# 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

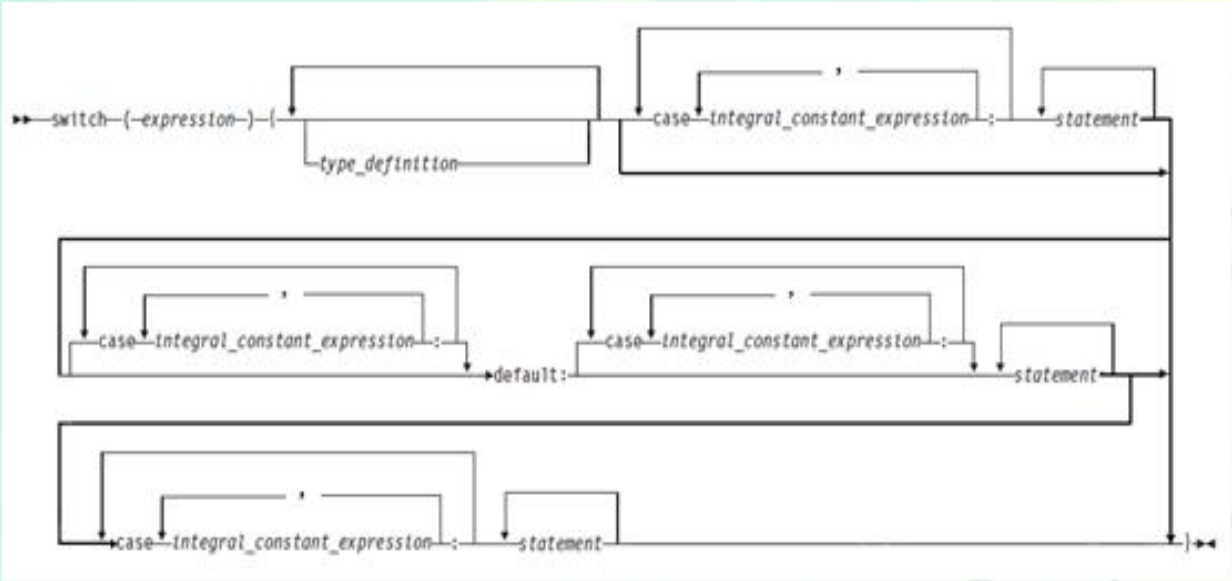
- 语句是用于完成函数功能的基本命令。
- 语句包括空语句、值表达式语句、复合语句、if语句、switch语句、for语句、while语句、do语句、break语句、continue语句、标号语句、goto语句等。
- 空语句：仅由分号“;”构成的语句。
- 值表达式语句：由数值表达式加上分号“;”构成的语句。例如：x=1; y=2;
- 复合语句：复合语句是由“{ }“括起的若干语句。例如：{ x=1; y=2; }
- if语句：也称分支语句，根据满足的条件转向不同的分支。两种形式：
  - if(x>1) y=3; //单分支：当x>1时，使y=3
  - if(x>1) y=3; else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

## ◆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。
- expression只能是不大于 long long 的整型类型包括枚举。
- 进入switch的“{ }”，不能定义新的局部变量
- 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2章 类型、常量及变量

```
#include <iostream>
int main()
{
    long long k = 0;
    switch( k + 1 )
    {
        case 1:
            int j1 = k + 1;
            std::cout << j1;
        case 2:
            int j2 = k + 2;
            std::cout << j2;
        default:
            std::cout << "Other";
    }
}
```

X

```
#include <iostream>
int main()
{
    long long k = 0;
    switch( k + 1 )
    {
        int j1 = k + 1;
        int j2 = k + 2;
        case 1:
                std::cout << j1;
        case 2:
                std::cout << j2;
        default:
                std::cout << "Other";
    }
}
```

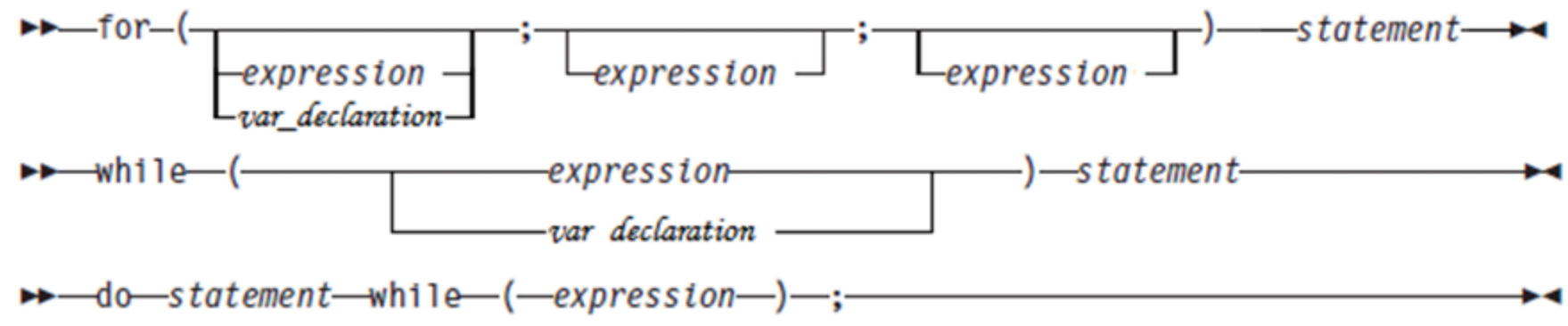
X

```
#include <iostream>
int main()
{
    long long k = 0;
    int j1 = k + 1;
    int j2 = k + 2;
    switch( k + 1 )
    {
        case 1:
                std::cout << j1;
        case 2:
                std::cout << j2;
        default:
                std::cout << "Other";
    }
}
```

# 第3章 语句、函数及程序设计

## ◆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。



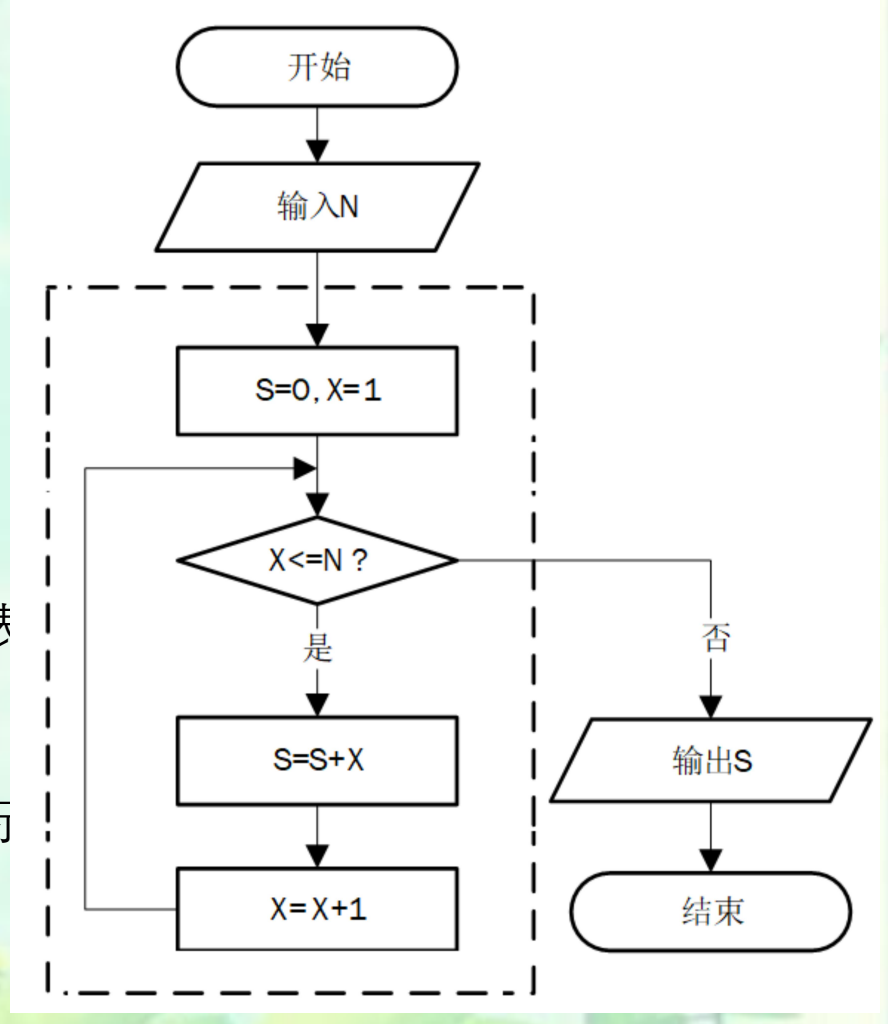

- ●while在条件满足时执行；而do先执行一次，再在条件满足下执行。
- ●循环体是一条语句，可以是一条复合语句，或另一个循环语句。
- ●当条件表达式永真或for无表达式，循环可以一直进行。for及while第一个表达式可以定义变量。

# 第3章 语句、函数及程序设计

## ◆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
        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

## ◆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

## ◆3.1 C++的语句

- asm语句可在C或C++程序中插入汇编代码。VS2019编译器使用“_asm”插入汇编代码。
- static_assert用于提供静态断言服务，即**在编译时判定执行条件是否满足。**使用格式为 static_assert(条件表达式，”输出信息“)。不满足则编译报错。

```
#include <stdio.h>
void f( ) {                              //说明函数f()没有返回值
    _asm mov EAX, 3                      //使用32位寄存器EAX保存整数3
} //VS2019用EAX保存返回值，相当于定义：int f( ){ return 3; }
void main(void) {                        //说明函数main不返回值
    int (*pf)( ) = (int(*)( ))f;         //强制类型转换：f返回整型值
    static_assert(sizeof(int)==4, “32 bit compiler”);  //注意：sizeof编译时可计算
    printf("return=%d",(*pf)( ));        //调用f( )函数，打印返回值3
    _asm mov EAX, 0;                     //相当于定义int main(void)，并返回0：程序正常
}
```

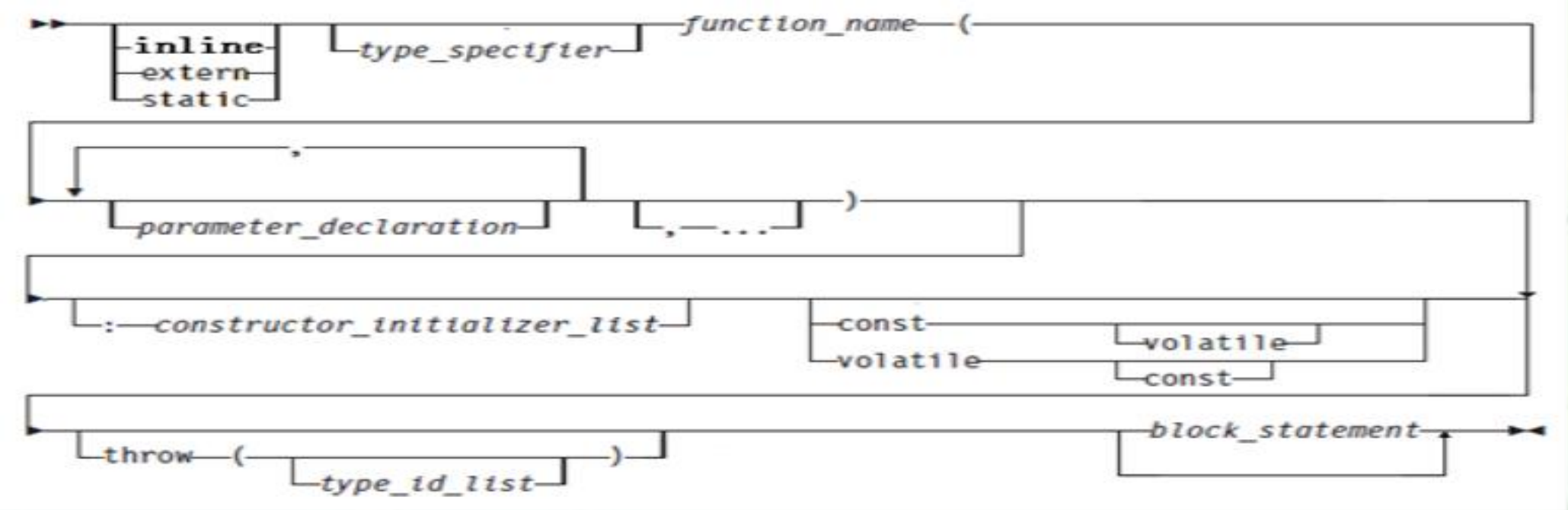
# 第3章 语句、函数及程序设计

## ◆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。它是文件局部文件作用域的，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文件局部文件作用域的，程序文件内的静态函数是文件局部文件作用域的。**

# 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，其他模块不能定义e()的函数体
inline void f( ) { }      //函数f：有函数体，不优化，内联。仅当前程序文件可调用
void g( ) { }             //全局函数g：有函数体，无优化。
static void h( ) { }      //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, 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
}
```

# 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中定义了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中定义了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
{  //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
}
```

- 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。

# 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，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)，或取函数地址，或调用时未见函数体，则内联失败。内联失败不代表程序有错，只是被编译为函数调用指令。

# 第3章 语句、函数及程序设计

```
inline int sum(int a, int b, int c)
{
    return a + b + c;
}

int main()
{
    int x = sum(1,2,3);
    int y = sum(4,5,6);
}
```

```
int main()
{
    int x = 1 + 2 + 3;
    int y = 4 + 5 + 6;
}
```

```
main proc
        mov  x, 3
        add   x, 2
        add   x, 1
        mov  y, 6
        add   y, 5
        add   y, 4
        ret
main endp
```

```
int sum(int a, int b, int c)
{
    return a + b + c;
}

int main()
{
    int x = sum(1,2,3);
    int y = sum(4,5,6);
}
```

```
sum proc
        push ebp
        mov  ebp, esp
        mov  eax, [ebp+08h]   ;a
        add   eax, [ebp+0ch]    ;a+b
        add   eax, [ebp+10h]    ;a+b+c
        pop   ebp
        ret
sum endp

main proc
{       push  3
        push  2
        push  1
        call    sum
        mov    x, eax    ;x = sum(1,2,3);
        push  6
        push  5
        push  4
        call    sum
        mov    y, eax    ;y = sum(4,5,6);
        ret
main endp
```

# 第3章 语句、函数及程序设计

**constexpr 是 C++11中新增的关键字，表示它修饰的对象是常量。**基本的常量表达式：字面值、全局变量/函数的地址、sizeof等关键字返回的结果。

**constexpr 用来修饰：变量、函数的返回值、构造函数。用 constexpr 定义的变量不能重新赋值（具有const 特性）。**

**(1) constexpr 修饰变量**

- **constexpr变量，必须在定义时赋值，并且编译时计算出其值为常量；**
- **constexpr变量，如果不是用函数调用赋值，则等价于const；**
- **constexpr变量，如果用函数调用赋值，则函数必须是constexpr且在编译时计算出的返回值是常量；**

**例如：constexpr int x = expression (常量表达式); 等价：const int x = …**

**constexpr int y = f( ); //f()必须是constexpr且编译时返回值是常量.**

**Problem: 在任何情况下，可以使用 constexpr 替换 const 吗？**

# 第3章 语句、函数及程序设计

## (2) constexpr 修饰函数的返回值

- 在调用时，一般需要将函数返回值赋值给一个constexpr变量，这时表示在编译时就能计算出constexpr函数的返回值，返回值一定是常量（否则报错）；
- 如果调用时，函数返回值没有赋值给一个constexpr变量，则等价于该函数没有constexpr属性。

例如：

```
constexpr  int  f(int x) { return x+1; }
int  a = 1;
f(1);  f(a);                    //正确，等价于f()没有constexpr属性
const  int x1 = f(1);     //正确，等价于f()没有constexpr属性
const  int x2 = f(a);     //正确，等价于f()没有constexpr属性
constexpr  int y1 = f(1);  //正确，编译为 constexpr  int y1 = 2;
constexpr  int y2 = f(a);  //错误，编译时不能算出f(a) (因为a不是常量)
```

# 第3章 语句、函数及程序设计

**(2) constexpr 修饰函数的返回值**

- constexpr函数内部调用函数时，被调用函数必须是constexpr；
- constexpr函数内部不能有goto语句或标号，也不能有try语句块；
- constexpr函数内部不能定义或使用static变量、线程本地变量等永久期限变量；
- constexpr函数作用域相当于static；
- 函数main为全局作用域，故不能定义为constexpr函数。

# 第3章 语句、函数及程序设计

**(3) constexpr 修饰类的构造函数**

**constexpr 构造函数必须将所有没有缺省值的成员变量放到初始化列表中，函数体内的内容必须全部是 constexpr 的。**

# 第3章 语句、函数及程序设计

```
#include <iostream>
constexpr int f(int n)    //调用此函数时，在编译时就能确定其返回值
{
   return n + 1;
}
int main()
{
   constexpr int a = f(1);                  //对：编译时能计算出 a 的值
   constexpr int b = a * 2 + 1;             //对：编译时能计算出 b 的值
   constexpr int c = f( std::cin.get() );   //错：编译时不能计算出 c 的值
   int d = f( std::cin.get() );             //对：不需要编译时计算出 d 的值，运行时计算d的值
   int k = 5;
   int x1[k];                               //错：数组维数不能为常规的变量（维数必须在编译时就能确定）
   int x2[b];                               //对：编译为 x[5]
   b = 0;                                   //错：不能重新对b赋值
}
```

(1) 删除f()定义中的constexpr，main()函数中的各条语句是否正确？

(2) 将 constexpr int a = f(1) 改成 int a = 1 或 const int a = 1, mian()中的各语句正确否？

# 第3章 语句、函数及程序设计

Problem：(1) 将 int v = 10 改成 const int v = 10，结果会怎样？

(2) 将 constexpr A c(1) 改成 constexpr A c(0)，结果会怎样？

```
#include <iostream>
struct A {
    int  x;
    int  y = 9;
    constexpr A(int a): x(a) { }                          //对：所有成员变量要么有缺省值、要么在初始化列表中
    constexpr A(int a, int b): x(a) { y = b; x += y; }    //对：函数体内是常量表达式
    constexpr A(int a): x(a) { std::cout << x + y; }      //错：函数体内调用了非constexpr函数
    constexpr A(int a): y(a) { x = x + y;  }              //错：初始化列表中没有x
    A( ): x(1), y(2) { std::cout << x+y; }                //对：会生成普通的非constexpr对象
};
int main()
{   int v = 10;
    constexpr A a(v);     //错：非const变量的值不能在编译时确定(运行时才能确定), 因此a.x不能在编译时确定
    constexpr A b;                //错：构造函数A()不能产生constexpr对象
    constexpr A c(1);             //对：生成constexpr对象c
    constexpr A d(1, 2);          //对：生成constexpr对象d
    A e(1), f;                    //对：e(1)=去除constexpr属性, f=构造函数A()产生普通对象（非constexpr）
    A g( );                       //对：声明一个函数g()，其返回值是A的对象
    int z1[c.x], z2[d.x+d.y];     //对：c、d是constexpr的，c.x、d.x+d.y的值在编译时确定，等价 z1[1]、z2[5]
    int z3[e.x];                  //错：e是普通对象（非constexpr的），e.x的值在编译时不能确定
    enum { X = c.x, Y = c.y};     //X = 1, Y = 9
    c.y = 10;                     //错：c是constexpr的，不能改变其成员
    e.y = 10;                     //对：e是普通对象
    f.y = 10;                     //错：对象f不存在
}
```

# 第3章 语句、函数及程序设计

## ◆3.2 C++的函数

- 与**进程**独立分配不同的是，**线程**可以共享同一个程序的内存变量。
- 如果多个线程操作同一个变量，则程序的行为变得不可预料。
- 但如果线程之间需要互斥，则这些线程必须共享一个互斥锁变量。
- 被锁住的线程代码不能并发执行，类std::mutex可用来定义互斥锁变量。
  - 基于作用域的 std::lock_guard，当作用域结束时自动解锁
  - 基于致命区的加开锁，用加锁lock与开锁unlock锁住一段致命代码
- 主函数main在使用 std::thread 类创建线程对象后，便会启动和执行被线程对象关联的函数。main需要等待其他线程结束，否则可能内存泄露。
- 线程本地变量：用 thread_local 定义的变量在每个线程对象启动后，都会为该变量分配内存并初始化或调用构造函数，使每个线程对象都有独立隔离的关于该变量的内存。

# 第3章 语句、函数及程序设计

```
#include <stdio.h>
#include <thread>
#include <mutex>
std::mutex mtx; //共享锁变量：用于线程互斥
struct S
{
    int i = 0;
    inline static int p; //默认为0, 不能在类体外定义p
    S( ) {
        mtx.lock( );    //加锁
        printf("S( ) called，  i=%d\n", i);//致命区
        mtx.unlock( );              //开锁
    }
};

//以下gs三个线程各有一份：main, a, b
thread_local S gs; //线程本地变量gs
```

```
void foo( ) {
    mtx.lock( );        //加锁
    gs.i += 1;          //开始执行致命代码区代码
    printf("In foo，gs is at %p, gs.i=%d\n", &gs, gs.i);
    mtx.unlock( );    //解锁
}

void bar( ) {  //以下语句加锁直到当前作用域结束
    std::lock_guard<std::mutex> lock(mtx);
    gs.i += 4;
    printf("In bar，gs is at %p, gs.i=%d\n", &gs, gs.i);
}  //std::lock_guard在当前作用域结束自动解锁;

int main( ){
    std::thread a(foo), b(bar);  //创建线程和foo、bar关联
    a.join( );     //等待线程对象a结束后main继续执行
    b.join( );    //等待线程对象b结束后main继续执行
    printf("In main，gs is at %p, gs.i=%d\n", &gs, gs.i);
}
```

# 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。
- **常量对象的作用域是常量对象所在的指令行。**
- **作用域越小、被访问的优先级越高。**
- **inline、static 修饰的变量和函数，作用域是模块文件内部，不同的模块可定义同名的 全局 或 inline、static 变量和函数**

# 第3章 语句、函数及程序设计

## ◆3.3 作用域

假设1个项目（程序）由文件 A.CPP 和 B.CPP 组成，A.CPP 的内容如下。

```
extern int y;                          //y是在B.CPP中定义的
int x = 1;                             //定义全局变量x，只能在A.cpp或B.cpp中共计定义一次
static int u = 2;                      //模块静态变量u，A.cpp或B.cpp均可定义各自的同名变量
static int v = 3;
static int g( ) {  return x+y;  }      //A::x  +  B::y
inline int h( )  { return u; }
static int m( ) { return 1; }
int f( ) {                             //作用域范围越小，被访问的优先级越高
   int u = 4;                          //函数局部非静态变量：作用域为函数f内部
   static int v = 5;                   //函数局部静态变量：作用域为函数f内部
   return u+v+x+ ::u;                  //f::u  +  f::v  +  A::x  + A::u
}
```

# 第3章 语句、函数及程序设计

## ◆3.3 作用域

B.CPP 的内容如下。

```
extern int f( );                          //f( ) 是在A.cpp定义的
extern int x;                             //x是在A.cpp定义的
int y = 10;                               //y是全局变量
static int u = 20;                        //A.cpp和B.cpp都定义了u
static int g( ) {  return x+y+u;  }       //A::x  +  B::y + B::u
inline int h( )  { return u; }

int main( ) {
    g( );              //B::g( )
    h( );              //B::h( )
    f( );              //A::f( )
    m( );              //error
    return  v;         //error
}
```

# 第3章 语句、函数及程序设计

## ◆3.3 作用域

- 函数体内的{ }构成块作用域：复合、switch、循环等语句。
- 同层块作用域可以定义同名变量，但他们是不同实体，值互相独立。
- 同层块作用域不能定义同名标号。故VS2019允许跨块转移，但转移位置必须在变量定义及初始化之前
- VS2019允许向内层块转移，但转移位置必须在变量定义及初始化之前。
- 外层作用域的变量不要引用内层作用域的自动变量（包括函数参数），否则导致变量的值不确定。
- 全局变量和static变量永久存储在数据段，局部自动变量和函数参数存在于栈段，单值常量又称立即数理论上没分配内存。包含多个元素的常量（如对象、数组）实际上在数据段存储，但理论上认为没分配内存。

# 第3章 语句、函数及程序设计

## ◆3.4 生命期

- 作用域是变量等存在的空间，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）。

# 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，B.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;          //f()的生命期在此结束
}
static int g( ) { return x; }  //静态函数g()：其作用域为A.cpp文件，生命期从调用时开始
```

# 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“B.cpp”：

```
extern int x;
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
extern int f( );
static int g( )                 //静态函数g()：其作用域为B.cpp文件，生命期从调用时开始
{
   return x + y++;              //x由A.cpp定义，y由B.cpp定义
}
void main( )                    //全局函数main()：其生命期和作用域为整个程序
{
   int a = f( );                //函数自动变量a：生命期和作用域为当前函数
   const int &&b = 2;           //传统右值无址引用变量b引用常量2：产生匿名变量存储2
   a = f( );                    //main()开始全局函数f()的生命期
   a = 3;                       //常量3的生命期和作用域为当前赋值表达式
   a = g( );                    //main()开始 B.cpp 的静态函数g()的生命期
} //为b产生的匿名变量的生命期在main()返回时结束
```