第四章 函数

模块4.4：变量的访问权限

4.4.1 变量的存储类别

- 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
• 外部全局变量 • 静态全局变量 • 静态局部变量 • 常量/常变量	• 自动变量 • 形参 • 堆（heap）	• 寄存器变量

4.4.1 变量的存储类别

- 变量属性
  - 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）
  - 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）
  - 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）

4.4.1 变量的存储类别

- 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

4.4.1 变量的存储类别

- 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

4.4.1 变量的存储类别

```
//书P306 程序9.4
…
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

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

4.4.1 变量的存储类别

- 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)

4.4.1 变量的存储类别

- 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)
{
    …
}
```

4.4.1 变量的存储类别

- 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

4.4.1 变量的存储类别

```
#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

4.4.1 变量的存储类别

- （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'）

4.4.1 变量的存储类别

- 单定义规则（ODR）
  - 变量只能有一次定义
- C++提供两种变量声明

```
//P311 全局变量：
double up;              //definition, up is 0
extern int blem;        //blem defined elsewhere
extern char gr = ‘z’;
          //definition because initialized
```

  - 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。

4.4.1 变量的存储类别

- extern不分配存储空间

书P311，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

4.4.1 变量的存储类别

- extern不分配存储空间

书P311，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
}
```

4.4.1 变量的存储类别

- 局部变量可隐藏同名的全局变量

书P312，程序9.5

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

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

书P312，程序9.6

```
//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
}
```

4.4.1 变量的存储类别

- 静态全局变量隐藏常规全局变量

//书P314 程序

```
//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 file 2
    …
}
```

static errors的链接性为内部

4.4.1 变量的存储类别

- 链接性为外部和内部的变量

书P314，程序9.7

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

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

书P315，程序9.8

```
//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。

4.4.1 变量的存储类别

- 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（本函数）	无	动态数据区	不确定
形参	自动（本函数）	代码块（本函数）	无	动态数据区	不确定
寄存器	自动（本函数）	代码块（本函数）	无	CPU的寄存器	不确定
静态局部	静态（程序执行中）	代码块（本函数）	无	静态数据区	0（’\0’）
静态全局	静态（程序执行中）	文件（本源程序文件）	内部	静态数据区	0（’\0’）
外部全局	静态（程序执行中）	文件（全部源程序文件）	外部	静态数据区	0（’\0’）

4.4.2 函数和链接性

- 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

4.4.2 函数和链接性

- 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！

4.4.2 函数和链接性

- 内部函数和外部函数

```
//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)
{          //内部链接性
    …
}
```

```
//file2.cpp
…
不可调用/声明
任何max函数
```

```
//file3.cpp
…
static void max (int x, int y)
{          //内部链接性
    …
}
```

不同的源程序文件中的内部函数可以同名

4.4.2 函数和链接性

- 内部函数和外部函数

//file1.cpp	//file2.cpp	//file3.cpp
float f2();	static float f2();	extern float f2();
main()	int f3()	int f4()
{ f2();	{ f2();	{ f2();
}	}	}
float f2()	float f2()	
{...}	{...}	
//正确		

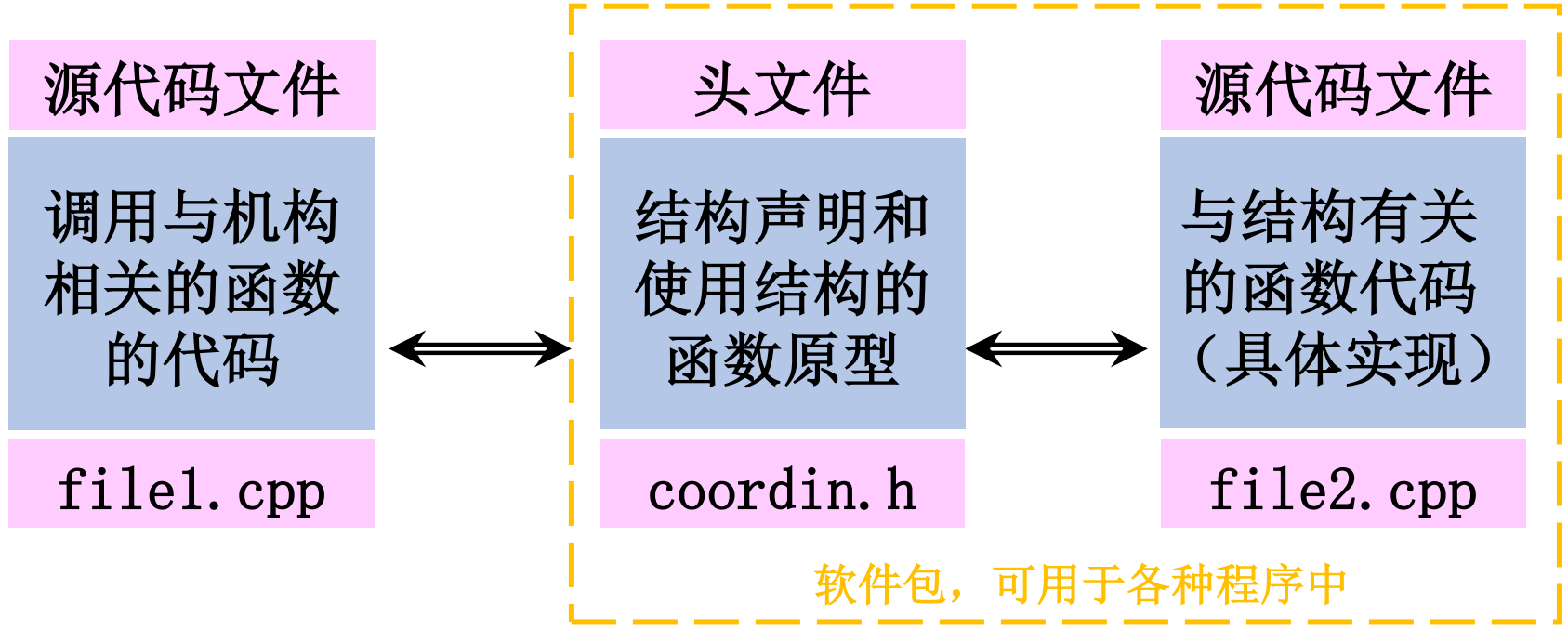
4.4.2 函数和链接性

- 内部函数和外部函数

//file1.cpp	//file2.cpp	//file3.cpp
float f2();	float f2();	extern float f2();
main()	int f3()	int f4()
{ f2();	{ f2();	{ f2();
}	}	}
float f2()	float f2()	
{...}	{...}	
//错误：非静态，找到两个定义		

4.4.3 多源程序

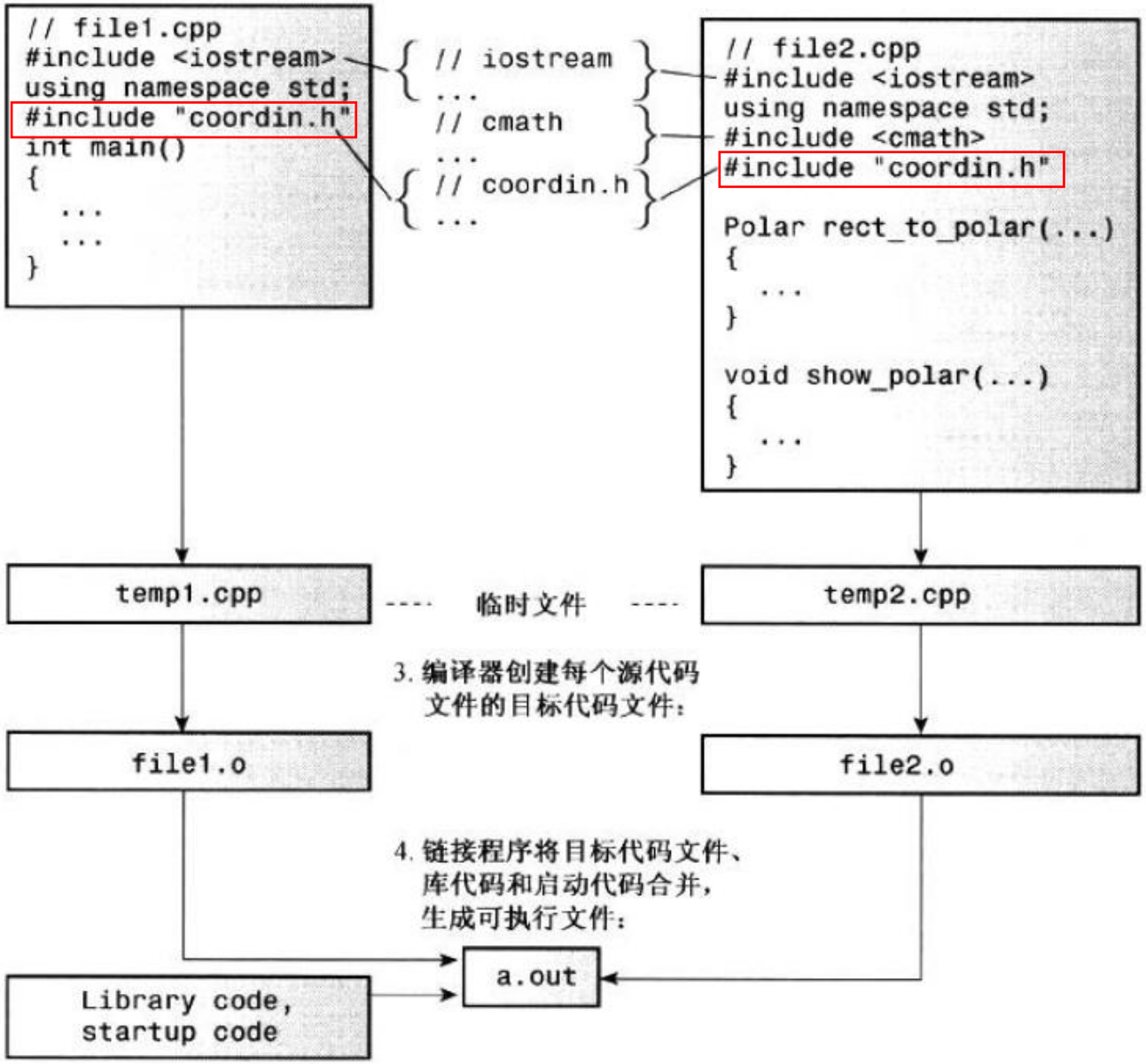
- 头文件的引入
  - P232 程序7.12的代码组织策略（结构为后续知识点，重点了解组织策略）



  - 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

4.4.3 多源程序

- 头文件的管理
  - 在一个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



4.4.3 多源程序

- 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
```

4.4.3 多源程序

- 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

4.4.3 多源程序

- 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

4.4.3 多源程序

- 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

4.4.3 多源程序

- 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

4.4.3 多源程序

- 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+VS2019为例：C:\Program Files(x86)\Microsoft Visual Studio\2019\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

4.4.3 多源程序

- 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

4.4.4 本节小结

- 应用程序执行时变量的存储分布
- 从持续性/作用域/链接性/存储区/初始化等角度全方位认识变量
- 自动变量与静态局部变量的特点
- 全局变量与局部变量的特点
- 使用extern扩展全局变量作用范围的正确方法（链接性）
- 内部函数与外部函数
- 头文件的作用、头文件的管理，多源程序的代码组织策略