南京大学 计算机科学与技术系
Department of Computer Science & Technology, NJU

Chapter 2 (cont.)

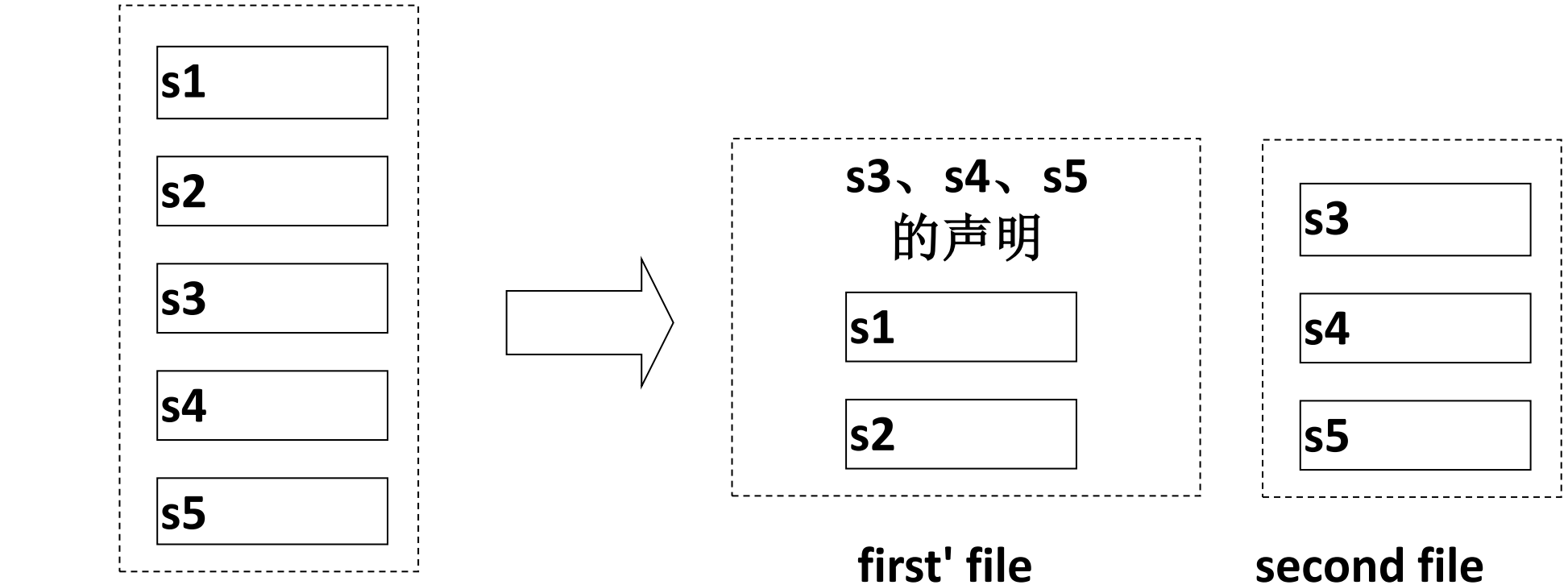
程序的模块设计方法（续）

刘奇志

- 简介
- 单模块程序设计
  - C语言函数（子程序）基础
    - 函数的定义、调用和声明
  - C语言函数的嵌套调用
    - 嵌套调用及其过程
    - 递归
- 多模块程序的设计（Multiple modules programming）
  - 文件包含命令
  - 头文件及其作用
  - 库函数
  - 标识符的属性
- 程序模块设计的优化

多个子程序分布在多个模块中

- 一个文件通常被看成一个模块。



s1调用s3，s2调用s4、s5

文件包含预处理（preprocess）命令

- #include
  - 使一个源文件可以将一个头文件的内容在预处理阶段包含（替换）进来。

- 为了便于多模块程序的开发，引入：
  - **文件包含预处理命令**
  - **头文件**

头文件（head file）

- **是一种特殊的源文件**
- 其中一般只有一些说明性的内容，比如，符号常量的定义、函数的声明等内容（**函数头前面的内容**），没有函数的定义等实质性的程序
- **用 .h 作扩展名**

//temp.h

```
#define LOWER 0
#define UPPER 200
#define STEP 10
```

//初步预处理后的main.cpp

```
#define LOWER 0
#define UPPER 200
#define STEP 10
#include <stdio.h>
int main()
{
    double f, c;
    f = LOWER;
    while(f <= UPPER)
    {    c = 5.0/9.0 * (f - 32.0);
         printf …;
         f +=  STEP;
    }
    return 0;
}
```

//预处理前的 main.cpp

```
#include "temp.h"
#include <stdio.h>
int main()
{
    double f, c;
    f = LOWER;
    while(f <= UPPER)
    {    c = 5.0/9.0 * (f - 32.0);
         printf …;
         f += STEP;
    }
    return 0;
}
```

//初步预处理后的main.cpp

```
#define LOWER 0
#define UPPER 200
#define STEP 10
#include <stdio.h>
int main()
{
    double f, c;
    f = LOWER;
    while(f <= UPPER)
    {   c = 5.0/9.0 * (f - 32.0);
        printf …;
        f +=  STEP;
    }
    return 0;
}
```

//temp.h

```
#define LOWER 0
#define UPPER 200
#define STEP 10
#include <stdio.h>
```

//预处理前的 main.cpp

```
#include "temp.h"
int main()
{
    double f, c;
    f = LOWER;
    while(f <= UPPER)
    {   c = 5.0/9.0 * (f - 32.0);
        printf …;
        f += STEP;
    }
    return 0;
}
```

例2.7

- 甲乙两人共同开发一个程序，实现求最大数的阶乘功能，其中，甲编写的模块（first.cpp）包括main函数和求最大值的函数，乙负责的模块（second.cpp）包括求阶乘函数。

```
//first.cpp甲
#include <stdio.h>
int myMax(int, int, int);
extern int myFactorial(int);   //声明乙编写的函数
int main()
{    int n1, n2, n3, max, f;
     printf("Please input three integers: \n");
     scanf("%d", &n1, &n2, &n3);
     max = myMax(n1, n2, n3);
     f = myFactorial(max);
     printf …
     return 0;
}

int myMax(int n1, int n2, int n3)
{     …   }
```

```
//second.cpp乙
int myFactorial(int n)
{   int f = 1;
    for(int i=2; i <= n; ++i)
          f *= i;
    return f;
}
```

```
//first.cpp甲
#include <stdio.h>

int myMax(int, int, int);
#include <second.h>
int main()
{ …
  max = myMax(n1, n2, n3);
  f = myFactorial(max);
  …
}

int myMax(int n1, int n2, int n3)
{ ... }
```

```
//second.h
extern int myFactorial(int);
/* 乙所编写的函数的声明更适合由乙来
   编写，而如果写在乙编写的源文件中，
   又不便其他人使用，写在头文件中即可
   */
```

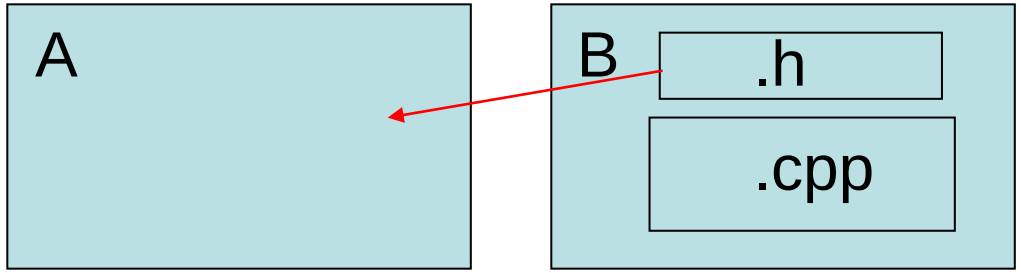
```
//second.cpp乙
int myFactorial(int n)
{
      ...
}
```

头文件的作用

- **保护源文件的代码**
  - 模块的实现者不必提供源文件中的代码，可以只提供编译好的目标文件给使用者，从而可以保护源文件中的代码不被篡改或抄袭。

- **使模块的划分更合理**

- 头文件使一个模块分为两部分：
  - 接口信息（interface，.h文件）
    - 给出在本模块中定义的、提供给其他模块使用的一些函数等程序实体的声明
  - 具体实现（implementation，.cpp文件）
    - 给出函数等程序实体的定义



  - 模块的实现者可以把接口信息提供给使用者，使用者用文件包含命令（#include）将头文件包含进自己的源文件中，无需了解模块的具体实现方法，从而减轻了使用者的工作量。

- 头文件的运用使程序的开发更为灵活与可靠。我们就是用这种方式使用库函数的。

- 头文件中的说明信息除了函数的声明、符号常量的宏定义之外，还可以是变量的声明等内容。

//xlib.cpp

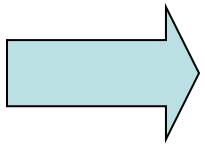
```
int x, y;
int F()
{     x = 1000;
      y = x*x;
      return y;
}
```

//my.cpp

```
extern int x, y;
int G()
{     int z;
      F();
      z = x + y;
      return z;
}
```

//xlib.h

```
extern int x, y;
```



//my.cpp

```
#include "xlib.h"
int G()
{     int z;
      F();
      z = x + y;
      return z;
}
```

- #include <<头文件名>>
  - 头文件在系统指定的目录（默认是系统安装时自动生成的\include目录）下
- #include "<头文件名>"
  - 头文件在当前所建立的项目源文件目录或系统指定的目录下

- 一个#include命令只能指定包含一个头文件
- 可以嵌套包含，即头文件中可以用#include命令包含别的头文件

C标准库函数

- 一个语言本身所提供的功能是有限的
  - 语言的设计者不可能预见程序设计所需要的所有功能
  - 语言本身提供太多的功能也会给语言的学习和实现（编译程序的设计）增加负担
  - 太多的功能给语言的扩充带来麻烦
- C语言的每个实现往往会提供一个标准库，其中定义了一些语言本身没有提供的功能：
  - 常用的数学函数
  - 字符串处理函数
  - 输入/输出
  - …
- 在C标准库中，根据功能对定义的程序实体进行了分类，把每一类程序实体的声明分别放在一个头文件中（在C++中，把从C语言保留下来的库函数，重新定义在名空间std中，并重新命名为：*.h -> c*）

- 在标准库的头文件cmath（math.h）中声明的一些数学函数：

```
double fabs( double x );                    //double型的绝对值
double sin( double x );                     //正弦函数
double cos( double x );                     //余弦函数
double tan( double x );                     //正切函数
double asin( double x );                    //反正弦函数
double acos( double x );                    //反余弦函数
double atan( double x );                    //反正切函数
double ceil( double x );                    //不小于x的最小整数
double floor( double x );                   //不大于x的最大整数
double log( double x );                     //自然对数
double log10( double x );                   //以10为底的对数
double sqrt( double x );                    //平方根
double pow( double x, double y );           //x的y次幂
```

- 在标准库的头文件cstdlib（stdlib.h）中声明的一些函数：
  - `int abs( int n );` //int型的绝对值
  - `long labs( long n );` //long int型的绝对值
  - `int rand( );` //生成一个伪随机数
  - `void srand( unsigned int seed );` //为rand设置//"种子"的值
  - `void exit( int status );` //终止整个C程序的执行，
    //status用于指出终止的原因，一般取0表示程序正常终止
  - `void abort( );` //终止整个C程序的执行，
    //与exit的主要区别是不做“关闭文件”等“善后”处理，
    //可能会使程序写到文件中的一些数据丢失！

- 在标准库的头文件cctype（ctype.h）中声明的一些函数：
  - `int isdigit( int c );` //判断c是否为数字，返回非零：是，返回0：不是
  - `int isalpha( int c );` //判断c是否为字母，返回非零：是，返回0：不是
  - `int isalnum( int c );` //判断c是否为字母或数字，返回非零：是
  - `int isupper( int c );` //判断c是否为大写字母，返回非零：是
  - `int islower( int c );` //判断c是否为小写字母，返回非零：是
  - `int tolower( int c );` //大写字母，则返回相应的小写字母，否则返回c
  - `int toupper( int c );` //小写字母，则返回相应的大写字母，否则返回c

- 子程序及程序的多模块结构提高了程序开发的灵活性与可靠性，同时带来两个问题：
  - 如何确保一个程序中不同的子程序或不同的模块在需要的时候能够共同操作（共享）同一个操作数?
  - 如何区分这些子程序或模块中定义的同名标识符?
  - C语言根据程序的具体结构、标识符的性质、标识符的定义和声明的位置、以及相应的修饰关键字，赋予标识符一系列属性，然后根据操作数的属性可以判断是否可以共享该操作数，还可以正确区分同名标识符，甚至有利于在程序运行期间合理分配数据占用内存的时间，节约内存开销。

标识符的属性

- 作用域（scope）
  - 链接（linkage）
  - 名空间（namespace）
- 生存周期（storage duration）

作用域（scope）

- 即标识符的有效范围
  - 标识符：指的是变量与函数等程序实体名
  - 有效范围：指的是在程序代码中能够操作该标识符的段落
  - 标识符的作用域与它的声明和定义位置有关
  - 一般是：有声明，从声明开始，否则从定义开始 有效

- C语言标识符的作用域分为
  - 文件作用域：在一个源文件内有效（从声明或定义开始）
  - 块（局部）作用域：在一个语句块内有效(从声明或定义开始)
  - 函数作用域：在一个函数体内有效
  - 函数原型作用域：在一个函数原型内有效

文件作用域

```
// second.cpp
extern int s; //从声明开始有效
void mySum(int n) //从定义开始有效
{
   int sum = 0;
   for(int i=1; i <= n; ++i)
       sum += i;
   s = sum;
}
```

```
// first.cpp
#include <stdio.h>
int s = 0; //从定义开始有效
extern void mySum(int); //从声明开始有效
int main()
{     int n;
      printf("Please input an integer: \n");
      scanf("%d", &n);
      if(n <= 0) goto L1;
      mySum(n);
L1:   printf("s = %d \n",  s);
      return 0;
}
```

文件作用域

```
// second.cpp
static void mySum(int n) //从定义开始有效
{
      int sum = 0;
      for(int i=1; i <= n; ++i)
            sum += i;
      printf("%d", sum);
}
```

```
// first.cpp
#include <stdio.h>

static int s = 0; //从定义开始有效
int main()
{     int n;
      printf("Please input an integer: \n");
      scanf("%d", &n);
      if(n <= 0) goto L1;
      s = ... ;
L1:   printf("s = %d \n", s);
      return 0;
}
```

文件作用域

```
// first.cpp
#include <stdio.h>
int s = 0;
void mySum(int); //从声明开始有效
int main()
{
    int n;
    printf("Please input an integer: \n");
    scanf("%d", &n);
    if(n <= 0) goto L1;
    mySum(n);
L1: printf("s = %d \n", s);
    return 0;
}
```

```
void mySum(int n)
{
    int sum = 0;
    for(int i=1; i <= n; ++i)
        sum += i;
    s = sum;
} //写在 first.cpp 文件后部
```

块作用域

```
// first.cpp
#include <stdio.h>
int s = 0;
int main()
{
    void mySum(int);     //mySum
    int n;               //n
    printf("Please input an integer: \n");
    scanf("%d", &n);
    if(n <= 0) goto L1;
    mySum(n);
L1: printf("s = %d \n", s);
    return 0;
}
```

```
void mySum(int n)                    //n
{
    int sum = 0;                     //sum
    for(int i=1; i <= n; ++i)        //i
        sum += i;
    s = sum;
} //写在 first.cpp 文件后部
```

函数作用域

```
void mySum(int n)
{
    int sum = 0;
    for(int i=1; i <= n; ++i)
        sum += i;
    s = sum;
} //写在 first.cpp 文件后部
```

```
// first.cpp
#include <stdio.h>
int s = 0;
void mySum(int n);
int main()
{
    int n;
    printf("Please input an integer: \n");
    scanf("%d", &n);
    if(n <= 0) goto L1;
    mySum(n);
L1: printf("s = %d \n", s); //L1
    return 0;
}
```

函数原型作用域

```
// first.cpp
#include <stdio.h>
int s = 0;
void mySum(int n); //n
int main()
{
    int n;
    printf("Please input an integer: \n");
    scanf("%d", &n);
    if(n <= 0) goto L1;
    mySum(n);
L1: printf("s = %d \n", s);
    return 0;
}
```

```
void mySum(int n)
{
    int sum = 0;
    for(int i=1; i <= n; ++i)
        sum += i;
    s = sum;
} //写在 first.cpp 文件后部
```

不同作用域中，可以定义同名标识符

- **比如，**
  - 不同源文件中可以定义同名函数或同名全局变量（只要加static保证链接不发生错误即可）

  （以下做法风格不好）
  - 不同函数中甚至同一个函数的不同复合语句中可以自由定义同名局部变量
  - 不同函数中可以有相同的语句标号
  - 不同函数原型中的形参名（如果没省略）可以同名
  - 源文件中的全局变量可以与某个局部变量同名
  - 函数中复合语句外的局部变量可以与某个语句里的局部变量同名

- 内层作用域的标识符会“挤走”外层作用域的同名标识符。

```
int myFactorial(int n)
{
    int f = 1;
    for(int i=2; i <= n; ++i)
    {
        int f = 0;
        f *= i;
    }
    return f;
}
```

//for语句里的f每次都初始化为0

//这里的f仍为1

- 在内层作用域中若要使用与其同名的外层作用域中的标识符，则需要用全局域选择符（::）进行修饰。

```
int f = 1;
int myFactorial(int n)
{
      int f = 0;
      for(int i=2; i <= n; ++i)
      {
            ::f *= i;
      }
      return f;
}
```

for 语句中循环变量的作用域

- for语句头部定义的变量，其作用域仅限于for语句

```
for(int i=1; i <= 10; ++i)
{
    …
}
```

- 部分编译器未按任何标准设计

链接（linkage）

- **链接器根据链接属性判断是不是同一个实体**
  - 同名标识符，链接属性种类相同是同一个实体
- **无链接**
  - 只有定义没有声明
    - 不同名的标识符之间
    - 非函数且非变量的同名标识符之间
    - 同名函数参数之间
    - 同名局部变量之间
    - 先定义后使用的函数或全局变量
- **有链接**
  - 不是先定义后使用的
    - 同名全局变量之间
    - 同名函数之间

编译器是根据作用域属性区分不同的同名实体

```c
//first.c-by 甲
#include <stdio.h>
int m = 2;                              //外部变量的定义，被外部链接
extern int MyFactorial(int);            //外部函数的声明
int MyMax(int, int, int);               //内部函数的声明
int main()
{       int n1, n2, n3;
        scanf("%d%d%d", &n1, &n2, &n3);         //n1, n2, n3, max, f无链接
        int max = MyMax(n1, n2, n3);            //MyMax函数的调用，内部链接
        int f = MyFactorial(max);               //MyFactorial函数的调用，外部链接
        printf("The factorial of max. is: %d \n", f);
        return 0;
}
static int MyMax(int n1, int n2, int n3)   //内部函数的定义，被内部链接
{       int max;
        if(n1 >= n2)            // n1, n2无链接
                max = n1;       // max无链接
        else
                max = n2;
        if(max < n3)            // n3无链接
                max = n3;
        return max;
```

```
//second.c-by 乙
int MyFactorial(int n)   //外部函数的定义，被外部链接
{
      extern int m;      //外部变量的声明
      int f = 1;
      for(int i=2; i <= n; ++i)      //i, f, n无链接
            f *= i;
      return m*f;                    //m的操作，外部链接
}
int MyMax(int n1, int n2, int n3)    //外部函数的定义，没有被链接过
{
      int max;
      if(n1 >= n2)
            max = n1;
      else
            max = n2;
      if(max < n3)
            max = n3;
      return max;
}
```

- **内部链接（优先）：**
  - 函数和全局变量，定义时加static，具有内部链接属性，可称为内部函数和内部变量
  - 声明内部函数时，也要加关键字static修饰（注意，不能在块作用域内声明内部函数）
  - 内部变量先定义后使用，一般无需声明。如果定义之后有声明，则声明必须加extern，以便与变量的定义相区别，而且这样的声明并不改变其内部链接这一属性
- **外部链接：**
  - 定义 没加static（加或不加extern均可）的全局变量，具有外部链接属性，可称为外部变量。声明时，必须加extern
  - 定义 没加static （加或不加extern均可）的函数，一般具有外部链接属性，可称为外部函数。除非在本源文件前面声明时加了static把它转化为内部函数。声明外部函数时，一般加extern（extern可省略）
- **在有些教科书上，把内部链接属性看作文件作用域属性，把外部链接属性看作全局作用域属性（这样看待不够准确）**

名空间（namespace）

- C：隐含的名空间

- C++：
  - 隐含的名空间
  - 程序中指定名空间和使用名空间

隐含的名空间

- 是一种抽象的标识符的容器。
- 程序中同一个作用域内逻辑上相关的标识符隐藏在同一个名空间里，否则，隐藏于不同的名空间。

- 隐含的名空间有四种：
  - 语句标号的名空间；
  - 标签的名空间（派生类型名/枚举符）；
  - 某个派生类型（结构/联合）的所有成员的名空间；
  - 其他标识符的名空间，这些标识符包括变量名、函数名、形参名。

一个程序中，相同作用域内的同一个名空间里不允许定义同名标识符

- **否则编译时会出现“重复定义”错误。**
- **比如，**
  - 一个源文件中不可以定义同名函数或同名全局变量
  - 函数里复合语句外不可以定义同名或与形参同名的局部变量
  - 一个复合语句中不可以定义同名局部变量
  - 一个函数中不能有相同的语句标号
  - 函数原型中的形参名如果没省略的话，相互不可以重名

- 多模块程序可能会面临的一个问题：
  - **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：外部函数或外部变量），而这两个全局程序实体的名字相同。**

C++程序中指定名空间和使用名空间

- 指定名空间：
  - namespace X{…}
  - 在一个名空间中定义的全局标识符，其作用域为该名空间

- 使用名空间：
  - using namespace x
  - x::
  - 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()
  { //......
  }
}
```

```
//接口1(mh1.h)
namespace A
{ extern int x;
  void f();
}
```

```
//模块2
namespace B
{ int x=0;
  void f()
  { //......
  }
}
```

```
//接口2(mh2.h)
namespace B
{ extern int x;
  void f();
}
```

要在头文件和源文件中同时定义名空间

1、

```
#include "mh1.h"
#include "mh2.h"
```

//模块3

```
A::x = 3;          //A中的x
A::f();            //A中的f
B::x = 5;          //B中的x
B::f();            //B中的f
```

2、

```
using namespace A;
x = 3;             //A中的x
f();               //A中的f
B::x = 5;          //B中的x
B::f();            //B中的f
```

3、

```
using A::f;
A::x = 3;          //A中的x
f();               //A中的f
B::x = 5;          //B中的x
B::f();            //B中的f
```

作用域：代码中的有效范围

- 文件
- 块
- 函数
- 函数原型
- 名空间作用域

两个标识符的作用域相同意味着：
作用域种类相同，而且有效范围在同一处结束

看到同名标识符，代表的是不同的实体，意味着：
要么在不同的作用域，要么在不同的名空间

数据的存储期（storage duration）

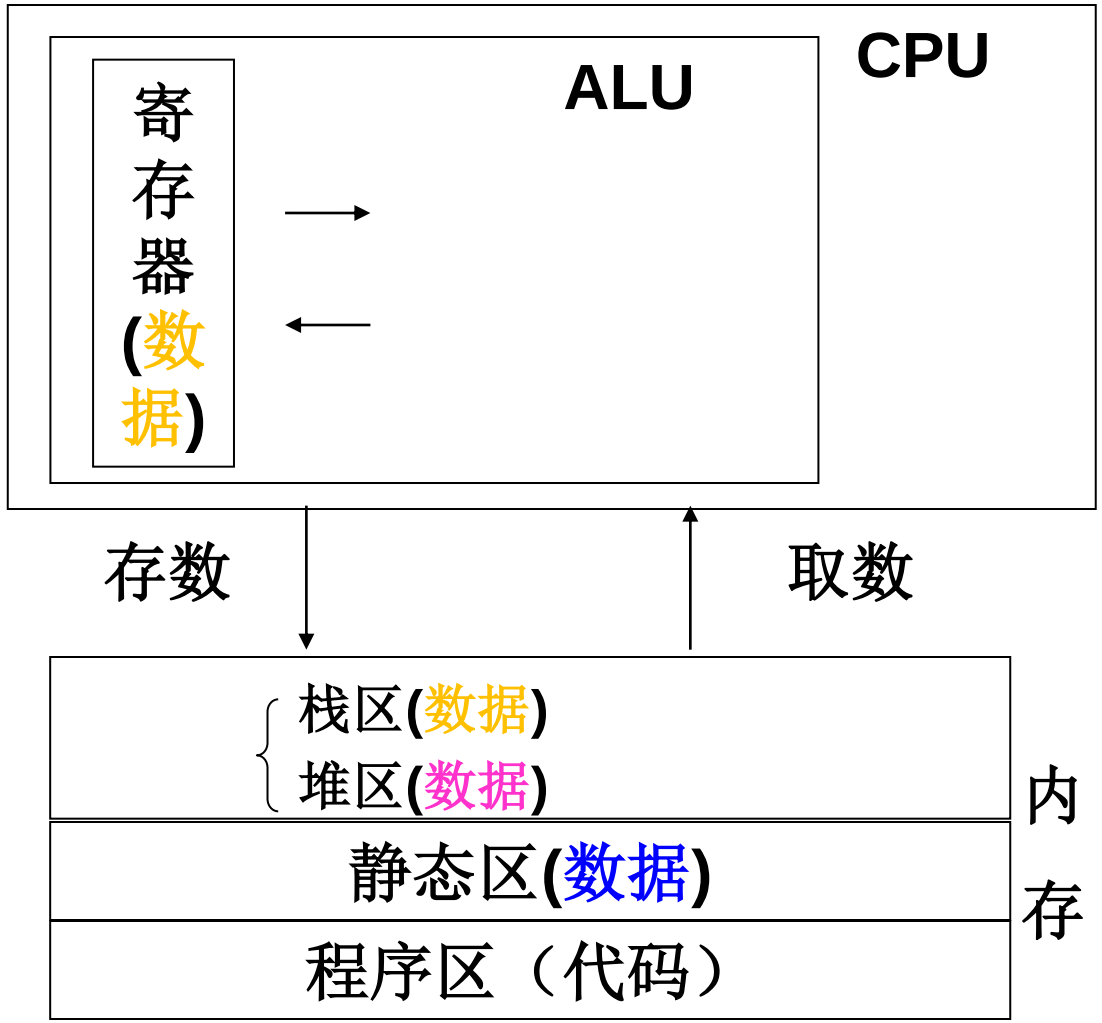
- **即数据的生存寿命（lifetime）**

从操作系统角度来看

  - 在程序执行期间，代码中的部分标识符（主要是变量）作为操作对象，一般需要有对应的内存空间来存储待处理的数据及处理结果。**为了合理分配时间，节约开销，**可以约定不同的数据存储期。
  - 例如，某子程序单独操作的数据只在该子程序执行期间占用空间，一旦该子程序执行结束，其空间即被收回或释放（以便再存储其他数据），**多个子程序共同操作的数据则在内存中驻留更长的时间。**

- C语言的数据存储期属性有：
  - **静态存储期**
  - **自动存储期**
  - **动态存储期**

- 在程序执行期间，不同存储位置中的数据存储期不同



内部函数与外部函数都存在程序区

静态存储期

- 对于**全局变量**（不论定义时是否加关键字static，这里static表示内部链接属性），先由编译器对其定义行进行特殊处理（规划所需内存，分析、处理其初始化值），程序执行时，由执行环境在静态数据区为之分配空间，并写入初始化值，**若未初始化则写入初值0**，此后程序可获取或修改其值，直到整个程序执行结束才收回其空间。这是一种静态内存分配方式。

- **对于定义时加static的局部变量**（通常称之为静态变量，这里static表示存储期属性），也是采用静态内存分配方式。程序员在分析含有静态变量且被多次调用的函数时，需注意，静态变量的定义行经过编译器的特殊处理后，**相关的内存空间分配和初值写入操作仅在程序刚开始执行时由执行环境执行一次**，此后静态变量可以被访问，即其值可以被获取或修改，但不会被重新分配空间或初始化，直到整个程序执行结束。

自动存储期

- **对于定义时没有加static的局部变量**（通常称之为自动变量，定义时可以加关键字auto，一般省略auto），其定义行有对应的目标代码，通过代码的执行在栈区获得空间，若已初始化则获得初始化值，若未初始化则其初值是内存里原有的值，此后程序可以访问其内存空间，获取或修改其值，一旦复合语句执行结束即收回其内存空间。如果一个复合语句在程序中被反复执行多次，则每次被执行时，都会重新分配和收回其中定义的自动变量的存储空间。这是一种动态内存分配方式。

- 对于被频繁访问的占用空间不大的自动变量（例如循环变量），**定义时可加关键字register**，以便在执行环境支持时存入寄存器，提高访问效率，如果执行环境不支持，一般仍当作自动变量处理。

- 对于**函数的形参**，也是采用动态内存分配方式。**定义时不可以加auto，可以加register**，效果与自动变量加register效果相同。如果一个函数在程序中被多次调用，则每次被调用时都会重新分配其形参的存储空间，每次函数执行结束时收回其形参的存储空间。

动态存储期

- **对于没有定义的动态变量，**由程序员编写的相关代码（**调用new操作或malloc库函数**）申请内存空间，通过代码的执行在堆区获得空间，由于没有初始化，其初值为内存里原有的值，此后程序可以访问其内存空间，进行赋值操作，以及获取或修改其值，最后通过程序员编写的相关代码（**调用delete操作或free库函数**）释放其内存空间。如果程序员忘记编写释放其内存空间的代码，则要等整个程序执行结束时才收回其内存空间。这是一种更为灵活的动态内存分配方式。

存储期

寄存器

不可进行取地址操作

加register的
局部变量或形参

堆区

栈区

静态区

内存空间在程序中显式分配与收回

存储空间在程序执行到定义它们的语句时才分配，当定义它们的复合语句执行结束时，所分配的空间将被收回

局部变量，
无修饰或加auto

函数的形式参数(定义时不可加auto，不可初始化)，
函数调用时有关信息(函数返回地址等)，
无修饰

默认值为0(最好显式初始化)。内存空间从程序开始执行时就进行分配，直到程序结束才收回所分配的空间

全局变量，
有/无修饰

常量，
无修饰

加static的
局部变量

例 输出任意三个不同的整数。

```
#include <stdio.h>
int myRand(void);
int main()
{     auto int m;      //自动变量
      m = myRand();
      printf("The m is: %d \n", m);
      m = myRand();
      printf("The m is: %d \n", m);
      m = myRand();
      printf("The m is: %d \n", m);
      return 0;
}
int myRand(void)
{     static int s = 1;          //静态变量
      s = (7 * s + 19) % 3;
      return s;
}
```

static

- **关键字static有两种不同的含义。**
  - 指定**局部变量**采用**静态存储分配**（降低了可读性，但具有数据保留作用，如果所在的函数再次被调用，该静态变量不再被初始化，而是保留上次的执行结果）
  - 把**外部链接**改变为**内部链接**（只希望在一个模块内共享的函数和全局变量，定义时可以加static，降低了通用性，但具有数据保护作用）----把一个文件的代码放到一个无名的名空间中，也可以起到相同的效果

要在头文件和源文件中同时定义名空间

```
//模块1
namespace A
{ int x=1;
  void f()
  { //......
  }
}
…//可以使用 x 和 f
```

```
//模块2
namespace B
{ int x=0;
  void f()
  { //......
  }
}
```

```
//模块3
#include "mh1.h"
#include "mh2.h"
x = 3;      //?无法使用A中的 x
f();        //?无法使用A中的 f
B::x = 5;   //B中的 x
B::f();     //B中的 f
```

作业4

- **Ex1. 设计程序，用递归函数求n个圆盘的河内塔问题的最少移动次数。**

$$H(n) = \begin{cases} 1 & (n=1) \\ 2 \cdot H(n-1)+1 & (n>1) \end{cases}$$

```
int H(int n)
{
    if(n == 1)
        return 1;
    else
        return 2*H(n-1)+1;
}
```

- 2^n-1

```
int heneita(int n)
{
    int x;
    if(n == 1)
        x = 2;
    else
        x = 2 * heneita(n - 1);
    return x;
} //求解2^n
```

```
int main(){
    int n;
    cin >> n;
    int b = heneita(n) - 1;
    cout << b << endl;
    return 0;
}
```

- **Ex2. 编程，用递归函数求埃尔米特（Hermite）多项式中第n+1项Hn(x)的值。Hn(x)定义为：**

H0(x) = 1

H1(x) = 2x

Hn(x) = 2xHn-1(x) – 2(n-1)Hn-2(x)　　(n > 1)

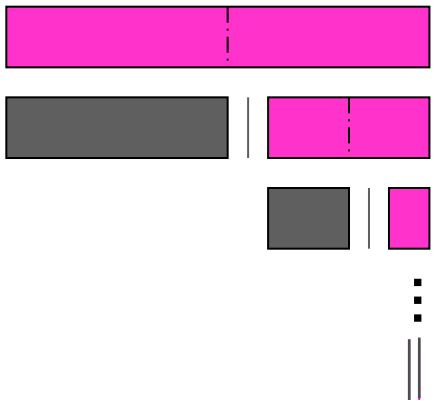
（测试数据：输入0 3，输出1；输入1 3，输出6；输入2 3，输出34；输入5 3，输出3816）

$$H_0(x) = 1$$
$$H_1(x) = 2x$$
$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x)$$

```
double Hermite_Recur(int n, double x)
{     if(n == 0)
            return 1;
      else if(n == 1)
            return 2*x;
      else
            return 2*x*Hermite_Recur(n-1, x)
                  - 2*(n-1)*Hermite_Recur(n-2, x);
}
```

第8周自主训练任务

- 1. 一只猴子第一天摘下若干个桃子，当即吃了一半，不过瘾，又多吃了一个；第二天早上将剩下的桃子吃掉一半，又多吃了一个；以后每天早上都吃了前一天剩下的一半零一个；到第十天早上，还剩两个桃子，被猴子吃掉了。编写递归函数求倒数第n（n>=1）天猴子吃桃之前所剩桃子个数，并在main函数中调用该函数，输出第一天猴子摘了多少个桃子



```
int PeachR(int n)
{
     if(n == 1)
          return 1;
     else
          return 2 * (PeachR(n-1)+1);
}

int main()
{
    int day = 10;
    //cin >> day;
    cout << "第1天猴子摘了" << PeachR(day) << "个桃子\n";
    return 0;
}
```

- **2. 在屏幕上显示如下杨辉三角（提示：可将空位置看作0，用递归函数求解其中的数值）：**

```
        1
     1     1
  1     2     1
1     3     3     1
```

0	0	0	**1**	0	0	0
0	0	**1**	0	**1**	0	0
0	**1**	0	**2**	0	**1**	0
1	0	**3**	0	**3**	0	**1**

```
#include <stdio.h>
int n;
int PascTri(int i, int j)
{
        if((i == 1) && (j == n))
                return 1;
        else if((i == 1) && (j != n))          //含j<1或j>n
                return 0;
        else
                return PascTri(i-1, j-1) + PascTri(i-1, j+1);
}
```

0	0	0	0	**1**	0	0	0
	0	0	**1**	0	**1**	0	0
	0	**1**	0	**2**	0	**1**	0
	1	0	**3**	0	**3**	0	**1**

0

n为4
4行7列

```
int main()
{
    int i, j, k;
    printf("Enter n (<=8):");
    scanf("%d", &n);
    for(i=1; i <= n; ++i)
    {   for(j=1; j <= n*2-1; ++j)
            if((k=PascTri(i, j)) == 0)
                printf("%5c", ' ');
            else
                printf("%5d", k);
        printf("\n");
    }
    return 0;
}
```

- 3. 编写递归函数，实现阿克曼（Ackermann）函数的计算（在main函数中输入m、n [0, 4)，输出Ackermann函数值）。Ackermann函数的值随参数快速递增，用于可计算理论领域，其计算方法为：

$$A(m,n) = \begin{cases} (n+1) & m=0 \\ A(m-1,\ 1) & m>0,\ n=0 \\ A(m-1,\ A(m,\ n-1)) & m>0,\ n>0) \end{cases}$$

```
int Ack(int m, int n)
{     if(m == 0)                    //不管n是否为0
            return n+1;
      else if(m>0 && n==0)
            return Ack(m-1, 1);
      else
            return Ack(m-1, Ack(m, n-1));
}
```

- **5. 分析下面程序的执行过程与结果，并上机验证。**

```
#include <stdio.h>
void EchoPrint(int);
int main()
{
    int n;
    printf("input n:");
    scanf("%d", &n);
    EchoPrint(n);
    printf("\n");
    return 0;
}
```

```
void EchoPrint(int n)
{
    if(n >= 10)
        EchoPrint(n/10);
    printf("%d", n%10);
}
```

提示

```
EchoPrint(1234);
```

```
void EchoPrint(int n)
{
    if(n >= 10)
        EchoPrint(n/10);
    cout << n%10;
}
```

n: 1234

```
EchoPrint(123);
cout << n%10;
```

n: 123

```
EchoPrint(12);
cout << n%10;
cout << n%10;
```

n: 12

```
EchoPrint(1);
cout << n%10;
cout << n%10;
cout << n%10;
```

n: 1

```
cout << n%10;
cout << n%10;
cout << n%10;
cout << n%10;
```

- 6. 编写一个函数intRevs(n)，它返回一个整数n的逆序整数，例如：intRevs(89532) = 23598，intRevs(580) = 85

```
0*10 + 2
2*10 + 3
23*10 + 5
235*10 + 9
2359*10 + 8
```

```
int intRevs(int n)
{     int n_revs = 0;
      while(n > 0)
      {     n_revs = n_revs*10 + n%10;
            n /= 10;
      }
      return n_revs;
}
```

如果是负数

```
int intRevs(int n)
{     int flag = 1, n_revs = 0;
      if(n < 0)
      {     n = -n;
            flag = -1;
      }
      while(n > 0)
      {     n_revs = n_revs*10 + n%10;
            n /= 10;
      }
      return flag*n_revs;
}
```

intRevsR(89532): $2*10^4$+ intRevsR(8953)

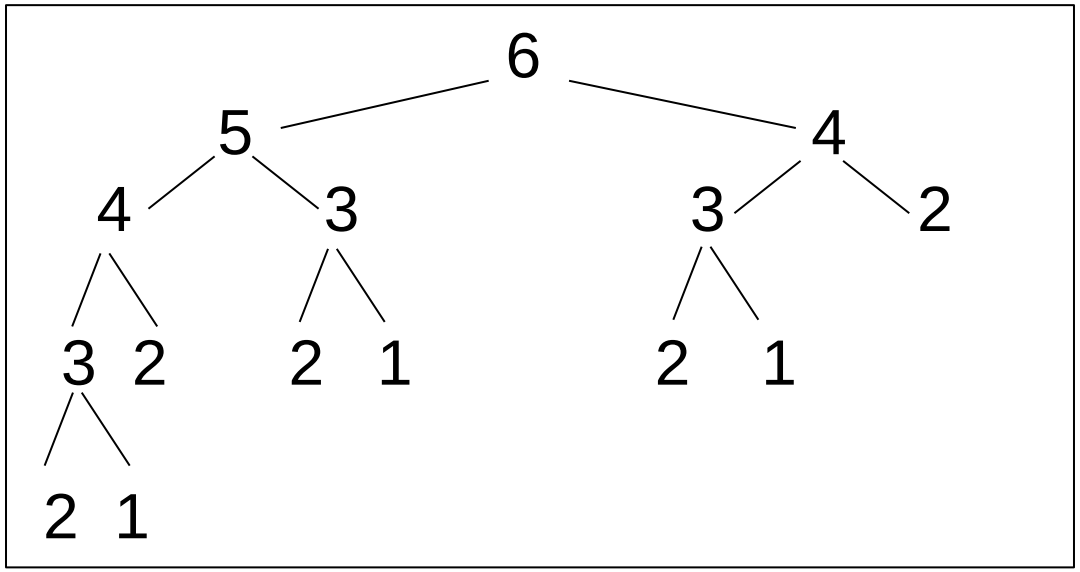
```
int intRevsR(int n)
{     int m = myPow(10, countDigit(n) - 1);
      if(n/10 == 0)   return n;
      else        return n%10*m + intRevsR(n/10);
}
```

```
int countDigit(int n)
{     int count = 0;
      while(n != 0)
      {     n = n/10;
            ++count;
      }
      return count;
}
```

```
int myPow(int x, int n)
{     int z = 1;
      while(n >= 1)
      {     z *= x;
            --n;
      }
      return z;
}
```

- **2．阅读下面的程序，思考全局变量的执行结果，体会递归函数MyFibR存在的重复计算问题。**

```
#include <stdio.h>
int MyFibR(int);
int count = 0;
int main()
{   printf("第6个月有 %d 对兔子.\n", MyFibR(6));
    printf("递归函数一共被调用执行了%d次.\n", count);
    return 0;
}
int MyFibR(int n)
{   ++count;
    if(n==1 || n==2)
          return 1;
    else
          return MyFibR(n-2) + MyFibR(n-1);
}
```



- 简介
- 单模块程序设计
- 多模块程序的设计（Multiple modules programming）
  - 文件包含命令
  - 头文件及其作用
  - 库函数
  - 标识符的属性
- 程序模块设计的优化
  - 带参数的宏定义
  - 内联函数
  - 条件编译
  - 函数名重载（C++）
  - 带默认值的形式参数（C++）

带默认值的形式参数

跟模块设计有关的优化

- 子程序（函数）机制可以使程序获得良好的结构，然而调用函数需要一些额外开销，因为系统要保护调用者的运行环境、进行参数传递、执行调用和返回指令等等，所以，函数一般会降低程序的执行效率，特别是频繁调用一些小函数往往会得不偿失。

带参数的宏定义

- #define <宏名>(<参数表>) <文本>
  - 程序中的<宏名>(<实参表>)在编译前会被相应宏定义中的<文本>替换，<文本>中的参数将被程序中<实参表>里相应的实参替换，整个替换过程又叫宏调用。
  - 宏调用主要用于解决对小函数的调用效率不高的问题。

例 带参数的宏定义实现乘法函数的功能。

```
#define Multiply(x, y) x*y
int main()
{
      double x1, y1;
      scanf("%lf%lf", &x1, &y1);
      printf("%f", Multiply(x1, y1));   // printf("%f", x1*y1);
}
```

注意事项

- 宏名与参数表之间不能有空格

  `#define <宏名>  (<参数表>) <文本>` ×

  `#define Multiply (x, y) x*y` ×

例 带参数的宏定义实现乘法函数的功能。

```
#define Multiply(x, y) x*y
int main()
{
    double x1, y1;
    scanf("%lf%lf", &x1, &y1);
    printf("%f", Multiply(x1, y1));  // printf("%f", x1*y1);
}
```

```
#define Multiply(x, y) ((x)*(y))
```

```
printf("%f", 1/Multiply(x1+y1, x2-y2));
```

```
printf("%f", 1/x1+y1*x2-y2);
```

```
printf("%f", 1/((x1+y1)*(x2-y2)));
```

- **带参数的宏定义的缺陷：**
  - 需要加上很多的括号，否则会出错
  - 不进行参数类型检查和转换
  - 不利于一些软件工具（比如调试程序）对程序的处理
    - 预处理后程序中的宏不复存在，给源程序与目标程序之间的交叉定位带来困难。

内联函数

- inline
  - 在函数定义时，函数头的返回值类型前加关键字inline，以示建议编译器把该函数的函数体展开到调用点

```
inline double Multiply(double x, double y)
{
   return x*y;
}
```

具有宏定义和函数二者的优点

- 内联函数形式上是函数
  - 遵循参数类型检查与转换等规定
- 效果上具有宏定义的高效率
  - 避免了函数调用的开销

内联函数的局限性

- 不是所有的编译器都支持内联函数机制（C++98/C99标准开始支持）；
- 递归函数一般不能定义为内联函数；
- 对于内联函数，编译器不生成独立的函数代码，而是用内联函数的函数体替换对内联函数的调用，所以，内联函数必须先定义后调用（调用前仅有函数声明也不行），具有文件作用域属性。
  - 实际应用中可以把内联函数的定义放在头文件中

- C语言还有另外两种宏定义：
  - #define <宏名>
    - 表示<宏名>已被定义（不作任何文本替换）
  - #undef <宏名>
    - 表示取消<宏名>的定义（其后如果有该宏名的文本替换也会被取消）

- 它们通常用来辅助实现条件编译

条件编译

- 预处理命令：根据不同的情况选择部分代码参加编译

```
#ifdef <宏名>
   <程序段1>
[#else
   <程序段2>]
#endif
```

/ #if defined(<宏名>)
/ #ifndef <宏名> / #if !defined(<宏名>)

- <宏名>可以在程序中用#define定义，也可以在编译器的选项中给出。

- 如果在程序中定义宏

```
#define ABC
#ifdef ABC
  <代码1> //如果宏名ABC有定义，编译之
#else
  <代码2> //如果宏名ABC没有定义，编译之
#endif

  <代码3> //必须编译的代码
```

- 缺点：要修改程序！

- **如果在开发软件中的程序编译命令或集成环境中加一个宏定义选项：**
  - 命令行方式
    - cl <源文件1> <源文件2> … -D ABC …
    - 其中的宏定义选项“-D ABC”使得宏名ABC有定义。
  - **集成环境方式（以Microsoft Visual Studio 2008为例）**
    -> Project -> XXX Properties -> Configuration Properties -> C/C++ -> Preprocessor -> Preprocessor definitions中添加要定义的宏名ABC。

† **优点：不用修改程序**

条件编译命令的另一种格式

```
#if <常量表达式1>
      <程序段1>
#elif <常量表达式2>
      <程序段2>
...
#elif <常量表达式n>
      <程序段n>
[#else
      <程序段n+1>]
#endif
```

/ #ifdef <宏名> / #if defined(<宏名>) / #ifndef <宏名> / #if !defined(<宏名>)

- <常量表达式>中只能包含字面常量或用#define定义的宏名

- 条件编译的作用：
  - 避免重复包含问题
  - 基于多环境的程序编制
  - 程序调试
  - …

- 条件编译命令可以用于避免重复包含头文件问题。

```
//module1.h
#ifndef MODULE1
#define MODULE1
...       //module1.h中的原文
#endif
```

  - 这样，在一个源文件中如果多次包含上面的module1.h文件，系统只会对第一次包含的内容进行处理。

main.cpp

```
#include <stdio.h>

#include "module1.h"
#include "module2.h"

int main()
{
      //……
}
```

module2.h

```
#include "module1.h"
double myFun(double, double);
```

module1.h

```
#define N 100
double mySin(double);
```

main.cpp

```
#include <stdio.h>

#define N 100
double mySin(double);
#define N 100
double mySin(double);
double myFun(double, double);

int main()
{
      //……
}
```

module2.h

```
#include "module1.h"
double myFun(double, double);
```

module1.h

```
#define N 100
double mySin(double);
```

main.cpp

```
#include <stdio.h>

#include "module1.h"
#include "module2.h"

int main()
{
      //……
}
```

module2.h

```
#include "module1.h"
double myFun(double, double);
```

module1.h

```
#ifndef MODULE1
#define MODULE1

#define N 100
double mySin(double);

#endif
```

main.cpp

```
#include <stdio.h>

#define MODULE1

#define N 100
double mySin(double);
#include "module2.h"

int main()
{
      //……
}
```

module2.h

```
#include "module1.h"
double myFun(double, double);
```

module1.h

```
#ifndef MODULE1
#define MODULE1

#define N 100
double mySin(double);

#endif
```

- 条件编译也可以用于多环境的程序编写。比如，

```
#if 'W'
  ……        //适合于WINDOWS环境的代码
#elif 'U'
  ……        //适合于UNIX环境的代码
#elif 'M'
  ……        //适合于MAC_OS环境的代码
#else
  ...       //适合于其他环境的代码
#endif

  ...       //与环境无关的公共代码
```

- 条件编译还可以用于程序的调试。

- 很多程序集成开发环境提供了动态调试工具，利用调试工具可以让程序执行一步停一下，或让程序执行到指定的断点停下，以便逐步观察一些变量的值是否与预期相符。

- 部分程序开发环境没有提供方便的调试工具，为了调试程序，程序员常常在程序中加一些输出语句，以便跟踪观察程序的执行情况。这种调试方式下，程序中的调试信息需要在开发结束后去掉。利用条件编译命令可以避免去掉调试信息的繁琐工作。

加入调试信息

```
#ifdef DEBUG
...    //调试信息，主要是输出
#endif
```

- 调试程序时，定义宏名DEBUG，调试结束，去掉宏名DEBUG的定义即可。

C++

- 函数名重载
- 带默认值的形式参数

函数名重载

- 对于一些功能相同、参数类型或个数不同的函数，有时给它们取相同的名字会带来使用上的方便。例：

```
void Print_int(int i) { ..... }
void Print_double(double d) { ..... }
void Print_char(char c) { ..... }
void Print_A(A a) { ..... } //A为自定义类型
```

定义为：

```
void Print(int i) { ..... }
void Print(double d) { ..... }
void Print(char c) { ..... }
void Print(A a) { ..... }
```

- C++规定：在相同的作用域中，可以用同一个名字定义多个不同的函数，这时，要求定义的这些函数应具有不同的参数
- 上述的函数定义形式称为**函数名重载**

对重载函数调用的绑定

- 确定一个对重载函数的调用对应着哪一个重载函数定义的过程称为绑定（binding，又称定联、联编、捆绑）。
  - 例：Print(1.0)将调用void Print(double d) { …… }

- 对重载函数调用的绑定在编译时刻由编译程序根据实参与形参的匹配情况来决定。从形参个数与实参个数相同的重载函数中按下面的规则选择一个：
  - 精确匹配
  - 提升匹配
  - 标准转换匹配
  - 自定义转换匹配
  - 匹配失败

精确匹配

- 类型相同
- 对实参进行“微小”的类型转换：
  - 数组变量名->数组首地址
  - 函数名->函数首地址
  - 等等
- 例如，对于下面的重载函数定义：

```
void Print(int);
void Print(double);
void Print(char);
```

下面的函数调用：

```
Print(1); 绑定到函数：void Print(int);
Print(1.0); 绑定到函数：void Print(double);
Print('a'); 绑定到函数：void Print(char);
```

提升匹配

- 先对实参进行下面的类型提升，然后进行精确匹配：
  - 按整型提升规则提升实参类型
  - 把float类型实参提升到double
  - 把double类型实参提升到long double
- 例如，对于下述的重载函数：

  void Print(int);

  void Print(double);

  根据提升匹配，下面的函数调用：

  Print('a'); 绑定到函数：void Print(int);

  Print(1.0f); 绑定到函数：void Print(double);

标准转换匹配

- **任何算术类型可以互相转换**
- **枚举类型可以转换成任何算术类型**
- **零可以转换成任何算术类型**或指针类型
- 任何类型的指针可以转换成void *
- 派生类指针可以转换成基类指针
- **每个标准转换都是平等的。**

绑定失败

- 例如，对于下述的重载函数：

  float sqrt(float);

  double sqrt(double);

  根据标准转换匹配，下面的函数调用：

  sqrt(2.56); 绑定到函数：double sqrt(double);

  而sqrt(256); 会绑定失败

- 如果不存在匹配或存在多个匹配，则绑定失败
  - 根据标准转换，256（属于int型）既可以转成float，又可以转成double

- 解决办法是：
  - 对实参进行显式类型转换，如，
    - sqrt((float)256)或sqrt((double)256)
  - 增加额外的重载，如，
    - 增加一个重载函数定义int sqrt(int);

带默认值的形式参数

- 在C++中允许在定义或声明函数时，为函数的某些参数指定默认值。如果调用这些函数时没有提供相应的实参，则相应的形参采用指定的默认值，否则相应的形参采用调用者提供的实参值。
- 例如，对于下面的函数声明：

```
void Print(int value, int base=10);
```

下面的调用：

```
Print(28); //28传给value；base为10
Print(32,2); //32传给value；2传给base
```

- 在指定函数参数的默认值时，应注意下面几点：
  - 有默认值的形参应处于形参表的右部。例如：
    - void F(int a, int b=1, int c=0); //OK
    - void F(int a, int b=1, int c); //Error
  - 对参数默认值的指定只在函数声明或定义处有意义。
  - 在不同的源文件中，对同一个函数的声明可以对它的同一个参数指定不同的默认值；在同一个源文件中，对同一个函数的声明只能对它的每一个参数指定一次默认值。

小结

- 多模块程序的设计
  - 文件包含命令
  - 头文件及其作用
  - 库函数
  - 标识符的作用域、存储期、名空间
- 与程序模块设计有关的优化
  - 带参数的宏定义
  - 内联函数
  - 条件编译
  - 函数名重载（C++）
  - 带默认值的形式参数（C++）

- 要求：
  - 理解程序多模块结构的相关概念
  - 能够实现多模块结构的程序，并分析其功能与结果
  - 继续保持良好的编程习惯
    - 头文件、实体名命名…

课程的主要内容

- 起步
  - Ch0
    - 初识C程序
    - 开始强调良好的编程习惯
- 进阶
  - 程序的流程控制Ch1
  - 程序的模块设计Ch2
    - C语言函数（子程序）
    - 多模块程序的设计
    - 程序模块设计的优化

  - 简单数据的操作与描述Ch3-4
  - 复杂数据的操作与描述Ch5-9

- 概览
  - 程序与程序设计的本质Ch10

- **重点：**
  - 分支、循环流程的控制方法及其运用
  - 函数的定义与调用
- **难点**
  - 嵌套的循环
  - 递归函数、作用域与存储期

Thanks !