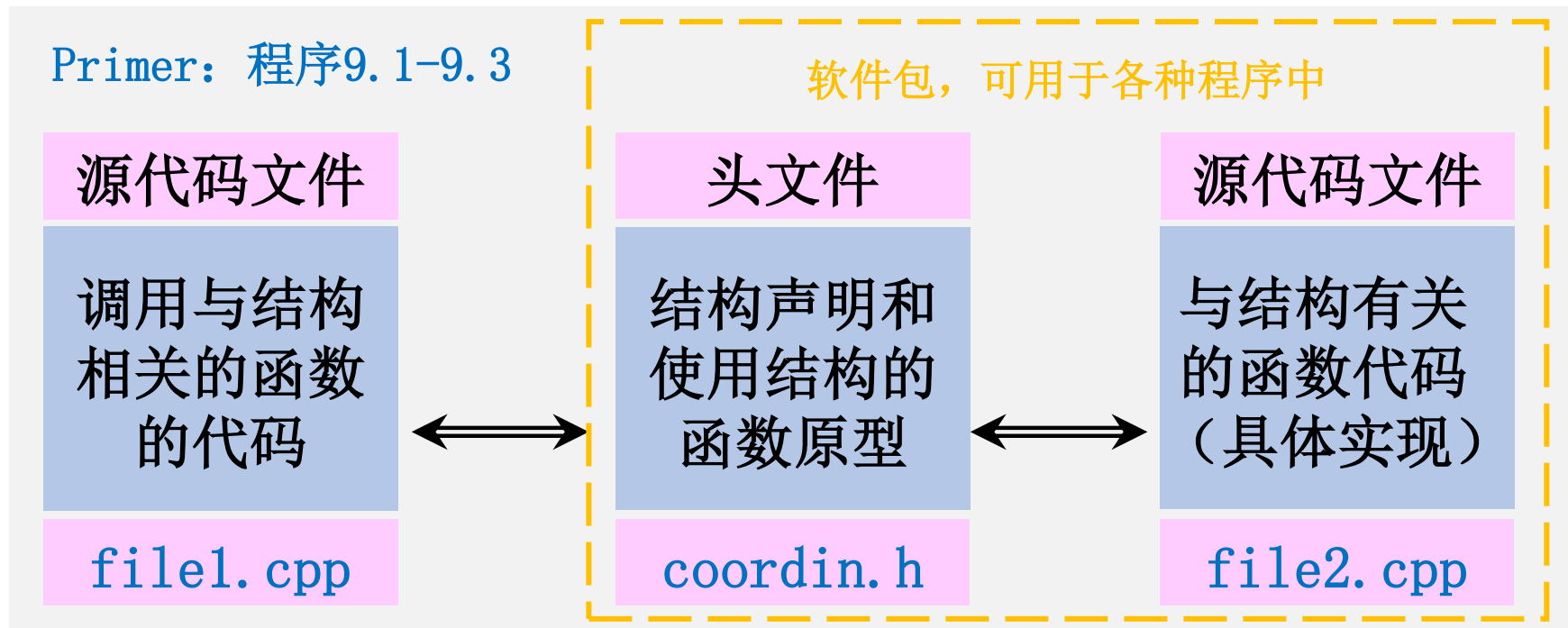


# 目录

- 多源程序
  - 头文件的引入
  - 头文件的管理
  - 头文件命名约定
  - 头文件的内容
  - 头文件示例
  - 头文件的包含方式

## 3.1 头文件的引入

- 程序的代码组织策略（结构为后续知识点，重点了解组织策略）

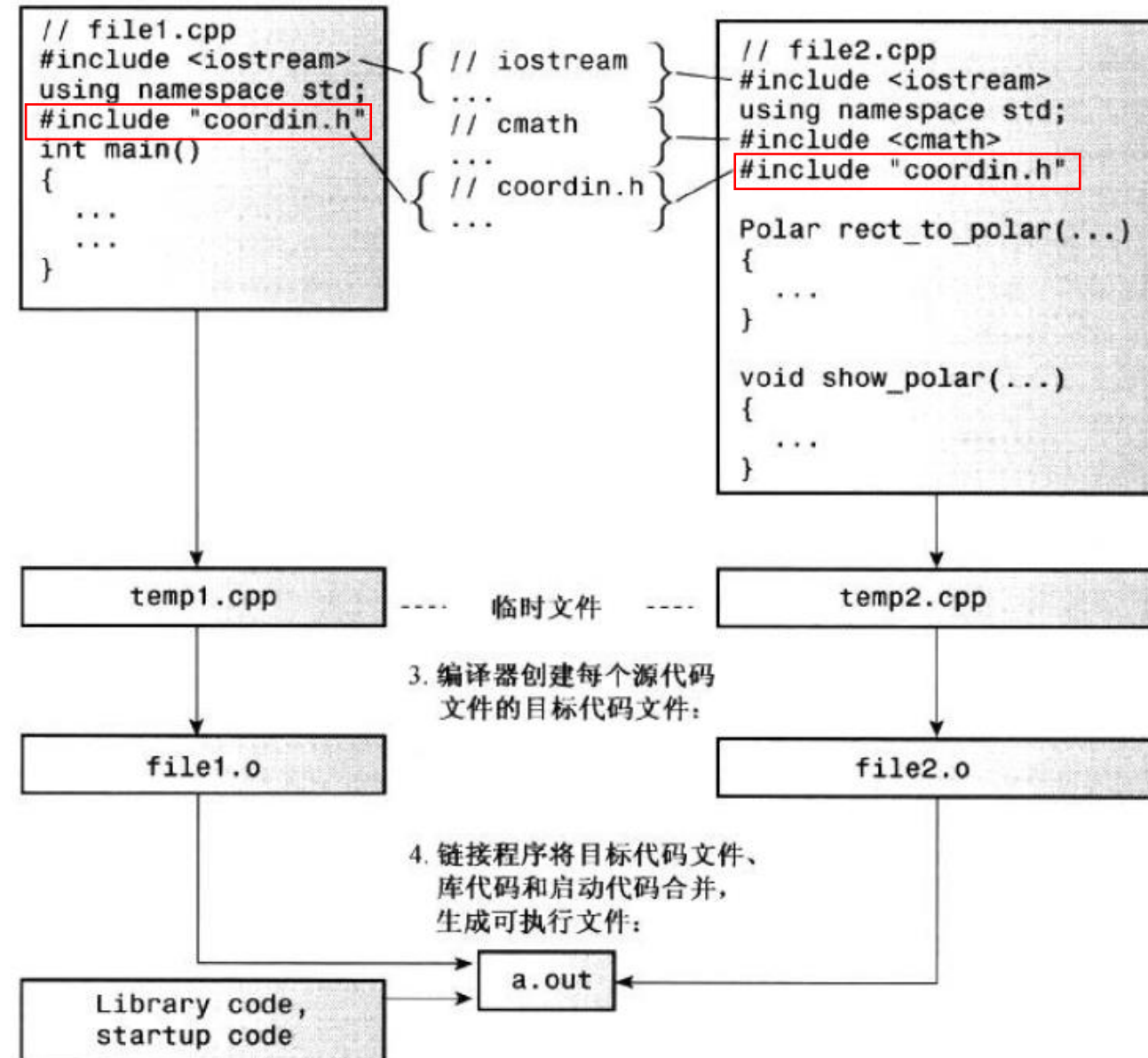


- 将在不同源程序文件中的各种信息归集在一起，方便多次调用和集中修改



## 3.2 头文件的管理

- 在源程序文件中包含头文件时，头文件的所有内容会被理解为包含到 `#include` 位置处
- 只需将源代码文件加入到项目中，而不用加入头文件





## 3.2 头文件的管理

- 在同一个文件中只能将同一个头文件包含一次
  - 解决办法：使用基于预处理器编译指令`#ifndef` (if not defined)

```
//coordin.h  
  
#ifndef COORDIN_H_  
  
#define COORDIN_H_  
  
//place include file contents here  
  
#endif
```



## 3.3 头文件命名约定

头文件类型	约定	示例	说明
C++旧式风格	以.h结尾	iostream.h	C++程序可以使用
C旧式风格	以.h结尾	math.h	C、C++程序可以使用
C++新式风格	没有扩展名	iostream	C++程序可以使用，使用 namespace std
转换后的C	加上前缀c， 没有扩展名	cmath	C++程序可以使用，可以使用不 是C的特性，如namespace std



## 3.4 头文件的内容

- ❖ 函数原型
  - ❖ 使用#define或const定义的符号常量
  - ❖ 结构声明
  - ❖ 类声明
  - ❖ 模板声明
  - ❖ 内联函数 (inline)
- } 后续内容，现阶段了解即可





## 3.5 头文件示例（函数原型）

```
//test.cpp
...
#include "add.h"
int main()
{
    int i = 1, j = 2;
    cout<<"i+j="<<add(i, j)<<endl;
    return 0;
}
```

//直接调用add函数

```
//add.h
#ifndef ADD_H_
#define ADD_H_
int add(int a, int b);
#endif
```

通过头文件使维护  
简单，避免多处修  
改导致的不一致性

//只声明：函数的原型  
//不关心：函数的实现

```
//add.cpp
...
#include "add.h"
int add(int a, int b)
{
    return a+b;
}
```

//函数的实现



## 3.5 头文件示例（符号常量）

```
//test1.cpp
...
#include "headtest.h"
int main()
{
    ...
    PI...
    ...
}
```

//使用了PI

```
//headtest.h
#ifndef HEADTEST_H_
#define HEADTEST_H_
#define PI 3.14
...
#endif
```

通过头文件使维护简单，避免多处修改导致的不一致性

//定义了符号常量

```
//test2.cpp
...
#include "headtest.h"
double circleArea(double r)
{
    ...
    PI...
    ...
}
```

//使用了PI



## 3.6 头文件的包含方式

✓ #include <文件名>:

❖ 直接到系统目录中寻找，找到则包含进来，找不到则报错

以本机64位OS+VS2###为例：C:\Program Files(x86)\Microsoft Visual Studio\2###\Community\VC\Tools\MSVC\14.24.28314\include

✓ #include "文件名":

❖ 先在当前目录中寻找，找到则包含进来，找不到则再到系统目录中寻找，找到则包含进来，找不到则报错



## 3.6 头文件的包含方式

- 文件包含<>和""的差别

当前目录下的demo.h

```
int a=10;
```

系统目录下的demo.h

```
int b=10;
```

```
//demo.cpp
#include <iostream>
using namespace std;
#include <demo.h>
int main()
{
    cout << a << endl;
    return 0;
}
```

//编译报错: <>寻找的是系统目录, 找到的demo.h无a的定义

```
//demo.cpp
#include <iostream>
using namespace std;
#include "demo.h"
int main()
{
    cout << b << endl;
    return 0;
}
```

//编译报错: ""寻找的是当前目录, 找到的demo.h无b的定义



# 目录

- 命名空间
  - 传统C++命名空间
  - 新的命名空间
  - 命名空间示例
  - 命名空间及其前途



## 4.1 传统C++命名空间

✓**声明区域 (declaration region)** 是可以在其中进行声明的区域。

可以在函数外面声明全局变量，其声明区域为其声明所在的文件

✓**域 (scope)** 是变量对程序而言可见的范围

✓**潜在作用域 (potential scope)**，变量的潜在作用域从声明点开始，到其声明区域的结尾。由于变量必须定义后才能使用，因此潜在作用域比声明区域小

✓C++关于全局变量和局部变量的规则定义了命名空间。每个声明区域都可以声明名称，这些名称独立于在其他声明区域中声明的名称



声明区 (全局命名空间)

```
#include<iostream>
using namespace std;
```

```
void orp(int);
int ro = 10;
int main()
{
```

声明区 (代码块)

```
    int  goo;
```

```
    for (int i = 0; i < ro; i++)
    {
```

```
        int temp = 0;
        ++
```

```
        int goo = temp * i;
        ++
```

```
    }
    return 0;
```

```
void orp(int ex)
{
```

```
    int m;
```

```
    ...
    {
```

```
        int ro = 2;
        ++
```

```
    }
    ...
}
```

声明区 (代码块)

声明区 (代码块)

声明区 (代码块)

声明区域



```
#include<iostream>
using namespace std;
```

```
void orp(int);
int ro = 10;
int main()
{
```

```
    int  goo;
```

```
    for (int i = 0; i < ro; i++)
    {
```

```
        int temp = 0;
        ++
```

```
        int goo = temp * i;
        ++
```

```
    }
```

```
    return 0;
```

```
}
void orp(int ex)
{
```

```
    int m;
```

```
    ...
    {
```

```
        int ro = 2;
        ++
```

```
    }
```

```
    ...
}
```

ro的潜在作用域

ro的作用域

goo的作用域

goo的潜在作用域

潜在作用域和作用域





## 4.2 新的命名空间

✓C++新增，通过定义一种新的声明区域来创建命名的命名空间，提供一个声明命名的区域

✓一个命名空间中的命名不会与另一个命名空间的相同名称发生冲突，同时允许程序的其他部分使用该命名空间中声明的东西

✓关键字namespace创建命名空间

```
namespace Jill{  
    double bucket(double n){...}; // function definition  
    double fetch; // variable declaration  
    int pal;      // variable declaration  
    struct Well{...}; // structure declaration  
}
```

```
namespace Jack{  
    double pail; // variable declaration  
    void fetch(); // function prototype  
    int pal;     // variable declaration  
}
```



## 4.2 新的命名空间

- ✓除了用户定义的命名空间外，还存在全局命名空间 (global namespace)。它对应于文件级声明区域，因此全局变量被描述为位于全局命名空间中
- ✓任何命名空间中的命名都不会与其他命名空间发生冲突
- ✓命名空间中的声明和定义规则同全局声明和定义规则相同
- ✓命名空间是开放的 (open)，可以把命名加入到已有命名空间中

```
namespace Jill{  
    char * goose(const char *) //将名称goose添加到Jill中已有的名称列表中  
}
```



## 4.2 新的命名空间

✓访问命名空间中的命名的方法，通过作用域解析运算符::

```
Jack::pail = 12.34; //use a variable  
Jill::Hill mole;   // create a type Hill structure  
Jack::fetch();     // use a function
```

- ✓未被装饰的命名（如pail）称为未限定的命名（unqualified name）
- ✓包含命名空间的命名（如Jack::pail）称为限定的命名（qualified name）



## 4.2 新的命名空间

✓ using声明和using编译指令

❖ using声明使特定的标识符可用，由被限定的命名和它前面的关键词using组成，将特定的名称添加到它所属的声明区域中

```
namespace Jill{
    double fetch;
}
char fetch;
int main()
{
    using Jill::fetch; // put fetch into local namespace
    double fetch; // Error! Already have a local fetch
    cin >> fetch; // read a value into Jill::fetch
    cin >> ::fetch; // read a value into global fetch
    ...
}
```



## 4.2 新的命名空间

✓ using声明和using编译指令

❖ 在函数外使用using声明，将把命名添加到全局命名空间中

```
void other();
namespace Jill{
    double bucket (double n){...}
    double fetch;
    struct Hill {...}
}
using Jill::fetch; // put fetch into global namespace
int main()
{
    cin >> fetch;    // read a value into Jill::fetch
    other()

    ...
}
void other ()
{
    cout << fetch;  // display Jill::fetch

    ...
}
```



## 4.2 新的命名空间

✓ using声明和using编译指令

❖using声明使一个名称可用，而using编译指令使整个命名空间可用。由关键字using namespace组成，它使得命名空间中所有的命名都可用，不需要使用作用域解析

```
#include<iostream> // place names in namespace std
using namespace std; // make names available globally
int main()
{
    using namespace Jack; // make names available in main()
}
```



## 4.2 新的命名空间

✓ using声明和using编译指令

❖在代码中使用作用域解析运算符，不会存在二义性

```
// 不存在二义性
```

```
Jack::pal = 3;
```

```
Jill::pal = 10;
```

```
//上述两个变量是不同的标识符，表示不同的内存单元
```

```
using Jack::pal;
```

```
using Jill::pal;
```

```
pal = 4; //which one? Now have a conflict
```

```
//编译器不允许同时使用using声明，这将导致二义性
```




## 4.2 新的命名空间

### ✓ using声明和using编译指令的比较

- ❖ 使用using编译指令导入一个命名空间中所有的命名与使用多个using声明是不一样的
- ❖ 如果某个命名已经在函数中声明了，则不能用using声明导入相同的命名
- ❖ 如果使用using编译指令导入一个已经在函数中声明的命名，则局部命名将隐藏命名空间名，就像隐藏同名的全局变量一样





```
namespace Jill {
    double bucket(double n) { ... }
    double fetch;
    struct Hill { ... };
}

char fetch; //global namespace
void main() {
    using namespace Jill; // import all namespace names
    Hill Thrill;          // create a type Jill::Hill structure
    double water = bucket(2); // use Jill::bucket()
    double fetch;         // not an error; hide Jill::fetch with local fetch
    cin >> fetch;          // read a value into the local fetch
    cin >> ::fetch;         // read a value into global fetch
    cin >> Jill::fetch;     // read a value into Jill::fetch
    ...
}

int foom()
{
    Hill top;             //error: Hilltop is not defined in the global namespace
    Jill::Hill crest;     // OK
}
```



## 4.2 新的命名空间

**注意：** 假设命名空间和声明区域定义了相同的命名

- ❖ 如果试图使用using声明将命名空间的命名导入该声明区域，则这两个命名会发生冲突从而出错
- ❖ 如果使用using编译指令将该命名空间的命名导入该声明区域，则局部版本将隐藏命名空间版本



## 4.2 新的命名空间

✓ using声明和using编译指令的比较

❖ 一般来说，**使用using声明比使用using编译指令更安全**。using

声明它只导入指定的名称。如果该命名与局部名称发生冲突，编译器会发出指示

❖ using编译指令导入所有命名，包括可能并不需要的命名。如果与局部命名发生冲突，则局部命名将覆盖命名空间版本，而编译器并不会发出警告



## 4.2 新的命名空间

✓ using声明和using编译指令的比较

❖命名空间的开放性，意味着命名空间的命名可能分散在多个地方，这使得难以知道添加了哪些命名

❖有多种选择，既可以使用解析运算符，也可以使用using声明

```
int x;  
std::cin >> x;  
std::cout << x << std::endl;
```



```
using std::cin;  
using std::cout;  
using std::endl;  
int x;  
cin >> x;  
cout << x << endl;
```



## 4.2 新的命名空间

### ✓ 命名空间的其他特性

#### 1. 可以将命名空间声明进行嵌套

```
namespace elements
{
    namespace fire
    {
        int flame;
        ...
    }
    float water;
}
```

```
using namespace elements::fire
```

```
namespace myth
{
    using Jill::fetch;
    using namespace elements;
    using std::cout;
    using std::cin;
}

std::cin >> myth::fetch;
std::cout << Jill::fetch;
```



## 4.2 新的命名空间

### ✓ 命名空间的其他特性

#### 2. using编译指令是可传递的

- ❖ 如果A op B且B op C, 则A op C, 则说操作op是可传递的
- ❖ 例如 > 运算符是可传递的: 如果A > B且B > C, 则A > C

```
namespace myth
{
    using Jill::fetch;
    using namespace elements;
    using std::cout;
    using std::cin;
}
```

```
using namespace myth;
```

等价

```
using namespace myth;
using namespace elements;
```



## 4.2 新的命名空间

### ✓ 命名空间的其他特性

#### 3. 可以给命名空间**创建别名**

❖ 例如:

```
namespace my_very_favorite_things {...};  
namespace mvft = my_very_favorite_things; //创建别名
```

❖ 可以使用这种技术来简化对嵌套命名空间的使用:

```
namespace MEF = myth::elements::fire;  
using MEF::flame
```



## 4.2 新的命名空间

### ✓ 命名空间的其他特性

#### 4. 未命名的命名空间

❖ 可以通过省略命名空间的命名来创建未命名的命名空间

❖ 在该命名空间中声明的名称的潜在作用域为：从声明点到该声明区域末尾。它们与全局变量相似

```
namespace // unnamed namespace
{
    int ice;
    int bandycoot;
}
```





## 4.2 新的命名空间

### ✓ 命名空间的其他特性

#### 4. 未命名的命名空间

❖ 由于这种命名空间没有名称，因此，不能显式地使用using编译器指令或using声明来使它在其他位置可以使用

```
namespace // unnamed namespace
{
    int ice;
    int bandycoot;
}
```



## 4.2 新的命名空间

### ✓ 命名空间的其他特性

#### 4. 未命名的命名空间

❖ 不能在未命名的命名空间所属文件之外的其他文件中，使用该命名空间中的命名。这提供了链接性为内部的静态变量的替代品

```
static int counts;  
int other();  
int main()  
{...}  
int other()  
{...}
```

```
namespace  
{  
    int counts;  
}  
int other();  
int main() {...}  
int other() {...}
```



## 4.3 命名空间示例

```
// namesp. h
#include <string>
// create the pers and debts namespaces
namespace pers
{
    struct Person
    {
        std::string fname;
        std::string lname;
    };
    void getPerson(Person &);
    void showPerson(const Person &);
}
```

```
namespace debts
{
    using namespace pers;
    struct Debt
    {
        Person name;
        double amount;
    };

    void getDebt(Debt &);
    void showDebt(const Debt &);
    double sumDebts(const Debt ar[], int n);
}
```

```
// namesp.cpp -- namespaces
#include <iostream>
#include "namesp.h"

namespace pers
{
    using std::cout;
    using std::cin;
    void getPerson(Person & rp)
    {
        cout << "Enter first name: ";
        cin >> rp.fname;
        cout << "Enter last name: ";
        cin >> rp.lname;
    }

    void showPerson(const Person & rp)
    {
        std::cout << rp.lname << ", " <<
rp.fname;
    }
}
```

```
namespace debts
{
    void getDebt(Debt & rd)
    {
        getPerson(rd.name);
        std::cout << "Enter debt: ";
        std::cin >> rd.amount;
    }
    void showDebt(const Debt & rd)
    {
        showPerson(rd.name);
        std::cout << ": $" << rd.amount <<
std::endl;
    }
    double sumDebts(const Debt ar[], int n)
    {
        double total = 0;
        for (int i = 0; i < n; i++)
            total += ar[i].amount;
        return total;
    }
}
```



```
// usenmsp.cpp -- using namespaces part1
#include <iostream>
#include "namesp.h"

void other(void);
void another(void);
int main(void)
{
    using debts::Debt;
    using debts::showDebt;
    Debt golf = { {"Benny", "Goatsniff"}, 120.0 };
    showDebt(golf);
    other();
    another();
    return 0;
}
```

```
// usenmsp.cpp -- using namespaces part2
```

```
void other(void)
```

```
{
```

```
    using std::cout;
```

```
    using std::endl;
```

```
    using namespace debts;
```

```
    Person dg = {"Doodles", "Glister"};
```

```
    showPerson(dg);
```

```
    cout << endl;
```

```
    Debt zippy[3];
```

```
    int i;
```

```
    for (i = 0; i < 3; i++)
```

```
        getDebt(zippy[i]);
```

```
    for (i = 0; i < 3; i++)
```

```
        showDebt(zippy[i]);
```

```
    cout << "Total debt: $" << sumDebts(zippy, 3) << endl;
```

```
    return;
```

```
} //使用using编译指令导入整个命名空间
```

```
// usenmsp.cpp -- using namespaces part3
```

```
void another(void)
```

```
{
```

```
    using pers::Person;
```

```
    Person collector = { "Milo", "Rightshift" };
```

```
    pers::showPerson(collector);
```

```
    std::cout << std::endl;
```

```
}//使用using声明和作用域解析运算符来访问具体的名称
```



## 4.4 命名空间及其前途

✓ 熟悉了命名空间后，统一的编程理念指导原则

❖ 使用在已命名的命名空间中声明的变量，而不是使用外部全局变量

❖ 使用在已命名的命名空间中声明的变量，而不是使用静态全局变量

❖ 如果开发一个函数库或类库，将其放在一个命名空间中

❖ 仅将编译指令`using`作为一种将旧代码转换为使用命名空间的权宜之计

❖ 不要在头文件中使用`using`编译指令

❖ 导入名称时，首选使用作用域解析运算符或`using`声明的方法

❖ 对于`using`声明，首先将其作用域设置为局部而不是全局



## 4.4 命名空间及其前途

- ✓使用命名空间的主旨是简化大型编程项目的管理工作
- ✓对于只有一个文件的简单程序，使用using编译指令也是可以的
- ✓老式头文件（如`iostream.h`）没有使用命名空间，但新头文件`iostream`使用了std命名空间





# 总结

- 变量的存储类别

- 基本概念
- 变量属性
- 变量分类
- 变量存储

- 函数和链接性

- 内部函数和外部函数
- C++查找函数

- 多源程序

- 头文件的引入
- 头文件的管理
- 头文件命名约定
- 头文件的内容
- 头文件示例
- 头文件的包含方式

- 命名空间

- 传统C++命名空间
- 新的命名空间
- 命名空间示例
- 命名空间及其前途