C++程序设计精要教程

华中科技大学

第3章 语句、函数及程序设计

◆3.1 C++的语句

- 语句是用于完成函数功能的基本命令。
- 语句包括空语句、值表达式语句、复合语句、if语句、switch语句、for语句、while语句、do语句、break语句、continue语句、标号语句、goto语句等。
- 空语句：仅由分号“;”构成的语句。
- 值表达式语句：由数值表达式（求值结果为数值）加上分号“;”构成的语句。
  例如：x=1; y=2;
- 复合语句：复合语句是由“{}“括起的若干语句。例如：{x=1; y=2;}
- if语句：也称分支语句，根据满足的条件转向不同的分支。两种形式：
  - if(x>1) y=3; //单分支：当x>1时，使y=3
  - if(x>1) y=3; else y=4; //双分支：当x>1时，使y=3，否则使y=4

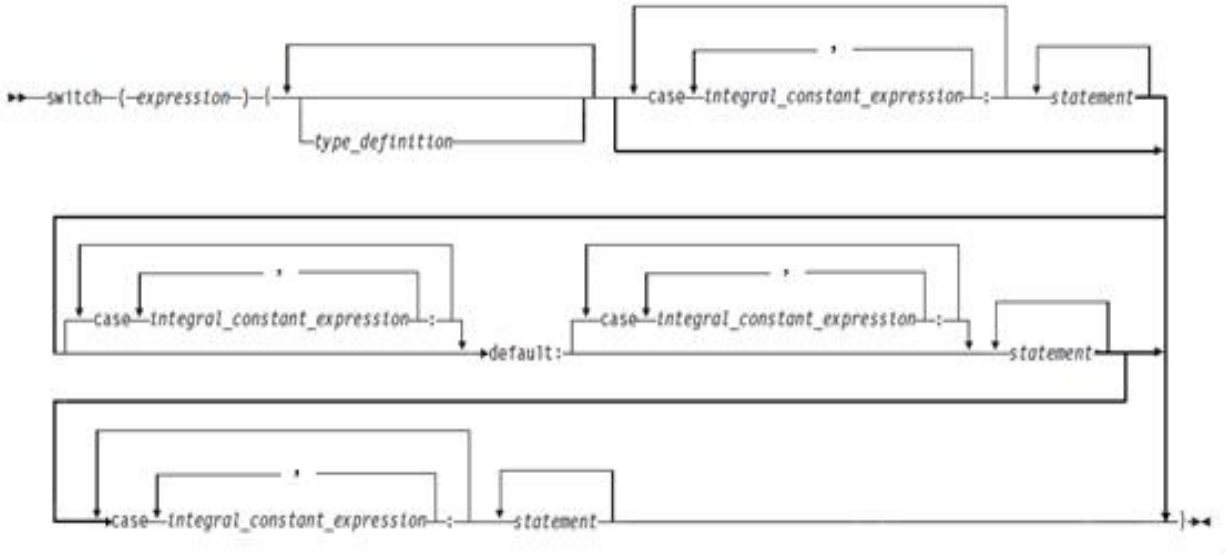
上述if语句红色部分可以是任何语句，包括新的if语句：称之为嵌套的if语句。

第3章 语句、函数及程序设计

巴科斯范式 以美国人巴科斯(Backus)和丹麦人诺尔(Naur)的名字命名的一种形式化的语法表示方法，用来描述语法的一种形式体系，是一种典型的元语言。又称巴科斯-诺尔形式(Backus-Naur form)。

◆3.1 C++的语句

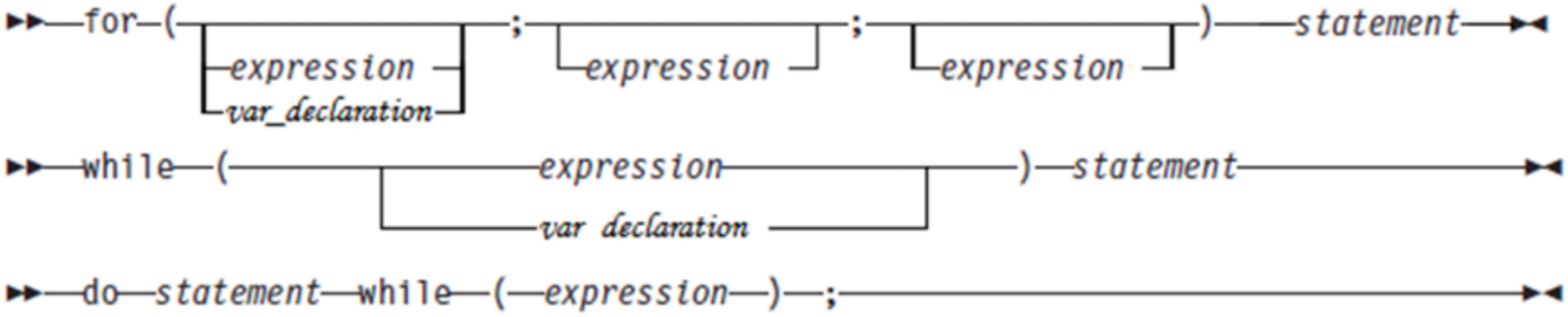
- switch语句：也称多路分支语句，可提供比if更多的分支。
- 括号中的expression只能是小于等于int的类型包括枚举。
- 进入swtich的“{}”，可以定义新的类型和局部变量
- bool, char, short, int等值均可。
- “default”可出现在任何位置。
- 未在case中的值均匹配“default”
- 若当前case的语句没有break，则继续执行下一个case直到遇到break或结束。
- switch的”()”及“{}”中均可定义变量，但必须先初始化再被访问。



第3章 语句、函数及程序设计（课堂略）

◆3.1 C++的语句

- 循环语句共三种类型：for循环，while循环和do循环。可相互转换。
- for语句常用于循环次数明确的循环，while和do仅有一个条件表达式。



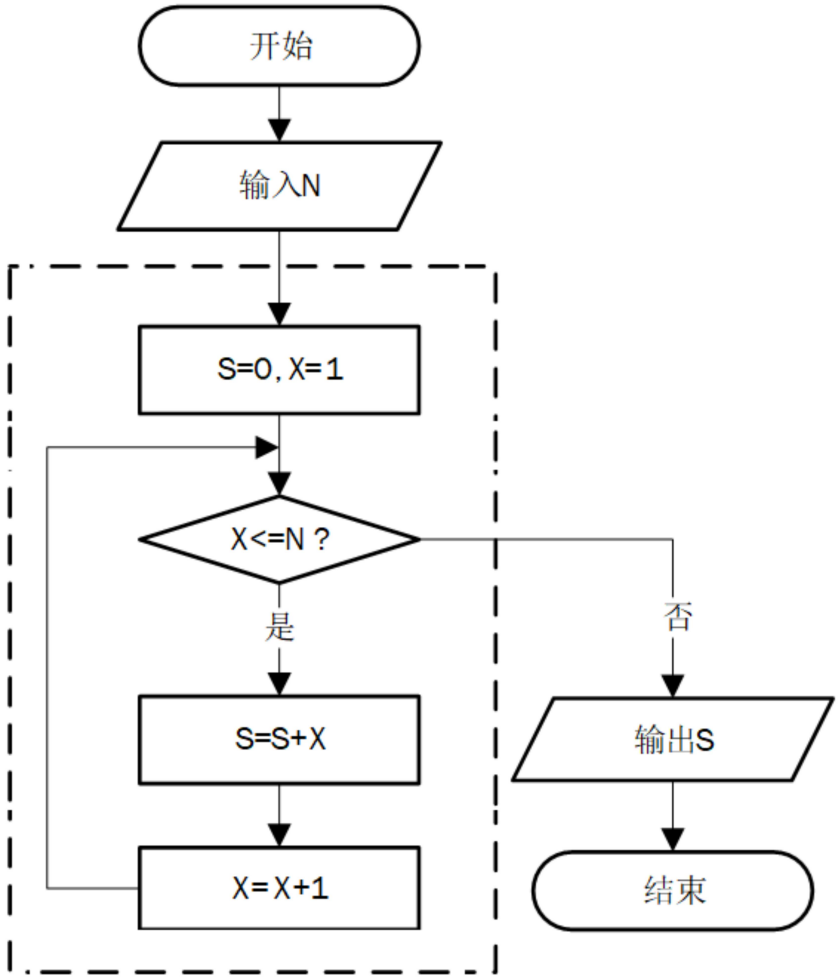
- while在条件满足时执行；而do先执行一次，再在条件满足下执行。
- 循环体是一条语句，可以是一条复合语句，或另一个循环语句。
- 当条件表达式永真或for无表达式，循环可以一直进行。for及while第一个表达式可以定义变量。

第3章 语句、函数及程序设计（课堂略）

◆3.1 C++的语句

●计算累加和 $S=\sum_{X=1}^{N} X$，其中N≥1。

```
#include <iostream>  //由于要用cin和cout，须include<iostream>
using namespace std;  //cin和cout在名字空间std中定义，须using
int main( ) {          //OS通过int型返回值知道main的执行状况
    int  N, X, S;      //定义累加边界N、循环变量X以及累加和S
    cout <<"Please input N: "; //通过<<输出提示要输入N
    cin >> N;                  //通过运算符重载函数>>输入N
    for(S=0, X=1; X<=N; X=X+1)//X=X+1可用++X或者X++表示
      S = S + X;       //将X累加至S，循环体是一条语句
    cout <<"\nThe cumulative sum S is "<< S << endl;
    return  0;                 //返回执行状态给操作系统，0表示
}
```



第3章 语句、函数及程序设计（课堂略）

◆3.1 C++的语句

- 循环语句可以相互转换。
- “for(S=0, X=1; X<=N; X=X+1) S=S+X;”可转换为：

```
S=0, X=1;
while(X<=N){//N=0不执行
    S = S + X;
    X=X+1;
}
```

```
S=0, X=1;
do{//如果N=0也执行一次
    S = S + X;
    X=X+1;
}while(X<=N);
```

- X=X+1可转换为X+=1 或者后置运算X++及前置运算++X。
- break可中断switch及循环语句的执行，转移至循环体外或switch的下一条语句执行。continue用于跳过后续语句，立即进入下一次循环。

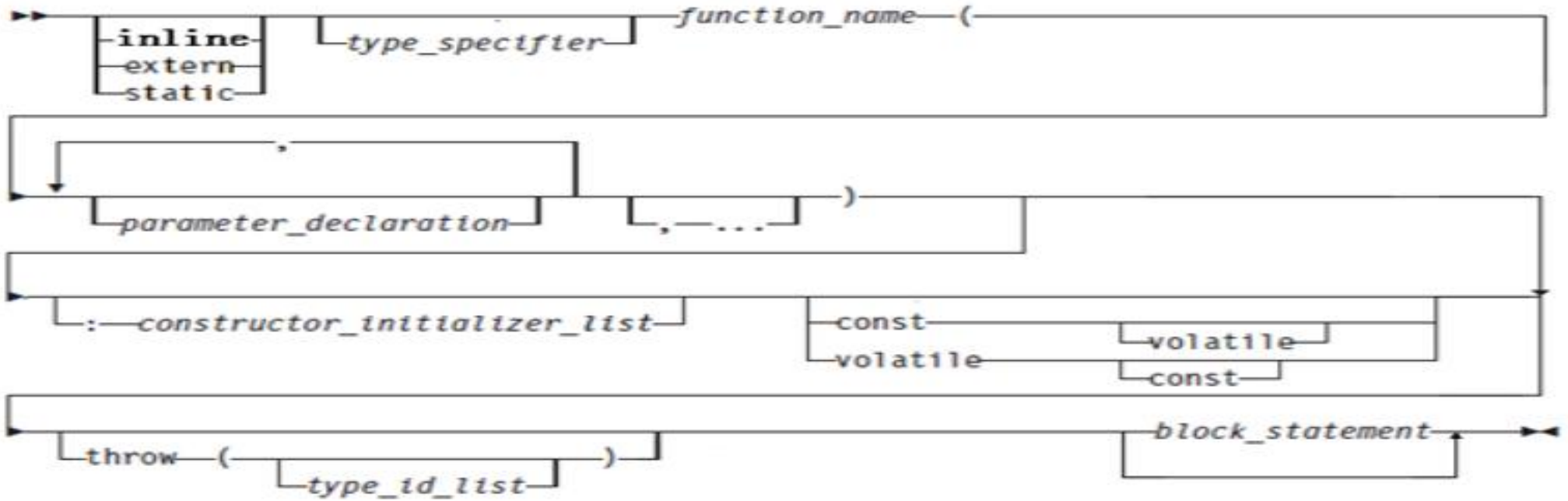
第3章 语句、函数及程序设计

◆3.2 C++的函数

- 函数用于“分而治之”的软件设计，以将大的程序分解为小的模块或任务。
- 函数说明（声明）不定义函数体，函数定义必须定义函数体。说明可多次，定义仅能实施一次。
- 函数可说明或定义为四种作用域：（1）全局函数(默认)；（2）内联即inline函数；（3）外部即extern函数；（4）静态即static函数
- 全局函数可被任何程序文件(.cpp)的程序用，只有全局main函数不可被调用(新标准)。故它是全局作用域的。
- 内联函数可在程序文件内或类内说明或定义，只能被当前程序文件的程序调用。它是局部文件作用域的，可被编译优化(掉）。
- 静态函数可在程序文件内或类内说明或定义。类内的静态函数不是局部文件作用域的，程序文件内的静态函数是局部文件作用域的。

第3章 语句、函数及程序设计

◆3.2 C++的函数



注意：没有函数体block_statment为函数说明，否则为函数定义

第3章 语句、函数及程序设计

◆3.2 C++的函数

- 函数说明（声明）可以进行多次，但定义只能在某个程序文件(.cpp)进行一次。

```
int d( ) { return 0; }                  //默认定义全局函数d：有函数体
extern int e(int x);                    //说明函数e：无函数体。可以先说明再定义，且可以说明多次
extern int e(int x) { return x; }       //定义全局函数e：有函数体
inline void f( ) { } //定义程序文件局部作用域函数f：有函数体，可优化，内联。仅当前程序文件可调用
void g( ) { }         //定义全局函数g：有函数体，无优化。
static void h( ){ } //定义程序文件局部作用域函数h：有函数体，无优化，静态。仅当前程序文件可调用
void main(void) {
    extern int d( ),  e(int);  //说明要使用外部函数：d, e均来自于全局函数。可以说明多次。
    extern void f( ),  g( );   //说明要使用外部函数：f来自于局部函数（inline）, g来自于全局函数
    extern void h( );          //说明要使用外部函数：h来自于局部函数（static）
}
```

第3章 语句、函数及程序设计

◆3.2 C++的函数

- 主函数main是程序入口，它接受来自操作系统的参数(命令行参数)。
- main可以返回0给操作系统，表示程序正常执行，返回其它值表示异常。
- main的定义格式为int main(int argc, char*argv[ ])，argc表示参数个数，argv存储若干个参数。
- 函数必须先说明或定义才能调用，如果有标准库函数则可以通过#include说明要使用的库函数的参数。
- 例如在stdio.h中声明了scanf和printf函数，分别返回成功输入的变量个数以及成功打印的字符个数：int printf(const char*, ...);
- 例如在string.h中声明了strlen函数，返回字符串s的长度(不包括字符串借宿标志字符‘\0’)：int strlen(const char *s);
- 注意在#include <string.h>之前，必须使用#define _CRT_SECURE_NO_WARNINGS

第3章 语句、函数及程序设计

◆3.2 C++的函数

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]) //定义char**argv也可
{//注意第1个参数为可执行程序.EXE的绝对路径名,第2个参数是传入的要求字符串长度的串
   if (argc != 2) {
      printf("The number of input string is wrong\n");
      return 1;                    //告知操作系统运行异常：可用于批命令程序的转移语句
   }
   int x = strlen(argv[1]);     //第1个参数argv[0]存放的是当前.EXE程序绝对路径名称
   printf("The lenth of the string is %d\n", x);
   return  0;                                  //告知操作系统运行正常
}
```

第3章 语句、函数及程序设计

◆3.2 C++的函数

- 省略参数...表示可以接受0至任意个任意类型的参数。通常须提供一个参数表示省略了多少个实参。

```
long sum(int n, ...) {
    long s = 0; int* p = &n + 1;   //p指向第1个省略参数
    for (int k = 0; k < n; k++)  s += p[k];
    return s;
}
void main( ) {
    int a = 4;  long s = sum(3, a, 2, 3); //执行完后s=9
}
```

- 注意：参数n和省略参数连续存放，故可通过&n+1得到第1个省略参数的地址；若省略参数为double类型，则应进行强制类型转换。

```
double *p=(double *)(&n+1);
```

第3章 语句、函数及程序设计

◆3.2 C++的函数

- 声明或定义函数时也可定义参数默认值，调用时若未传实参则用默认值。
- 函数说明或者函数定义只能定义一次默认值。
- 默认值所用的表达式不能出现同参数表的参数。
- 所有默认值必须出现在参数表的右边，默认值参数中间不能出现没有默认值的参数。VS2019的实参传递是自右至左的，即先传递最右边的实参.

```
int u=3;
int f(int x, int y=u+2, int z=3) { return x+y+z; }
int w=f(3)+f(2,6)+f(1,4,7); //等价于w=f(3,5,3)+f(2,6,3)+f(1,4,7);
```

- 若同时定义有函数int f(int m, ...); 则调用f(3)可解释为调用int f(int m, ...); 或调用int f(int x, int y=u+2, int z=3)均可，故编译会报二义性错误。

函数缺省参数

缺省参数：函数声明或定义时参数有缺省值。调用时若没有传入实参，就取参数的缺省值传给形参。

可用任意类型的表达式指定参数的缺省值，但表达式中不能出现同一参数表的参数；否则，由于参数的计算顺序问题(国际标准每规定自左至右或自右至左)，对不同编译会带来可移植性问题。

```
int f(int x, int y=x++); //错误：表达式有同参数表的参数x
```

对于x=3，f(x)表示使用参数y的缺省值进行调用，等价于f(x, x++)。自左至右计算参数等价于f(3, 3)，自右至左计算参数等价于f(4, 3)，故对不同的编译器来说是不可移植的。

所有缺省参数必须出现在非缺省参数的右部；

不能同时在声明和定义中定义缺省参数的(即使表达式相同)值。

```
int  w=3, b(int x=w);       //声明正确，指定缺省值x=w
int  u=++w;
int b(int x=w){return x;}    //再次定义则错，两w的值相等吗?
```

第3章 语句、函数及程序设计

◆3.2 C++的函数

- 编译会对内联inline函数调用进行优化，即直接将其函数体插入到调用处，而不是编译为call指令，这样可以减少调用开销，提高程序执行效率。
- 调用开销是指为完成调用所进行的实参传递、重要寄存器保护及恢复以及返回时的栈指针恢复到调用前的值所额外编译或执行的指令。
- 若f1、f2的调用开销分别为10、7个指令，函数体指令数分别为5、20，程序对f1和f2均有100个位置调用。则调用f1、f2编译后的指令数：
  - 成功内联f1=100*5，不内联f1=10*100+5=1005: 函数体小，内联合算
  - 成功内联f2=100*20，不内联f1=7*100+20=720: 函数体大，调用合算
  - 由此可见：函数体相对较小的函数，使用内联更合算。
- 若函数为虚函数、或包含分支(if, switch,?:,循环,调用)，或取过函数地址，或调用时未见函数体（函数体定义在调用处后面），则内联失败。失败不代表程序有错，只是被编译为函数调用指令。

内联函数

```
inline  double  girth (double r) ; //函数原型声明：无函数体
inline  double  area (double r) {   //函数定义：包括函数体
        return  3.1416*r*r;
} //inline函数只在当前文件可见
void main (void)  {//main内联不能做OS入口(OS要求全局作用域main)
        double   m;
        m=girth (5.0) ; //内联失败，编译为函数调用指令
        m=area (5.0) ; //内联成功，编译为m= 3.1416*5*5
}
double girth (double r)   { return  3.1416*2*r; }//声明时用了inline
                                                  // 定义时可不用inline修饰
```

函数重载

函数原型：用于描述函数的名称、参数和返回类型。参数只须说明参数类型，参数名称可不说明。调用时根据函数原型检查实参和形参是否相容。

```
double  sin(double x);          //有参数名的原型声明
double  cos(double);            //无参数名的原型声明
```

重载函数：通过**参数差异**识别重载函数，即若**参数的个数或者类型有所不同（至少一个参数类型不一致）**，则同名的函数被自动视为重载函数。

重载只与参数类型有关，与返回类型无关，若参数个数和类型完全相同、仅仅返回类型不同是不允许的。

函数重载

```
#include <iostream>
using  namespace std;
long GetTime(void){ return 1; }
long GetTime(int &hours, int &minutes, int &seconds){… ; return 1; }
void main(void){
        int  h, m, s;
        cout<<"Now is "<<GetTime(h, m, s) ;
        cout<<" seconds from midnight,";
        cout<<"Or "<<h<<":"<<m<<":"<<s<<"\n";
}
```

函数重载

当使用缺省参数和省略号时，函数重载容易引起歧义

```
void show (char * message){ … }
void show (char * message ,int code = 0){ …}
void show (char * message , …){ …}
show ( “Hello” ); //到底调用哪个?
```

在编译时编译器根据调用函数的实参决定使用哪个版本的重载函数（静态绑定，这也是为什么重载也被称为静态多态）。但这个例子编译器无法找到合适的函数入口地址，因此静态绑定失败。编译器会报错。

第3章 语句、函数及程序设计

◆3.3 作用域

- 程序可由若干代码文件(.cpp)构成，整个程序为全局作用域：全局变量和函数属于此作用域。
- 稍小的作用域是代码当前文件作用域：函数外的static变量和函数属此作用域。
- 更小的作用域是函数体：函数局部变量和函数参数属于此作用域。
- 在函数体内又有更小的复合语句块作用域。
- 最小的作用域是数值表达式：常量在此作用域。
- 除全局作用域外，同层不同作用域可以定义同名的常量、变量、函数。但他们为不同的实体。
- 如果变量和常量是对象，则进入面向对象的作用域。
- 同名变量、函数的作用域越小、被访问的优先级越高。

名字的作用域

每个名字（标识符：由程序员命名的名字）都会指向一个实体：变量、函数、类型等。

但同一个名字处于不同的位置，也可能指向不同的实体，这是由名字 的作用域(scope)决定的。C++绝大多数作用域用{}分割。

- 定义于所有花括号之外名字具有全局作用域，在整个程序范围内可见；
- 定义于{}之内的名字，其作用域始于名字的声明语句，结束于所在的{}的最后一条语句；这样的名字的作用域是块作用域(block scope);
- 作用域可以嵌套。作用域一旦申明了一个名字，它所嵌套的所有作用域都能访问该名字，同时允许在内部嵌套作用域内定义同名名字。
  - 作用域越小，访问优先级越高；因此内部嵌套作用域内的名字会隐藏外层作用域里同名名字；
  - 函数的形参相当于函数里定义的局部变量，其作用域是函数体；

名字的作用域

```
#include <iostream>

int gi = 1; //i具有全局作用域

void func(){
        //具有全局作用域的变量gi到处可以访问,输出0
        std::cout << gi << std::endl;

        int j = 10; //j的作用域是函数f
        {
                int j = 100; //嵌套作用域，可以定义同名变量j

                //嵌套作用域里的j隐藏了外面作用域里的j，输出100
                std::cout << j << std::endl;
        }

        std::cout << j << std::endl; //这时访问的是外层作用域里的j，输出10
}
```

第3章 语句、函数及程序设计

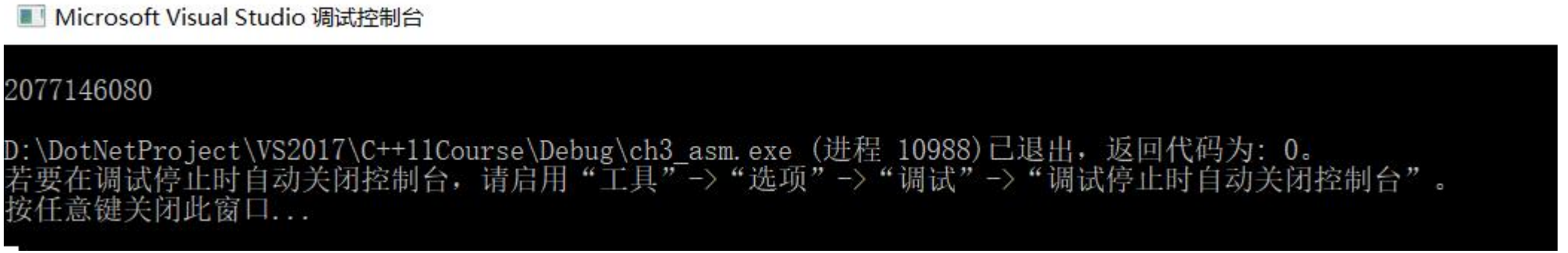
◆3.4 生命期

- 作用域是变量等可使用的空间，生命期是变量等存在的时间。
- 变量的生命期从其被运行到的位置开始，直到其生命结束（如被析构或函数返回等）为止。
- 常量的生命期即其所在表达式。
- 函数参数或自动变量的生命期当退出其作用域时结束。
- 静态变量的生命期从其被运行到的位置开始，直到整个程序结束。
- 全局变量的生命期从其初始化位置开始，直到整个程序结束。
- 通过new产生的对象如果不delete，则永远生存（内存泄漏）。
- 外层作用域变量不要引用内层作用域自动变量（包括函数参数），否则导致变量的值不确定：因为内存变量的生命已经结束（内存已做他用）。

外层作用域变量不要引用内层作用域自动变量

```
#include <iostream>

//返回内层作用域的自动变量
int & g() {
        int x = 10;
        return x;
}
int main()
{
        int & r = g(); //外层作用域引用变量r引用了g里面的自动变量
        std::cout << r << std::endl; //g返回后，自动变量弹出堆栈，生命周期已经结束
        //打印的是不确定的值
}
```



第3章 语句、函数及程序设计

【例3.24】试分析常量、变量和函数的生命期和作用域。代码文件“A.cpp”：

```
int x=2;                          //全局变量：生命期和作用域为整个程序
static int y=3;                   //模块静态变量：生命期自第一次访问开始至整个程序结束
int f( )                          //全局函数f()：其作用域为整个程序，生命期从调用时开始
{
   int u=4;                       //函数自动变量：生命期和作用域为当前函数
   static int v=5;                //函数静态变量：生命期自第一次调用开始至整个程序结束
   v++;
   return u+v+x+y;
}
static int g( ) { return x; }     //静态函数g()：其作用域为“A.cpp”文件
```