華中科技大學

第 3 章 语句、函数及程序设计

许向阳

xuxy@hust.edu.cn

3.1 C++的语句

3.1.1 简单语句

3.1.2 转移语句

3.1.3 分支语句

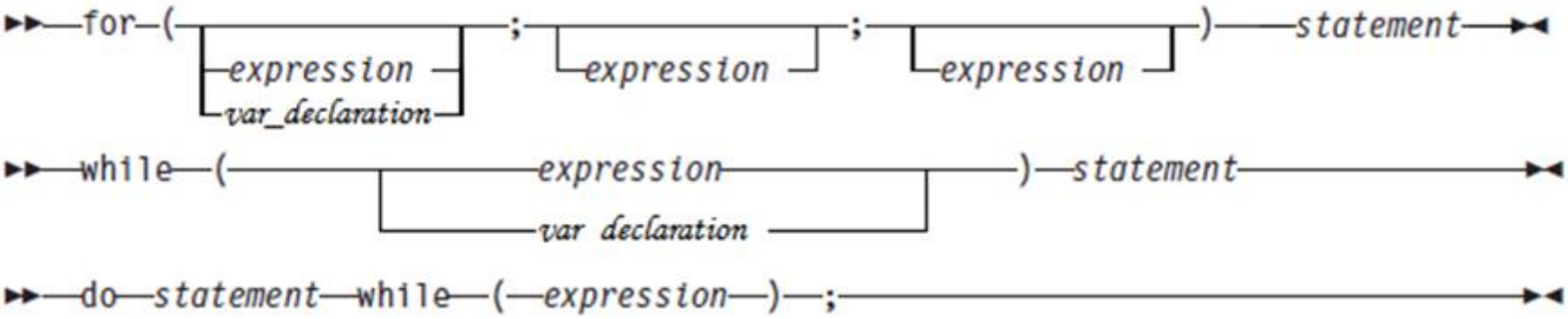
3.1.4 循环语句

3.1.5 break和continue语句

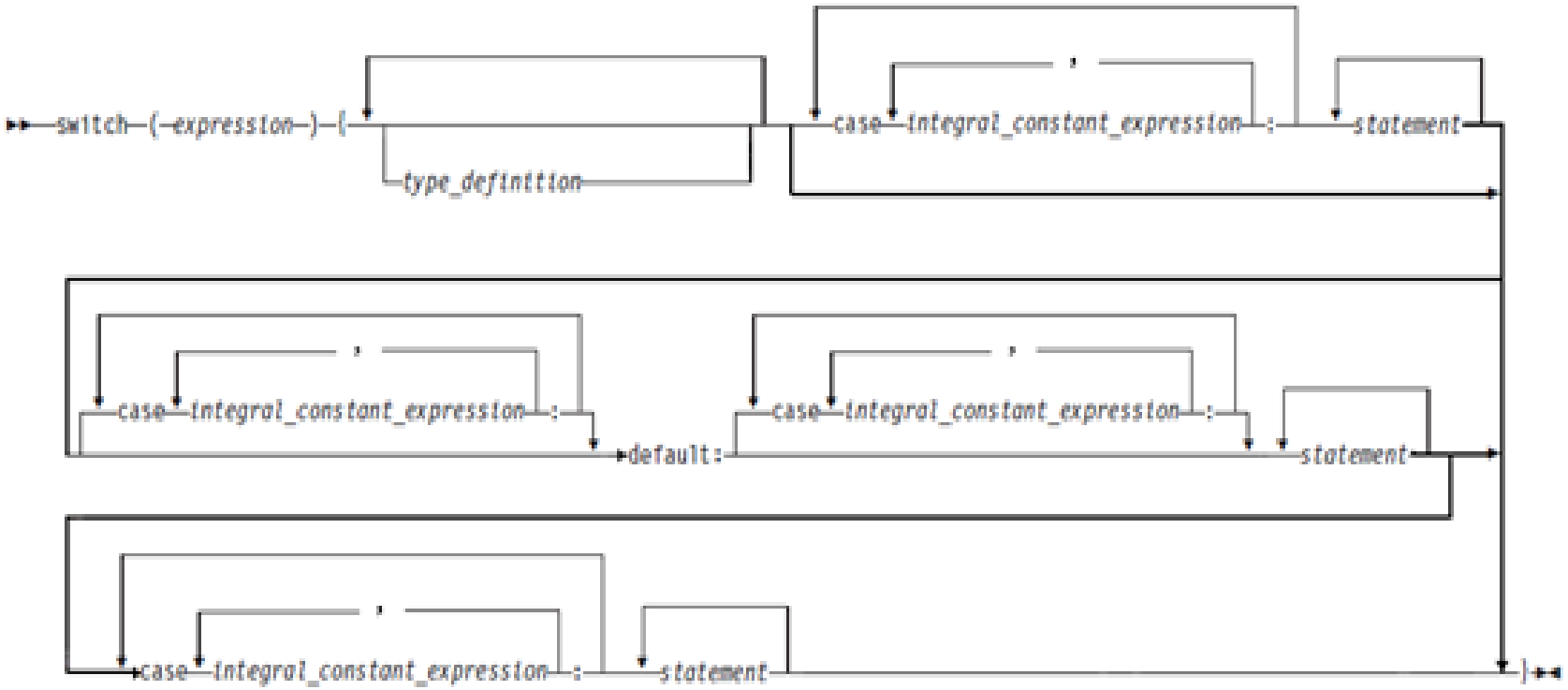
3.16 asm 和 static_assert语句

空语句、值表达式语句、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、复合语句等。

循环语句的语法图


for (; ;) statement
expression
var_declaration
expression
expression
while ( ) statement
expression
var declaration
do statement while (expression) ;

switch语句的语法图



```
int y=1;
switch (y)
{
   case 1: printf("hello\n");
                //   break;
   case 2:
      printf("good\n");
      break;
   default:
      printf("welcome\n");
}
```

Q: 漏写 case 1处的break, 运行结果是什么？

Q: 若将default 提到 case 1 之前, 运行结果是什么？

Q: switch的执行过程是什么？

- **expression 是计算结果不大于long long的整型表达式；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 “{ ……}” 中可定义变量。**

3.1 C++的语句

if

Q: if (条件) { S1….} else { S2…..}

if (! 条件） { S2….} else { S1…..}

两个分支，谁在前谁在后，有何讲究？

Q: if (条件1 && 条件2) ……

两个条件，谁在前谁在后，有何讲究？

Q: if (条件) {……}

else if ( ) {…… }

else { …….} 避免多层嵌套

3.1 C++的语句

If 与 switch

Q: if 语句能否转换为 switch 语句?

Q: switch 语句能否转为 if 语句?

Q: 能否给出例子，将多分支语句 转换为 无分支语句?

3.1 C++的语句

Example: if (x>0) { ……}

```
switch (x > 0)  {
    case true: ……
}
```

Example: if (x>0) { ……} else {…..}

```
switch (x > 0)  {
    case true: ……              // case 1 :
               break;
    case false:……              // case 0 :
}
```

3.1 C++的语句

Example: 统计一个字符数串中各个小写字母出现的次数

```
char  s[…]=“……” ;
int count[26];
switch (s[i]) {
        case  ‘a’: count[0]++;
                   break;
        case ’b’: count[1]++;
                   break;
         .......
}

count[s[i]- ‘a’ ]++;
```

3.1 C++的语句

Example: 当 i =0时，执行函数 fadd;
当 i=1时，执行函数 fsubtract

```
int fadd(int a, int b)
{   cout << "add";      return a + b;  }

int fsubtract(int a, int b)
{   cout << "subtract";  return a - b; }
```

```
int (*q)(int, int) ;      // 定义一个函数指针变量
q = fadd;            q=&fadd;    // 等同写法

int (*p[2])(int, int) = { fadd, fsubtract };      函数地址表
int z = p[i](20, 30);    // i=0，执行fadd
```

3.1 C++的语句

断言

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，不满足则编译报错，输出相应的信息。

static_assert(条件表达式，”输出信息”);

程序运行之中的断言，条件不满足，运行异常中断。

#include <assert.h>

assert(条件表达式);

Q: 在程序的何处使用 断言，有什么好处？

3.2 C++的函数

3.2.1 函数说明与定义

3.2.2 头文件与说明

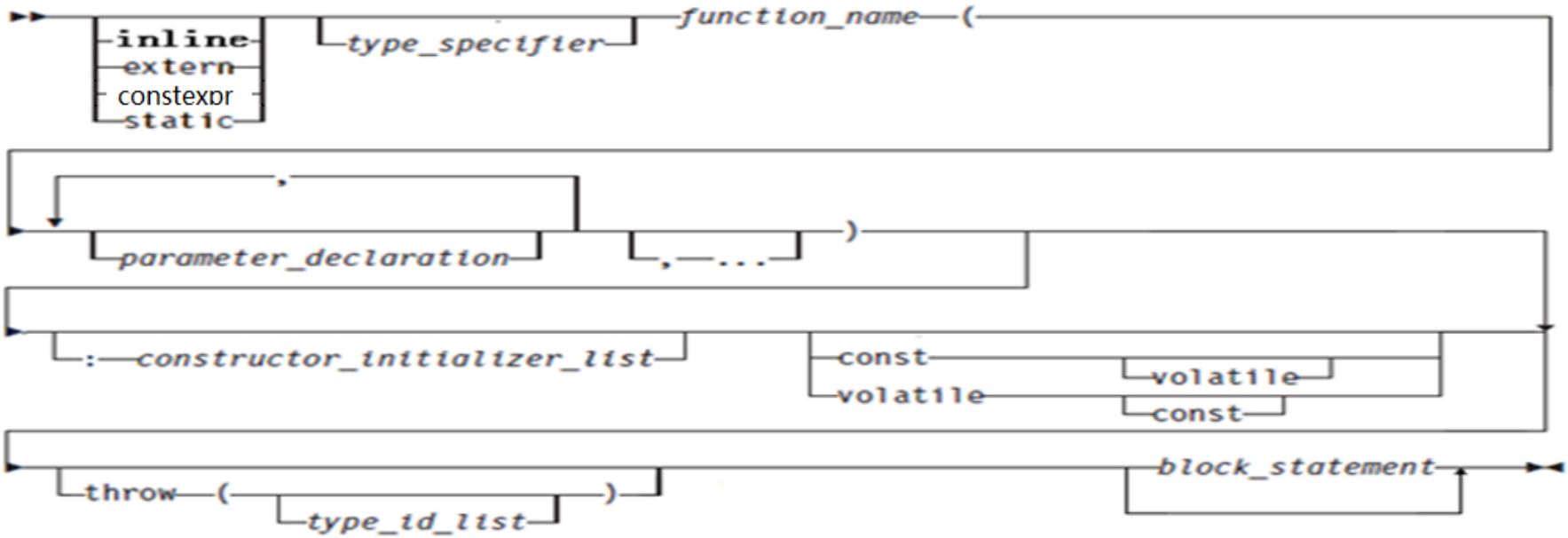
3.2.3 函数的参数说明

3.2.4 函数重载

3.2.5 inline及constexpr函数

3.2.6 线程互斥及线程本地变量

3.2.1 函数说明与定义



函数说明或定义：

（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；

（5）constexpr函数。

3.2.1 函数说明与定义

全局函数： 可被任何程序文件(.cpp)的程序调用，
只有全局main函数不可被调用。

内联函数： inline

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，
它是文件局部作用域的，可被编译优化(掉）。

静态函数： static

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作用域的，
程序文件内的静态函数是文件局部作用域的。

3.2.2 头文件与说明

- 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

```
#include <stdio.h>
  int printf(const char*, ...); // 返回成功打印的字符个数
#include <string.h>
    int strlen(const char *s);
     // 返回字符串长度,不包括字符串结束字符‘\0’
```

调用老版本函数，注意在文件最开头加：

```
#define _CRT_SECURE_NO_WARNINGS
```

3.2.3 函数的参数说明

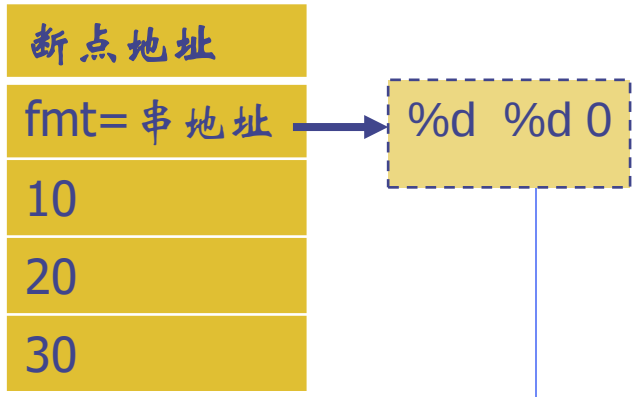
省略参数 .....

int printf(const char *fmt, ...);

...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。

Q：下面函数执行的结果是什么？

```
printf(“%d  %d ”,10, 20,30);
printf(“%d  %d”, 10);
```



3.2.3 函数的参数说明

编写n个整数求和，n是可变的

```
int s=sum(3,4,5,6);    / /执行完后s=15
    s=sum(2,10,20);    / /执行完后s=30

int sum(int n, …)
 {
     int  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;
}
```

3.2.3 函数的参数说明


```
int sum(int n,  …)
 {   int  s=0;
     int  *ap;
     ap=&n+1;
     for (int k=0; k<n; k++)
          s+=ap[k];
     return s;
}
```

```
int sum(int n, ...)
{  int s = 0;
   va_list  ap;
   va_start(ap, n);
   for (int k = 0; k < n; k++)
          s += va_arg(ap, int);
   va_end(ap);
   return s;
}
```

```
typedef char* va_list;
va_start(ap, n);
va_arg(ap, int);
va_end(ap);
```

```
char   *ap;
 ap=&n+1;
s+=*ap;    ap +=4;
 ap = 0;
```

3.2.3 函数的参数说明

va_start(ap, n);　　// ap = (char *) &n +4;

```
#define va_start   __crt_va_start
#define __crt_va_start(ap, x)
((void)(__vcrt_assert_va_start_is_not_reference<decltype(x)>(),
__crt_va_start_a(ap, x)))
 #define __crt_va_start_a(ap, v)
     ((void)(ap = (va_list)_ADDRESSOF(v) + _INTSIZEOF(v)))
```

va_arg(ap, int);　　// ap += 4;
*(int *)(ap-4)

```
#define va_arg    __crt_va_arg
#define __crt_va_arg(ap, t)
     (*(t*)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)))
```

3.2.3 函数的参数说明


```
void  DrawCircle(int x, int y, int r=10)
{
        .......
}

DrawCircle(100, 100, 20);

DrawCircle(100, 100);
```

缺省参数

如果某个参数给了缺省值，其右边的参数都需要给参数值。

思考：编译器会如何处理？

3.2.3 函数的参数说明


```
void  DrawCircle(int x, int y, int r=10)
{          .......
}
DrawCircle(100, 100);
 push        0Ah
 push        64h
 push        64h
 call       DrawCircle
 add         esp,0Ch
```

缺省参数

在编译函数调用语句时，生成了默认参数的传递语句。

通过将默认的实参值传递给形参，实现形参的初始化。

3.2.3 函数的参数说明


```
void f(int u, int v)
{
    cout << "u= " << u << "  v = " << v << endl;
}          // 显示 u =3,   v=4

int main()
{
   int  x = 3, y = 4;
   f(x++, y++);
   cout << "x= " << x << "  y = " << y << endl;
   return 0;          // 显示 x=4   y =5
}
```

3.2.3 函数的参数说明

華中科技大學

```
f(x++, y++);
mov       eax,dword ptr [y]
mov       dword ptr [ebp-0DCh],eax  //  y的值拷贝到一个临时空间
mov       ecx,dword ptr [y]         //  实现 y = y+1
add       ecx,1
mov       dword ptr [y],ecx
mov       edx,dword ptr [x]
mov       dword ptr [ebp-0E0h],edx  //  x的值拷贝到一个临时空间
mov       eax,dword ptr [x]         //  实现 x = x+1
add       eax,1
mov       dword ptr [x],eax
mov       ecx,dword ptr [ebp-0DCh]  //  从临时空间取数作为参数
push      ecx
mov       edx,dword ptr [ebp-0E0h]  //  从临时空间取数作为参数
push      edx
call      f (0C910E6h)
```

3.2.3 函数的参数说明

左值引用

```
void fr(int &u, int &v)
{
    cout << "u= " << u << "  v = " << v << endl;
    u = 20;   v = 30;
}                              // 显示 u =3,   v=4

int main()
{
   int  x = 3, y = 4;
   fr(x, y);
   cout << "x= " << x << "  y = " << y << endl;
   return 0;                   // 显示 x=20   y =30
}
```

3.2.3 函数的参数说明

左值引用

```
void fr(int &u, int &v)
{
    cout << "u= " << u << "  v = " << v << endl;
    u = 20;   v = 30;
}                              // 显示 u =4,   v=5

int main()
{
   int  x = 3, y = 4;
   fr(++x, ++y);
   cout << "x= " << x << "  y = " << y << endl;
   return 0;                   // 显示 x=20   y =30
}
```

3.2.3 函数的参数说明

左值引用

```
void fr(int &u, int &v)
{
    cout << "u= " << u << "  v = " << v << endl;
    u = 20;   v = 30;
}                              // 显示 u =4,   v=5

int main()
{
   int  x = 3, y = 4;
   fr(x++, y++);  // 语法错误，无法从int转换为int &
   cout << "x= " << x << "  y = " << y << endl;
   return 0;
}     // 想要传递一个临时对象的地址
```

3.2.3 函数的参数说明

右值引用

```
void frr(int &&u, int &&v)
{
    cout << "u= " << u << "  v = " << v << endl;
    u = 20;   v = 30;
}                              // 显示 u =3,   v=4

int main()
{
   int  x = 3, y = 4;
   frr(x++, y++);
   cout << "x= " << x << "  y = " << y << endl;
   return 0;
}              // 显示 x =4,   y=5
```

3.2.3 函数的参数说明

frr(x++, y++);

右值引用

```
mov         eax,dword ptr [y]
mov         dword ptr [ebp-0E0h],eax
mov         ecx,dword ptr [y]
add         ecx,1
mov         dword ptr [y],ecx
mov         edx,dword ptr [x]
mov         dword ptr [ebp-0ECh],edx
......
lea         ecx,[ebp-0E0h]    // 传递临时对象的地址
push        ecx
lea         edx,[ebp-0ECh]
push        edx
call        frr (0CE1168h)
```

3.2.3 函数的参数说明

右值引用：通过加 const 约束的左值引用来实现

```
void fcr(const int & u, const int & v)
{       cout << "u= " << u << "  v = " << v << endl;
        // 不能修改 u、 v 引用的对象
}

int  x = 3, y = 4;
fcr(x, y);                    // 三条语句均正确
fcr(x++, y++);
fcr(5, 6);
```

3.2.3 函数的参数说明

对比不加 const 约束的左值引用

```
void fcr(int & u, int & v)
{       cout << "u= " << u << "  v = " << v << endl;
        // 不能修改 u、 v 引用的对象
}

int  x = 3, y = 4;
fcr(x, y);

fcr(x++, y++);    // 无法将参数1从 int 转换为 int &
fcr(5, 6);
```

3.2.4 函数重载

- 函数名相同
- 参数个数或者参数类型有所不同

不能有相同函数名、相同参数，但是返回类型不同的函数

3.2.4 函数重载

Q：如下申明和定义是否会导致编译错误，为什么？

```
float g(int);              // 函数申明
int   g(int);
int   g(int, int y = 3);
int   g(int, ...);

int   i = g(8);            //  变量定义
```

3.2.5 inline 及constexpr函数

華中科技大學

3.2.6 线程互斥及线程本地变量


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

int counter = 0;
mutex mtx;
void increase(int time) {
    for (int i = 0; i < time; i++) {
        // 当前线程休眠1毫秒
        this_thread::sleep_for(chrono::milliseconds(1));
        mtx.lock();
        counter++;
        mtx.unlock();
        cout << " " << counter << " ";
    }
}
```

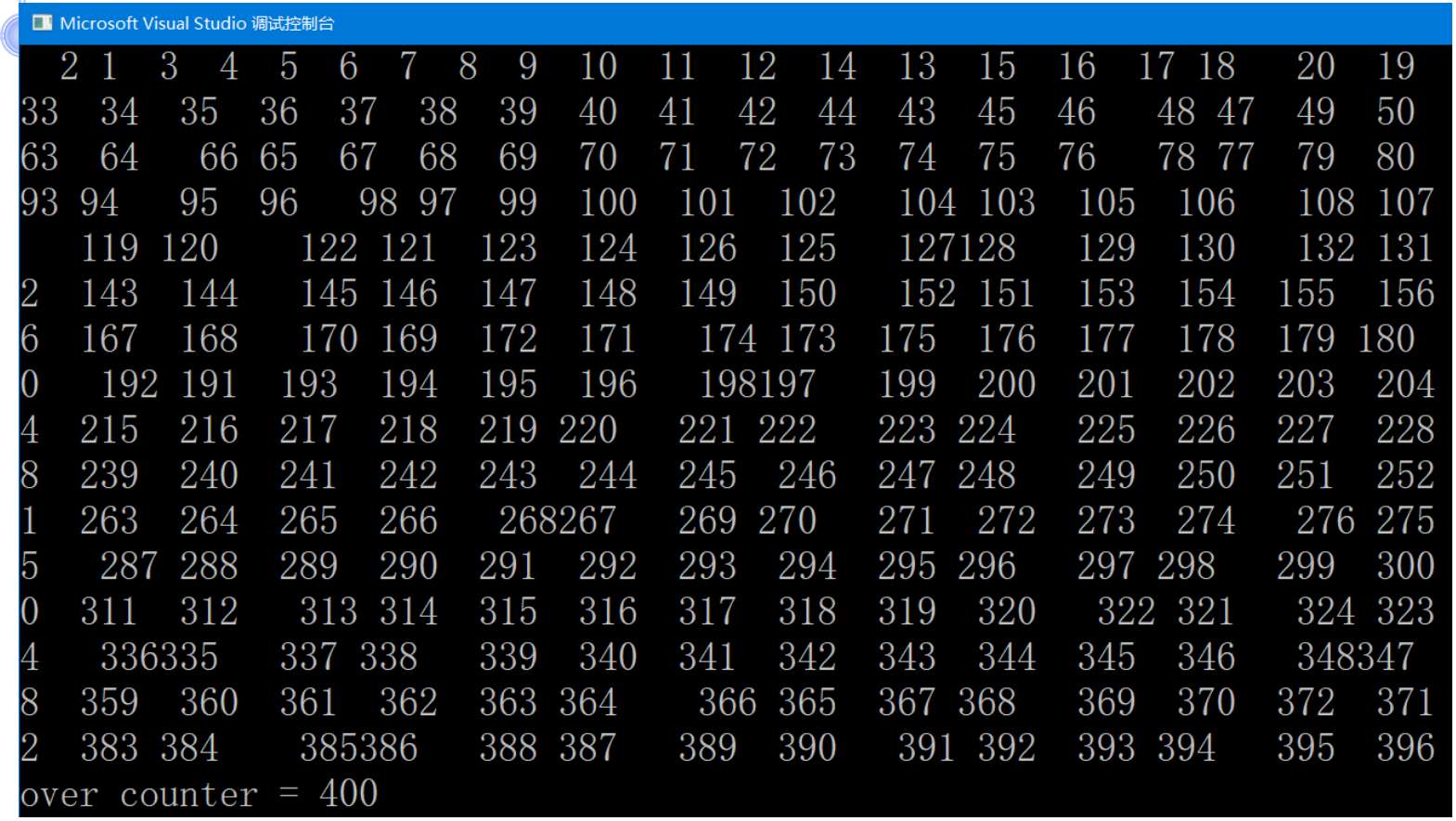
- 多线程共享一个全局变量 **counter**

不对counter加锁，运行结果有随机性。

3.2.6 线程互斥及线程本地变量

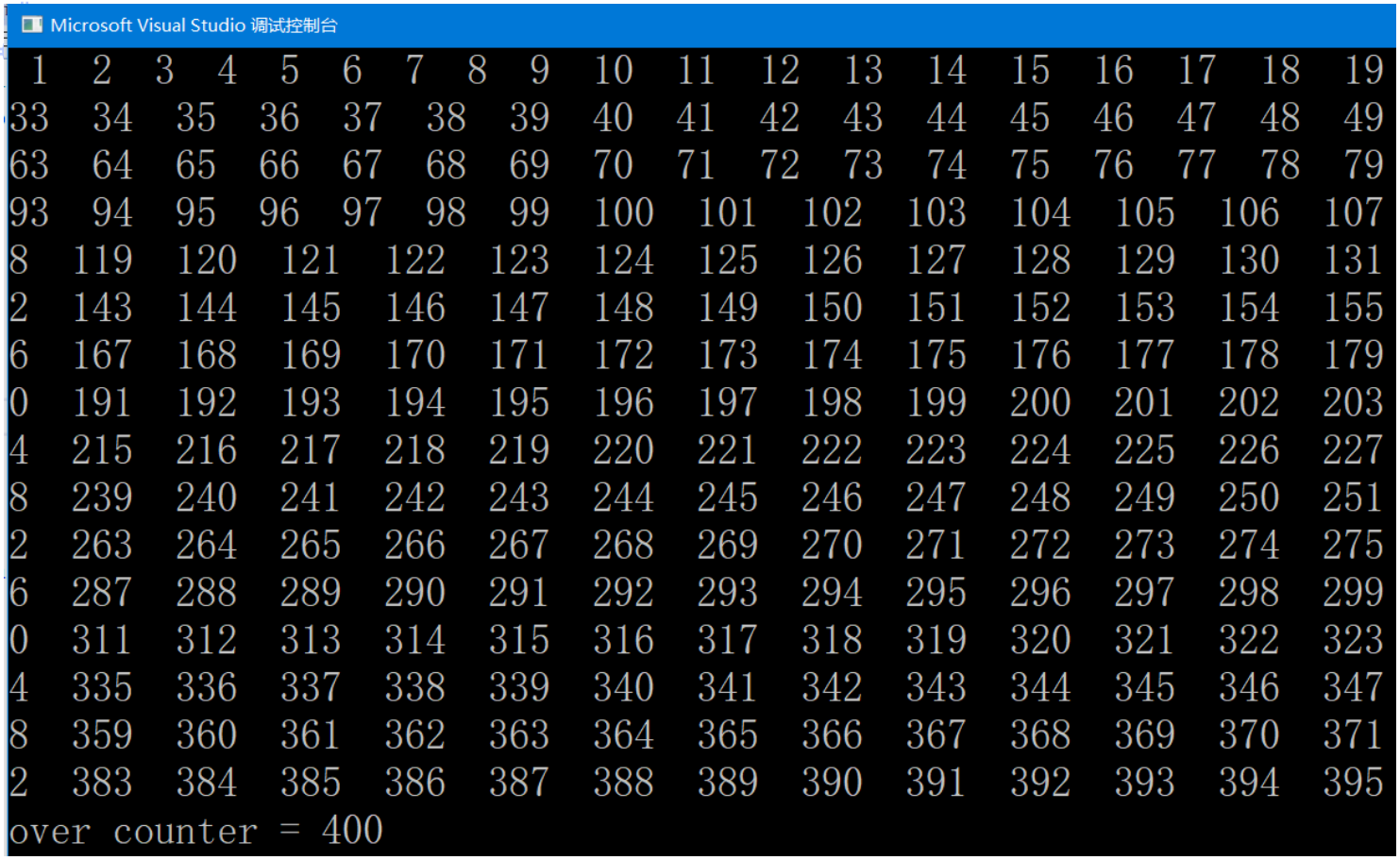

```
int main(int argc, char** argv) {
    std::thread t1(increase, 200);
    std::thread t2(increase, 200);
    t1.join();
    t2.join();
    cout <<endl<<"over counter = "<<counter<<endl;
    return 0;
}
```

3.2.6 线程互斥及线程本地变量



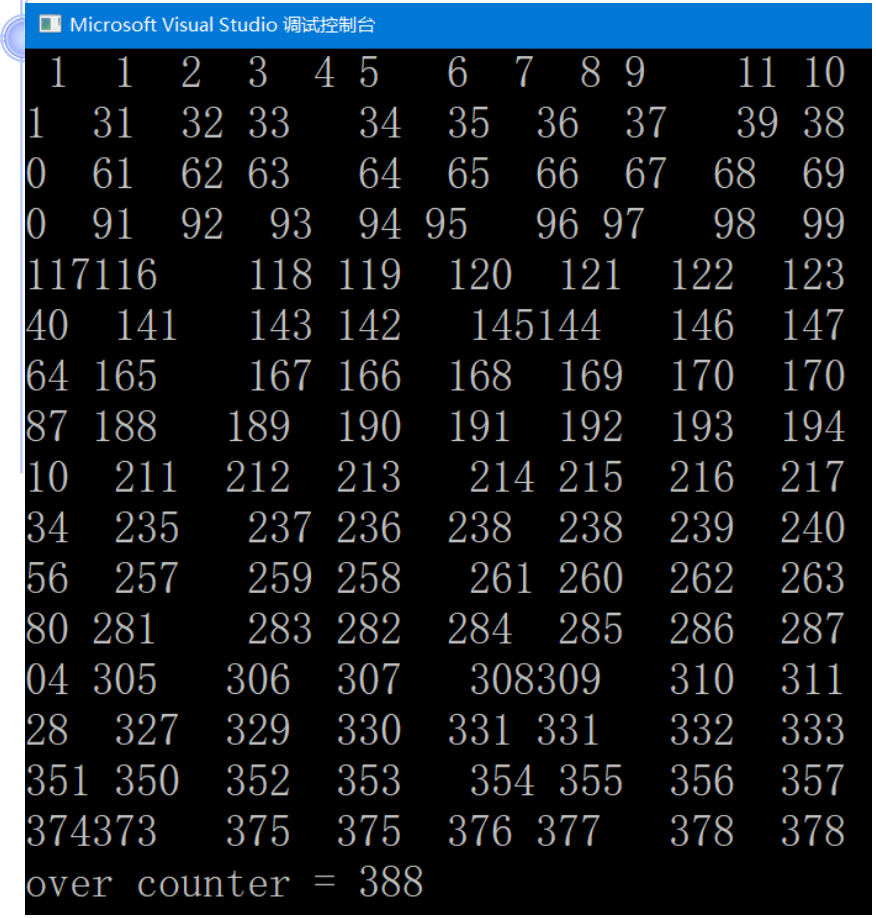
Counter的总数是 400，但显示的顺序不一定，如 2 在1前，20在19前等等

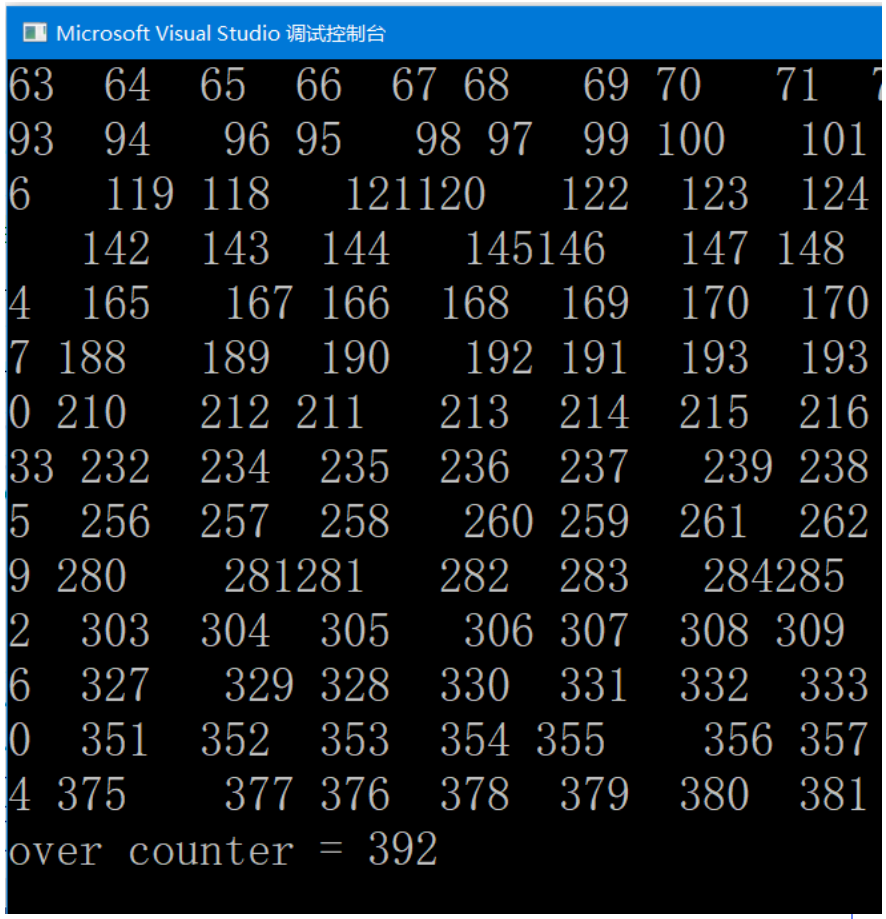
3.2.6 线程互斥及线程本地变量



将increase函数中的 counter 显示也加入封锁段，显示完全是顺序的。

3.2.6 线程互斥及线程本地变量





不加锁，运行结果有 随机性。

3.2.6 线程互斥及线程本地变量


```
thread = 2  counter =389
thread = 1  counter =390
thread = 1  counter =391
thread = 2  counter =392
thread = 2  counter =393
thread = 1  counter =394
thread = 2  counter =395
thread = 1  counter =396
thread = 1  counter =397
thread = 2  counter =398
thread = 1  counter =399
thread = 2  counter =400

over counter = 400
```

加锁，增加了一个 thread 编号参数

```
void increase(int time, int threadno )
```

3.2.6 线程互斥及线程本地变量

线程本地变量：

每个线程都独立的为该变量分配空间。

```
thread_local int counter = 0;
```

```
thread = 1  counter =197
thread = 1  counter =198
thread = 2  counter =198
thread = 2  counter =199
thread = 1  counter =199
thread = 2  counter =200
thread = 1  counter =200

over counter = 0
```

线程 t1, t2, 以及主线程，都为counter分配空间。

3.2.6 线程互斥及线程本地变量


```
thread_local int counter = 0;
```

```
thread = 2  counter =3
thread = 1  counter =3
thread = 2  counter =thread = 14  counter =
4
thread = 2thread =   counter =51
  counter =5
thread = 1  counter =6
thread = 2  counter =6
thread = 2  counter =7thread = 1
  counter =7
thread = 2  counter =thread = 1  counter =8
8
thread = 1  counter =9
thread = 2  counter =9
```

去掉了加锁；

线程 t1, t2各自的counter会按序增加；
但显示时被穿插打乱

3.3 作用域

3.3.1 全局作用域与模块作用域

3.3.2 局部作用域与块作用域

3.3 作用域

华中科技大学

全局作用域

程序可由若干代码文件构成，整个程序为全局作用域：全局变量和函数属于此作用域。

代码文件作用域

函数外的static变量和函数属此作用域。

函数体作用域

函数局部变量和函数参数属于此作用域。

3.3 作用域

全局作用域　　代码文件作用域　　函数体作用域

复合语句块作用域

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。

可在不同作用域定义同名变量

同名变量、函数的作用域越小、被访问的优先级越高。

如果变量和常量是对象，则进入面向对象的作用域。

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。

3.4 生命期

- 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）。

3.4 生命期

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）。

3.5 程序设计实例

3.5.1 栈编程实例

3.5.2 队列编程实例

3.5.3 有限自动机编程实例

3.5.1 栈编程实例


```
#include <iostream>
using namespace std;
struct STK {
    int* e;
    int  v;
    int  t;
};

void create(STK* const stk, int v)
{
    stk->e = (int*)malloc(sizeof(int)*v);
    stk->v = stk->e ? v : 0;
    stk->t = 0;
}
```

3.5.1 栈编程实例

```
STK* const push(STK* const stk, int x)
{
    if (stk->t >= stk->v) return 0;
    stk->e[stk->t++] = x;
    return stk;
}

int main()
{
    STK  s;
    STK s1;
    create(&s, 10);
    create(&s1, 10);
    push(&s, 5);

    push(push(push(&s, 9), 10), 15);
    *push(&s, 20) = s1;
    return 0;
}
```

Q：能否在push 中修改 stk？

Q：为何函数返回 STK *？

Q：去掉 push前的const有何影响？

3.5.1 栈编程实例

Q: 执行 *push(&s, 20) = s1;

s 中的数据会有何变化?

```
STK * push = stk; //  STK  * const stk = &s;
```

*push=s1; <=> *stk = s = s1;

监视 1

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
s	{e=0x015d0e50 {-842150451} v=10 t=0 }	STK
e	0x015d0e50 {-842150451}	int *
v	10	int
t	0	int
s1	{e=0x015d0e50 {-842150451} v=10 t=0 }	STK
e	0x015d0e50 {-842150451}	int *
v	10	int
t	0	int
&s	0x0135fb00 {e=0x015d0e50 {-842150451} v=10 t=0 }	STK *
&s1	0x0135faec {e=0x015d0e50 {-842150451} v=10 t=0 }	STK *

3.5.1 栈编程实例

Q: 能否写 push(&s, 15) = &s1; ?

= 的左操作数必须为左值。

无论 push前是否加 const, 它自动的是 const.

准确的类比

```
STK * const push = stk; //  STK  * const stk = &s;
    push=&s1;
```

编程练习：

让程序自动找出 人、狼、羊、草 安全过河的方法。