



华中科技大学

第3章 语句、函数及程序设计

许向阳

xuxy@hust.edu.cn





内容及难点

3.1 C++的语句

3.2 C++的函数

3.3 作用域

3.4 生命周期

- 分支语句、循环语句向 无分支、无循环的转换
不同写法的执行效率以及可读性
- 函数指针（是变量）
- 缺省参数、参数个数不定的实现机理
- &、&& 参数的理解
- 函数返回的理解
- 多线程编程





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

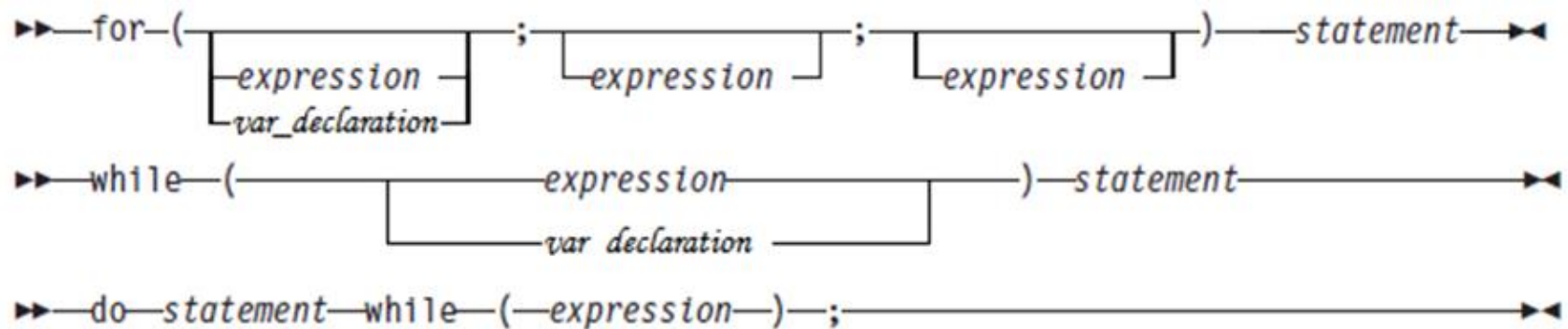
3.1.6 asm 和 static_assert语句

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。



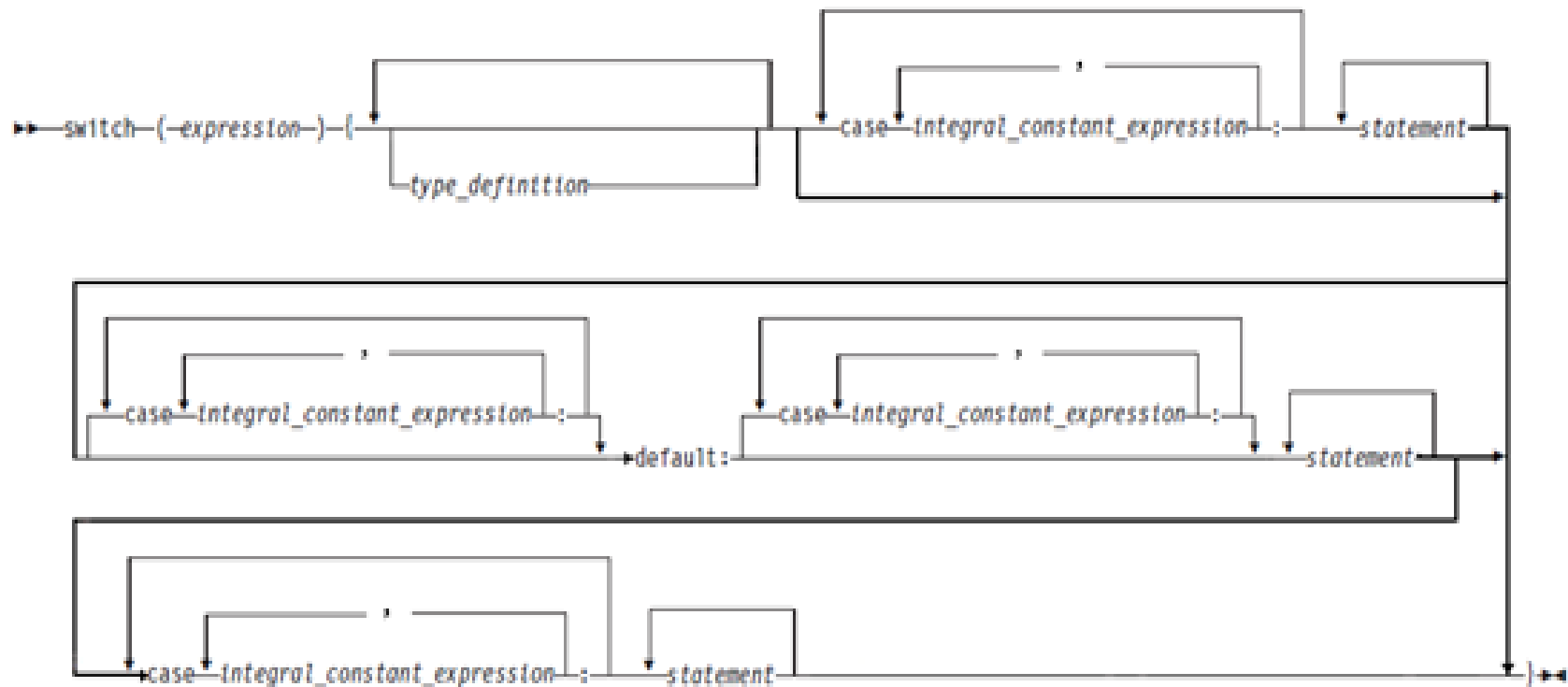
3.1 C++的语句

循环语句的语法图



3.1 C++的语句

switch语句的语法图





switch 小知识

```
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?





switch 小知识

important

- `switch(expression)` 中的 `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, 但放到中间应有 `break`;
- 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。



if 语句的建议

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？

短（即语句少的）分支在前

Q: if (条件1 && 条件2)
两个条件，谁在前谁在后，有何讲究？

不容易满足的条件在前

Q: if (条件) {.....}
else if () {.....}
else {} 可读性如何？

避免多层嵌套





if 与 switch 的转换

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?



if 与 switch 的转换

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 :  
}
```



有分支向无分支的转化

Example: 统计一个字符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']++;
```





函数指针

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 };
int z = p[i](20, 30);    // i=0, 执行fadd
```

函数地址表





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

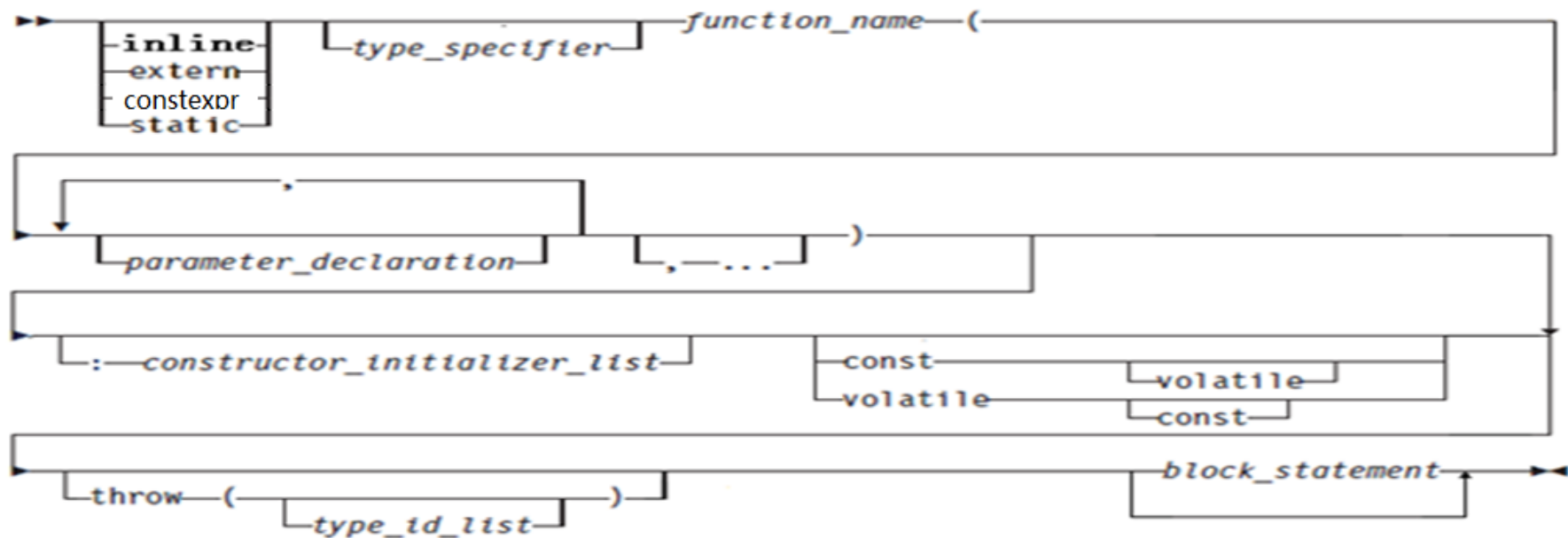
3.2.4 函数重载

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

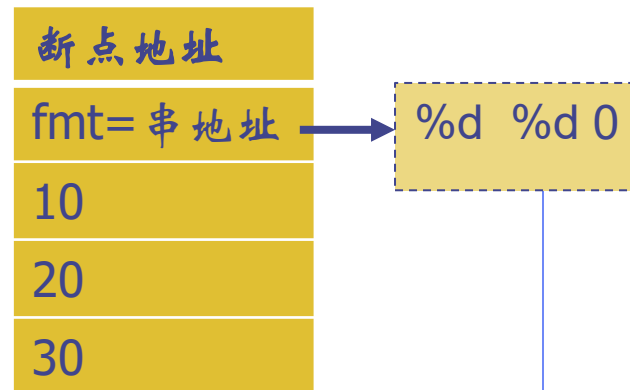
省略参数

```
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;    char *ap;
```

```
va_start(ap, n);          ap=(char *)&n + sizeof(int);
```

```
va_arg(ap, int);          s+=*(int *)ap;  ap +=4;
```

```
va_end(ap);              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)
{
    .....
}
```

缺省参数

```
DrawCircle(100, 100, 20);
```

```
DrawCircle(100, 100);
```

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)
{
    .....
}
```

缺省参数

```
DrawCircle(100, 100);
```

```
push    0Ah
```

```
push    64h
```

```
push    64h
```

```
call    DrawCircle
```

```
add     esp,0Ch
```

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



后置++

```
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

后置++ 的实现方式

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

参数为有址引用：传统左值有址引用示例

```
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

参数为有址引用：传统左值有址引用示例

```
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

函数定义参数为 传统左值有址引用，
但函数调用实参为 传统右值

```
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

无址引用

```
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++);`

无址引用

```
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，实现右值有址引用

```
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);
```

```
fcr(x++, y++);
```

```
fcr(5, 6);
```

// 三条语句均正确





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

```
inline double area(double r)
{
    return 3.14 * r * r;
}
```

```
double a;

a = area(5.0);
```





3.2.5 inline 及constexpr函数

inline

- 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。





3.2.5 inline 及constexpr函数

```
constexpr int increase(int x)
```

```
{
```

```
    return x+1;
```

```
}
```

```
constexpr int y=increase(1); // 编译生成的语句为 y=2;
```





3.2.5 inline 及constexpr函数

- 用constexpr定义的函数当其实参为常量时，可以被更彻底的优化；
- constexpr函数内不能有goto语句或标号，也不能有try语句块；
- constexpr函数不能调用非constexpr的函数，如printf函数；
- constexpr函数不能定义static变量、线程本地变量等永久期限变量；
- 类似inline函数，constexpr函数的函数体可能被优化掉，其作用域相当于static；
- 函数main为全局作用域，故不能定义为constexpr函数。





3.2.5 inline 及 constexpr 函数

const 与 constexpr 的比较

- 两者都可以在变量定义之前，定义只读变量，无差别；
`const int x=10; constexpr int y=20;`
- `const` 可以用于函数参数的说明，`constexpr` 不行；
- `const` 和 `constexpr` 都可以用在函数返回类型之前，但两种写法的意义不同。

`const char * f(...);` // 不能 `*f(...)=‘c’;`

`constexpr char * f(...);` // 可以 `*f(...)=‘c’;`

`const` 是存储可变特性

`constexpr` 是用于编译器对函数优化。





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

Microsoft Visual Studio 调试控制台

```
2 1 3 4 5 6 7 8 9 10 11 12 14 13 15 16 17 18 20 19
33 34 35 36 37 38 39 40 41 42 44 43 45 46 48 47 49 50
63 64 66 65 67 68 69 70 71 72 73 74 75 76 78 77 79 80
93 94 95 96 98 97 99 100 101 102 104 103 105 106 108 107
119 120 122 121 123 124 126 125 127 128 129 130 132 131
2 143 144 145 146 147 148 149 150 152 151 153 154 155 156
6 167 168 170 169 172 171 174 173 175 176 177 178 179 180
0 192 191 193 194 195 196 198 197 199 200 201 202 203 204
4 215 216 217 218 219 220 221 222 223 224 225 226 227 228
8 239 240 241 242 243 244 245 246 247 248 249 250 251 252
1 263 264 265 266 268 267 269 270 271 272 273 274 276 275
5 287 288 289 290 291 292 293 294 295 296 297 298 299 300
0 311 312 313 314 315 316 317 318 319 320 322 321 324 323
4 336 335 337 338 339 340 341 342 343 344 345 346 348 347
8 359 360 361 362 363 364 366 365 367 368 369 370 372 371
2 383 384 385 386 388 387 389 390 391 392 393 394 395 396
over counter = 400
```

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

```
Microsoft Visual Studio 调试控制台
1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
93 94 95 96 97 98 99 100 101 102 103 104 105 106 107
8  119 120 121 122 123 124 125 126 127 128 129 130 131
2  143 144 145 146 147 148 149 150 151 152 153 154 155
6  167 168 169 170 171 172 173 174 175 176 177 178 179
0  191 192 193 194 195 196 197 198 199 200 201 202 203
4  215 216 217 218 219 220 221 222 223 224 225 226 227
8  239 240 241 242 243 244 245 246 247 248 249 250 251
2  263 264 265 266 267 268 269 270 271 272 273 274 275
6  287 288 289 290 291 292 293 294 295 296 297 298 299
0  311 312 313 314 315 316 317 318 319 320 321 322 323
4  335 336 337 338 339 340 341 342 343 344 345 346 347
8  359 360 361 362 363 364 365 366 367 368 369 370 371
2  383 384 385 386 387 388 389 390 391 392 393 394 395
over counter = 400
```

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

Microsoft Visual Studio 调试控制台

```
1 1 2 3 4 5 6 7 8 9 11 10
1 31 32 33 34 35 36 37 39 38
0 61 62 63 64 65 66 67 68 69
0 91 92 93 94 95 96 97 98 99
117116 118 119 120 121 122 123
40 141 143 142 145144 146 147
64 165 167 166 168 169 170 170
87 188 189 190 191 192 193 194
10 211 212 213 214 215 216 217
34 235 237 236 238 238 239 240
56 257 259 258 261 260 262 263
80 281 283 282 284 285 286 287
04 305 306 307 308309 310 311
28 327 329 330 331 331 332 333
351 350 352 353 354 355 356 357
374373 375 375 376 377 378 378
over counter = 388
```

Microsoft Visual Studio 调试控制台

```
63 64 65 66 67 68 69 70 71 7
93 94 96 95 98 97 99 100 101
6 119 118 121120 122 123 124
142 143 144 145146 147 148
4 165 167 166 168 169 170 170
7 188 189 190 192 191 193 193
0 210 212 211 213 214 215 216
33 232 234 235 236 237 239 238
5 256 257 258 260 259 261 262
9 280 281281 282 283 284285
2 303 304 305 306 307 308 309
6 327 329 328 330 331 332 333
0 351 352 353 354 355 356 357
4 375 377 376 378 379 380 381
over counter = 392
```

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

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。

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。





3.3 作用域

- 同一个作用域内，不能定义同名变量
- 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。
- 作用域限制符 `::`
例如，全局变量 `int g=10;` `::g` 就表示全局变量
- 如果变量和常量是对象，则采用面向对象的作用域。





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）。

例如：不应返回一个函数局部变量的地址给调用函数；调用函数根据该地址进行操作存在不确定性。





华中科技大学

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;        // 元素指针
    int  v;        // 栈的大小
    int  t;        // 栈中已放的元素个数
};

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?

函数等价写法: `STK * const push = stk`;

函数调用时, 参数等价写法: `STK * const stk = &s`;

`*push=s1`; $\langle=\rangle$ `*stk = s1`; $\langle=\rangle$ `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`.

类比

函数等价写法: `STK * const push = stk;`

函数参数等价写法: `STK * const stk = &s;`

`push=&s1;` // = 的左操作数必须为左值





华中科技大学

