21307077 凌国明 第六次实验

作业6-1

```cpp
int main(){
    int i = 2;
    cout << (++i) * (i++);
    return 0;
}
```

```cpp
int main(){
    int i = 2;
    cout << (++i + 1) * (i++);
    return 0;
}
```

1. 使用C++,Java实现上述两段代码逻辑，并分析结果。
2. 将“int i =2;”变为“volatile int i=2;”之后，再次运行分析结果
3. 使用objdump打印1,2中的汇编代码，程序输出结果是9还是12,分析原因

1. 分别使用 CPP 和 Java 运行上述两段代码，分析结果

用 CPP 运行上述两段代码，输出结果都是 12

```
linggm@linggm-virtual-machine:~/homework/hw6$ g++ -o 11 11.cpp
linggm@linggm-virtual-machine:~/homework/hw6$ g++ -o 12 12.cpp
linggm@linggm-virtual-machine:~/homework/hw6$ ./11
12linggm@linggm-virtual-machine:~/homework/hw6$
linggm@linggm-virtual-machine:~/homework/hw6$ ./12
12linggm@linggm-virtual-machine:~/homework/hw6$
```

猜测这是因为寄存器和内存地址的值不同步导致的

代码一：(++i) * (i++)

++i 先执行，i 所在的寄存器的值变为 3，所在地址的内容也变为 3

i++ 后执行，因为括号的存在，i 所在地址的内容也变为 4

最终计算 3（寄存器旧值） * 4，得到 12。

代码二：(++i + 1) * (i++)

++i 将 i 从 2 增加到 3，并返回新值 3。

++i + 1 计算为 3 + 1，等于 4。

i++ 使用 i 的当前值 3 进行操作，并在操作完成后将 i 增加到 4。

因此，计算的结果是 4 * 3，等于 12。

以上只是一种猜测，并不完全正确，甚至有自相矛盾的地方，还是要看汇编

这种猜测是基于 C++中前缀和后缀加一和语义和顺序的不同

前缀加一（++i）：这个操作首先增加变量 i 的值，然后返回增加后的值。因此，如果 i 初始为 2，++i 会将 i 变为 3，同时表达式的值也是 3。

后缀加一（i++）：这个操作首先返回变量 i 当前的值，然后才对 i 进行增加。所以，如果 i 初始为 2，i++首先返回 2，然后将 i 增加到 3。

在复合表达式中，比如（++i）*（i++），这些操作的顺序和时间点就会变得复杂。由于 C++标准对某些情况下这些操作的精确顺序并没有明确规定（特别是当它们影响同一个变量时），不同的编译器可能会有不同的行为，导致未定义的结果。

用 Java 运行上述两段代码，左边代码输出 9，右边代码输出 12

```
linggm@linggm-virtual-machine:~/homework/hw6$ javac main11.java
linggm@linggm-virtual-machine:~/homework/hw6$ javac main12.java
linggm@linggm-virtual-machine:~/homework/hw6$ java main11
9
linggm@linggm-virtual-machine:~/homework/hw6$ java main12
12
```

分析：java 的运行结果相比 cpp 的要自然一些理想一些，毕竟 cpp 用两份不同的表达式跑出了相同的结果，而 java 的两份结果不同，具体分析见 objdump。

2. 将 int 加上 volatile 修饰符，变为 volatile int 后，再次运行

用 CPP 运行上述两段代码，左边代码输出 9，右边代码输出 12

```
linggm@linggm-virtual-machine:~/homework/hw6$ g++ -o 21 21.cpp
linggm@linggm-virtual-machine:~/homework/hw6$ g++ -o 22 22.cpp
linggm@linggm-virtual-machine:~/homework/hw6$ ./21
9linggm@linggm-virtual-machine:~/homework/hw6$
linggm@linggm-virtual-machine:~/homework/hw6$ ./22
12linggm@linggm-virtual-machine:~/homework/hw6$
```

分析：在 C++中，volatile 关键字用于通常用于操作硬件或在多线程编程中，它告诉编译器，变量的值可能会以编译器无法预知的方式改变。当一个变量被声明为 volatile 时，编译器会避免对这个变量的读写操作进行优化。加上 volatile 的运行结果看起来符合预期

用 Java 运行上述两段代码，编写代码如下

```
public class main21_0{
        public static void main(String args[]){
                volatile int i = 2;
                System.out.println((++i) * (i++));
        }
}
```

javac 遇到错误

```
linggm@linggm-virtual-machine:~/homework/hw6$ javac main21_0.java
main21_0.java:3: 错误：非法的表达式开始
                volatile int i = 2;
                ^
main21_0.java:4: 错误：需要<标识符>
                System.out.println((++i) * (i++));
                                   ^
main21_0.java:4: 错误：非法的类型开始
                System.out.println((++i) * (i++));
                                   ^
main21_0.java:6: 错误：需要class, interface或enum
}
^
4 个错误
```

volatile 关键字不能用于局部变量。volatile 关键字用于声明实例变量或类变量（静态变量），以确保多个线程之间对该变量的访问是可见的和有序的。

```
public class main21{
        private static volatile int i = 2;
        public static void main(String args[]){
                System.out.println((++i) * (i++));
        }
}
```

修改后，左边代码输出 9，右边代码输出 12

```
linggm@linggm-virtual-machine:~/homework/hw6$ javac main21.java
linggm@linggm-virtual-machine:~/homework/hw6$ javac main22.java
linggm@linggm-virtual-machine:~/homework/hw6$ java main21
9
linggm@linggm-virtual-machine:~/homework/hw6$ java main22
12
```

3. 用 objdump 打印汇编代码进行分析

1) 使用 objdump -d -M mips 11 > 11_objdump.txt 打印汇编代码

使用 mips 风格的汇编代码（计组学的 mips，更好分析），输出到 txt

```
00000000000007ca <main>:
7ca:    55                      push   %rbp                ; 将基指针寄存器 (rbp) 的值压栈
7cb:    48 89 e5                mov    %rsp,%rbp           ; 将栈指针寄存器 (rsp) 的值移动到基指针寄存器 (rbp)
7ce:    48 83 ec 10             sub    $0x10,%rsp          ; 从栈指针寄存器 (rsp) 减去16，为局部变量分配空间
7d2:    c7 45 fc 02 00 00 00    movl   $0x2,-0x4(%rbp)     ; 将整数2赋值给变量i (位于rbp-4的位置)
7d9:    83 45 fc 01             addl   $0x1,-0x4(%rbp)     ; 对变量i进行前置递增 (++i) , i变成3
7dd:    8b 45 fc                mov    -0x4(%rbp),%eax     ; 将变量i的值 (现在是3) 移动到eax寄存器
7e0:    8d 50 01                lea    0x1(%rax),%edx      ; 将eax寄存器的值 (3) 加1，并将结果 (4) 存储到edx寄存
器，(i++)
7e3:    89 55 fc                mov    %edx,-0x4(%rbp)     ; 将edx寄存器的值 (4) 回写到变量i，这是因为i++有括号
7e6:    0f af 45 fc             imul   -0x4(%rbp),%eax     ; 将变量i (现在是4) 与eax (之前保存的3) 进行相乘，结果
保存到eax
7ea:    89 c6                   mov    %eax,%esi           ; 将乘法结果 (现在是12) 移动到esi寄存器，准备输出
7ec:    48 8d 3d 2d 08 20 00    lea    0x20082d(%rip),%rdi ; 将cout对象的地址加载到rdi寄存器
7f3:    e8 a8 fe ff ff          callq  6a0 <_ZNSolsEi@plt> ; 调用输出函数，输出esi (即12)
7f8:    b8 00 00 00 00          mov    $0x0,%eax           ; 将0移动到eax寄存器，作为函数返回值
7fd:    c9                      leaveq                     ; 恢复栈帧，等同于mov %rbp，%rsp；pop %rbp
7fe:    c3                      retq                       ; 从函数返回
```

```
7ca: [x, x, x, x, x]  ; push %rbp                ; 将基指针寄存器 (rbp) 的值压栈
7cb: [x, x, x, x, x]  ; mov %rsp,%rbp            ; 将栈指针寄存器 (rsp) 的值移动到基指针寄存器 (rbp)
7ce: [x, x, x, x, x]  ; sub $0x10,%rsp           ; 从栈指针寄存器 (rsp) 减去16，为局部变量分配空间
7d2: [2, x, x, x, x]  ; movl $0x2,-0x4(%rbp)     ; 将整数2赋值给变量i (位于rbp-4的位置)
7d9: [3, x, x, x, x]  ; addl $0x1,-0x4(%rbp)     ; 对变量i进行前置递增 (++i) , i变成3
7dd: [3, 3, x, x, x]  ; mov -0x4(%rbp),%eax      ; 将变量i的值 (现在是3) 移动到eax寄存器
7e0: [3, 3, x, x, 4]  ; lea 0x1(%rax),%edx       ; 将eax寄存器的值 (3) 加1，并将结果 (4) 存储到edx寄存器，(i++)
7e3: [4, 3, x, x, 4]  ; mov %edx,-0x4(%rbp)      ; 将edx寄存器的值 (4) 回写到变量i，这是因为i++有括号
7e6: [4, 12, x, x, 4] ; imul -0x4(%rbp),%eax     ; 将变量i (现在是4) 与eax (之前保存的3) 进行相乘，结果保存到eax
7ea: [4, 12, x, x, 4] ; mov %eax,%esi            ; 将乘法结果 (现在是12) 移动到esi寄存器，准备输出
7ec: [4, 12, x, x, 4] ; lea 0x20082d(%rip),%rdi  ; 将cout对象的地址加载到rdi寄存器
7f3: [4, 12, x, x, 4] ; callq  6a0 <_ZNSolsEi@plt> ; 调用输出函数，输出esi (即12)
7f8: [4, 0, x, x, 4]  ; mov $0x0,%eax            ; 将0移动到eax寄存器，作为函数返回值
7fd: [x, 0, x, x, x]  ; leaveq                   ; 恢复栈帧，等同于mov %rbp，%rsp；pop %rbp
7fe: [x, 0, x, x, x]  ; retq                     ; 从函数返回
```

五元组的值分别代表：[-0x4(%rbp)，%eax，%ebx，%ecx，%edx]

以上是 cpp 运行代码 1.1 的汇编代码，我给 txt 写上了相应的注释，概括来说就是因为 imul 的第一个操作数是 i 的内存地址的内容(4)，第二个操作数却用了 i 在寄存器的旧值(++i 的结果 3)，所以最后的结果为 12。

代码 1.1：(++i) * (i++)

++i：i 的寄存器值为 3，i 的-0x4(%rbp)地址的值为 3

i++：计算为 4，因为括号，写回 i 的-0x4(%rbp)地址的值为 4

*：i 的寄存器旧值和 i 的-0x4(%rbp)地址的新值相乘，为 12

这里的-0x4(%rbp)指的是局部变量 i 的地址，其他存 i 的寄存器都可以说是 i 的 copy，由于 copy 和本体不一致，导致了结果不符合预期。

2) 使用 objdump -d -M mips 12 > 12_objdump.txt 打印汇编代码

使用 mips 风格的汇编代码，输出到 txt

```
00000000000007ca <main>:
7ca:    55                      push   %rbp              ; 将基指针寄存器（rbp）的值压栈
7cb:    48 89 e5                mov    %rsp,%rbp         ; 将栈指针寄存器（rsp）的值移动到基指针寄存器（rbp）
7ce:    48 83 ec 10             sub    $0x10,%rsp        ; 为局部变量分配栈空间，减少16个字节
7d2:    c7 45 fc 02 00 00 00    movl   $0x2,-0x4(%rbp)   ; 将整数2赋值给局部变量i（位于rbp-4的位置）
7d9:    83 45 fc 01             addl   $0x1,-0x4(%rbp)   ; 对变量i进行前置递增（++i），此时i变为3
7dd:    8b 45 fc                mov    -0x4(%rbp),%eax   ; 将变量i的值（现在是3）移动到eax寄存器
7e0:    8d 48 01                lea    0x1(%rax),%ecx    ; 将eax（即i的值，3）加1，结果（4）存入ecx寄存器
7e3:    8b 45 fc                mov    -0x4(%rbp),%eax   ; 再次将变量i的值（仍然是3）移动到eax寄存器
7e6:    8d 50 01                lea    0x1(%rax),%edx    ; 将eax（即i的值，3）加1，结果（4）存入edx寄存器，用于i++
7e9:    89 55 fc                mov    %edx,-0x4(%rbp)   ; 将edx寄存器的值（4）回写到变量i
7ec:    0f af c1                imul   %ecx,%eax         ; 将ecx（即++i + 1的结果，4）与eax（i的原始值，3）相乘，结果存入eax
7ef:    89 c6                   mov    %eax,%esi         ; 将乘法结果（现在是12）移动到esi寄存器，准备输出
7f1:    48 8d 3d 28 08 20 00    lea    0x200828(%rip),%rdi; 将cout对象的地址加载到rdi寄存器
7f8:    e8 a3 fe ff ff          callq  6a0 <_ZNSolsEi@plt>; 调用输出函数，输出esi（即12）
7fd:    b8 00 00 00 00          mov    $0x0,%eax         ; 将0移动到eax寄存器，作为函数返回值
802:    c9                      leaveq                   ; 恢复栈帧，等同于mov %rbp, %rsp; pop %rbp
803:    c3                      retq                     ; 从函数返回
```

```
7ca: [x, x, x, x, x]  ; push %rbp               ; 将基指针寄存器（rbp）的值压栈
7cb: [x, x, x, x, x]  ; mov %rsp,%rbp          ; 将栈指针寄存器（rsp）的值移动到基指针寄存器（rbp）
7ce: [x, x, x, x, x]  ; sub $0x10,%rsp         ; 为局部变量分配栈空间，减少16个字节
7d2: [2, x, x, x, x]  ; movl $0x2,-0x4(%rbp)    ; 将整数2赋值给局部变量i（位于rbp-4的位置）
7d9: [3, x, x, x, x]  ; addl $0x1,-0x4(%rbp)    ; 对变量i进行前置递增（++i），此时i变为3
7dd: [3, 3, x, x, x]  ; mov -0x4(%rbp),%eax    ; 将变量i的值（现在是3）移动到eax寄存器
7e0: [3, 3, x, 4, x]  ; lea 0x1(%rax),%ecx     ; 将eax（即i的值，3）加1，结果（4）存入ecx寄存器
7e3: [3, 3, x, 4, x]  ; mov -0x4(%rbp),%eax    ; 再次将变量i的值（仍然是3）移动到eax寄存器
7e6: [3, 3, x, 4, 4]  ; lea 0x1(%rax),%edx     ; 将eax（即i的值，3）加1，结果（4）存入edx寄存器，用于i++
7e9: [4, 3, x, 4, 4]  ; mov %edx,-0x4(%rbp)    ; 将edx寄存器的值（4）回写到变量i
7ec: [4, 12, x, 4, 4] ; imul %ecx,%eax          ; 将ecx（即++i + 1的结果，4）与eax（i的原始值，3）相乘，结果存入eax
7ef: [4, 12, x, 4, 4] ; mov %eax,%esi          ; 将乘法结果（现在是12）移动到esi寄存器，准备输出
7f1: [4, 12, x, 4, 4] ; lea 0x200828(%rip),%rdi; 将cout对象的地址加载到rdi寄存器
7f8: [4, 12, x, 4, 4] ; callq  6a0 <_ZNSolsEi@plt>; 调用输出函数，输出esi（即12）
7fd: [4, 0, x, 4, 4]  ; mov $0x0,%eax          ; 将0移动到eax寄存器，作为函数返回值
802: [x, 0, x, x, x]  ; leaveq                   ; 恢复栈帧，等同于mov %rbp, %rsp; pop %rbp
803: [x, 0, x, x, x]  ; retq                     ; 从函数返回
```

以上是 cpp 运行代码 1.2 的汇编代码，概括来说就是因为 imul 的第一个操作数是(++i+1=4)，第二个操作数却用了 i++前的寄存器的旧值 3，导致结果为 12

代码 1.2：(++i + 1) * (i++)

++i 将 i 从 2 增加到 3，将 i 的-0x4(%rbp)地址的值改为 3

++i + 1 计算为 3 + 1，等于 4，放入寄存器。

i++ 将 i 的-0x4(%rbp)地址的值 3 加载到寄存器，+1 后写回-0x4(%rbp)

因此，计算的结果是 4（寄存器 ecx） * 3（eax），等于 12。

这里的-0x4(%rbp)指的是局部变量 i 的地址，其他存 i 的寄存器都可以说是 i 的 copy，由于 copy 和本体不一致，导致了结果不符合预期。

3) 使用 objdump -d -M mips 21 > 21_objdump.txt 打印汇编代码

使用 mips 风格的汇编代码，输出到 txt

```
00000000000007ca <main>:
 7ca:   55                      push   %rbp              ; 将基指针寄存器（rbp）的值压栈
 7cb:   48 89 e5                mov    %rsp,%rbp         ; 将栈指针寄存器（rsp）的值移动到基指针寄存器（rbp）
 7ce:   48 83 ec 10             sub    $0x10,%rsp        ; 为局部变量分配栈空间，减少16个字节
 7d2:   c7 45 fc 02 00 00 00    movl   $0x2,-0x4(%rbp)   ; 将整数2赋值给局部变量i（位于rbp-4的位置）
 7d9:   8b 45 fc                mov    -0x4(%rbp),%eax   ; 将变量i的值（现在是2）移动到eax寄存器
 7dc:   8d 50 01                lea    0x1(%rax),%edx    ; 将eax的值加1，结果（3）存入edx寄存器，准备执行++i
 7df:   89 55 fc                mov    %edx,-0x4(%rbp)   ; 将edx寄存器的值（3）回写到变量i
 7e2:   8b 45 fc                mov    -0x4(%rbp),%eax   ; 再次将变量i的值（现在是3）移动到eax寄存器
 7e5:   8d 48 01                lea    0x1(%rax),%ecx    ; 将eax的值加1，结果（4）存入ecx寄存器，用于i++
 7e8:   89 4d fc                mov    %ecx,-0x4(%rbp)   ; 将ecx寄存器的值（4）回写到变量i
 7eb:   0f af c2                imul   %edx,%eax         ; 将edx（++i的结果，3）与eax（i++前的值，3）相乘，结果存入eax
 7ee:   89 c6                   mov    %eax,%esi         ; 将乘法结果（现在是9）移动到esi寄存器，准备输出
 7f0:   48 8d 3d 29 08 20 00    lea    0x200829(%rip),%rdi        # 201020 <_ZSt4cout@@GLIBCXX_3.4>
 7f7:   e8 a4 fe ff ff          callq  6a0 <_ZNSolsEi@plt>; 调用输出函数，输出esi（即9）
 7fc:   b8 00 00 00 00          mov    $0x0,%eax         ; 将0移动到eax寄存器，作为函数返回值
 801:   c9                      leaveq                   ; 恢复栈帧，等同于mov %rbp, %rsp; pop %rbp
 802:   c3                      retq                     ; 从函数返回
```

```
7ca: [x, x, x, x, x]  ; push %rbp               ; 将基指针寄存器（rbp）的值压栈
7cb: [x, x, x, x, x]  ; mov %rsp,%rbp           ; 将栈指针寄存器（rsp）的值移动到基指针寄存器（rbp）
7ce: [x, x, x, x, x]  ; sub $0x10,%rsp          ; 为局部变量分配栈空间，减少16个字节
7d2: [2, x, x, x, x]  ; movl $0x2,-0x4(%rbp)    ; 将整数2赋值给局部变量i（位于rbp-4的位置）
7d9: [2, 2, x, x, x]  ; mov -0x4(%rbp),%eax     ; 将变量i的值（现在是2）移动到eax寄存器
7dc: [2, 2, x, x, 3]  ; lea 0x1(%rax),%edx      ; 将eax的值加1，结果（3）存入edx寄存器，准备执行++i
7df: [3, 2, x, x, 3]  ; mov %edx,-0x4(%rbp)     ; 将edx寄存器的值（3）回写到变量i
7e2: [3, 3, x, x, 3]  ; mov -0x4(%rbp),%eax     ; 再次将变量i的值（现在是3）移动到eax寄存器
7e5: [3, 3, x, 4, 3]  ; lea 0x1(%rax),%ecx      ; 将eax的值加1，结果（4）存入ecx寄存器，用于i++
7e8: [4, 3, x, 4, 3]  ; mov %ecx,-0x4(%rbp)     ; 将ecx寄存器的值（4）回写到变量i
7eb: [4, 9, x, 4, 3]  ; imul %edx,%eax          ; 将edx（++i的结果，3）与eax（i++前的值，3）相乘，结果存入eax
7ee: [4, 9, x, 4, 3]  ; mov %eax,%esi           ; 将乘法结果（现在是9）移动到esi寄存器，准备输出
7f0: [4, 9, x, 4, 3]  ; lea 0x200829(%rip),%rdi; 将cout对象的地址加载到rdi寄存器
7f7: [4, 9, x, 4, 3]  ; callq  6a0 <_ZNSolsEi@plt>; 调用输出函数，输出esi（即9）
7fc: [4, 0, x, 4, 3]  ; mov $0x0,%eax           ; 将0移动到eax寄存器，作为函数返回值
801: [x, 0, x, x, x]  ; leaveq                  ; 恢复栈帧，等同于mov %rbp, %rsp; pop %rbp
802: [x, 0, x, x, x]  ; retq                    ; 从函数返回
```

以上是 cpp 运行代码 2.1 的汇编代码，概括来说就是因为 imul 的第一个操作数是(++i=3)，第二个操作数用了 i++前的寄存器的旧值 3，结果为 9

代码 2.1：(++i) * (i++)

++i 将 i 从 2 增加到 3，将 i 的-0x4(%rbp)地址的值改为 3

i++ 将 i 的-0x4(%rbp)地址的值 3 加载到寄存器，+1 后写回-0x4(%rbp)

因此，计算的结果是 3（寄存器 edx） * 3（eax），等于 9。

这里的-0x4(%rbp)指的是局部变量 i 的地址

4) 使用 objdump -d -M mips 22 > 22_objdump.txt 打印汇编代码

使用 mips 风格的汇编代码，输出到 txt

```
00000000000007ca <main>:
 7ca:   55                      push   %rbp              ; 将基指针寄存器（rbp）的值压栈
 7cb:   48 89 e5                mov    %rsp,%rbp         ; 将栈指针寄存器（rsp）的值移动到基指针寄存器（rbp）
 7ce:   48 83 ec 10             sub    $0x10,%rsp        ; 为局部变量分配栈空间，减少16个字节
 7d2:   c7 45 fc 02 00 00 00    movl   $0x2,-0x4(%rbp)   ; 将整数2赋值给局部变量i（位于rbp-4的位置）
 7d9:   8b 45 fc                mov    -0x4(%rbp),%eax   ; 将变量i的值（现在是2）移动到eax寄存器
 7dc:   83 c0 01                add    $0x1,%eax         ; 将eax的值加1，准备执行++i
 7df:   89 45 fc                mov    %eax,-0x4(%rbp)   ; 将加1后的eax的值（现在是3）回写到变量i
 7e2:   8d 48 01                lea    0x1(%rax),%ecx    ; 将eax的值加1，结果（4）存入ecx寄存器，用于计算++i + 1
 7e5:   8b 45 fc                mov    -0x4(%rbp),%eax   ; 再次将变量i的值（现在是3）移动到eax寄存器
 7e8:   8d 50 01                lea    0x1(%rax),%edx    ; 将eax的值加1，结果（4）存入edx寄存器，用于i++
 7eb:   89 55 fc                mov    %edx,-0x4(%rbp)   ; 将edx寄存器的值（4）回写到变量i
 7ee:   0f af c1                imul   %ecx,%eax         ; 将ecx（++i + 1的结果，4）与eax（i++前的值，3）相乘，结果存入eax
 7f1:   89 c6                   mov    %eax,%esi         ; 将乘法结果（现在是12）移动到esi寄存器，准备输出
 7f3:   48 8d 3d 26 08 20 00    lea    0x200826(%rip),%rdi; 将cout对象的地址加载到rdi寄存器
 7fa:   e8 a1 fe ff ff          callq  6a0 <_ZNSolsEi@plt>; 调用输出函数，输出esi（即12）
 7ff:   b8 00 00 00 00          mov    $0x0,%eax         ; 将0移动到eax寄存器，作为函数返回值
 804:   c9                      leaveq                   ; 恢复栈帧，等同于mov %rbp, %rsp; pop %rbp
 805:   c3                      retq                     ; 从函数返回
```

```
7ca: [x, x, x, x, x]  ; push %rbp               ; 将基指针寄存器（rbp）的值压栈
7cb: [x, x, x, x, x]  ; mov %rsp,%rbp          ; 将栈指针寄存器（rsp）的值移动到基指针寄存器（rbp）
7ce: [x, x, x, x, x]  ; sub $0x10,%rsp         ; 为局部变量分配栈空间，减少16个字节
7d2: [2, x, x, x, x]  ; movl $0x2,-0x4(%rbp)   ; 将整数2赋值给局部变量i（位于rbp-4的位置）
7d9: [2, 2, x, x, x]  ; mov -0x4(%rbp),%eax    ; 将变量i的值（现在是2）移动到eax寄存器
7dc: [2, 3, x, x, x]  ; add $0x1,%eax          ; 将eax的值加1，准备执行++i
7df: [3, 3, x, x, x]  ; mov %eax,-0x4(%rbp)    ; 将加1后的eax的值（现在是3）回写到变量i
7e2: [3, 3, x, 4, x]  ; lea 0x1(%rax),%ecx     ; 将eax的值加1，结果（4）存入ecx寄存器，用于计算++i + 1
7e5: [3, 3, x, 4, x]  ; mov -0x4(%rbp),%eax    ; 再次将变量i的值（现在是3）移动到eax寄存器
7e8: [3, 3, x, 4, 4]  ; lea 0x1(%rax),%edx     ; 将eax的值加1，结果（4）存入edx寄存器，用于i++
7eb: [4, 3, x, 4, 4]  ; mov %edx,-0x4(%rbp)    ; 将edx寄存器的值（4）回写到变量i
7ee: [4, 12, x, 4, 4] ; imul %ecx,%eax         ; 将ecx（++i + 1的结果，4）与eax（i++前的值，3）相乘，结果存入eax
7f1: [4, 12, x, 4, 4] ; mov %eax,%esi          ; 将乘法结果（现在是12）移动到esi寄存器，准备输出
7f3: [4, 12, x, 4, 4] ; lea 0x200826(%rip),%rdi; 将cout对象的地址加载到rdi寄存器
7fa: [4, 12, x, 4, 4] ; callq  6a0 <_ZNSolsEi@plt>; 调用输出函数，输出esi（即12）
7ff: [4, 0, x, 4, 4]  ; mov $0x0,%eax         ; 将0移动到eax寄存器，作为函数返回值
804: [x, 0, x, x, x]  ; leaveq                ; 恢复栈帧，等同于mov %rbp, %rsp; pop %rbp
805: [x, 0, x, x, x]  ; retq                  ; 从函数返回
```

以上是 cpp 运行代码 2.2 的汇编代码，概括来说就是因为 imul 的第一个操作数是(++i+1=4)，第二个操作数用了 i++前的寄存器的旧值 3，结果为 9

代码 2.2：(++i+1) * (i++)

++i 将 i 从 2 增加到 3，将 i 的-0x4(%rbp)地址的值改为 3

++i+1 将 i 的 ecx 处的值从 3 改为 4，不写回-0x4(%rbp)

i++ 将 i 的-0x4(%rbp)地址的值 3 加载到寄存器，将 4 写回-0x4(%rbp)

因此，计算的结果是 4（寄存器 ecx） * 3（eax），等于 12。

这里的-0x4(%rbp)指的是局部变量 i 的地址

作业6-2

- 1. 一个表达式判断一个uint32是不是2的整数次幂：
- 2.一个表达式判断一个uint32是不是3的整数次幂。

```
// 判断 a 是否为 b 的整数次幂
bool f(uint a, uint b, uint t) { return (a % b != 0 && a != 1) && (t == 0 || a >= b) ? false : (a < b) ? (a == 1) : (f(a/b, b, t+1) ); }

int main(){
    uint a, b;
    cin >> a >> b;
    if(f(a, b, 0))
        cout << a << " 是 " << b << " 的整数次幂 ";
    else
        cout << a << " 不是 " << b << " 的整数次幂 ";
    return 0;
}
```

bool f(uint a, uint b, uint t) { return (a % b != 0 && a != 1) && (t == 0 || a >= b) ? false : (a < b) ? (a == 1) : (f(a/b, b, t+1)); }

a 不是 b 的倍数时，必然不是 b 的整数次幂，展开为以下代码

```
bool f(uint a, uint b, uint t){
    if ((a % b != 0 && a != 1) && (t == 0 || a >= b)){
        return false;
    }
    else{
        if(a < b){
            return a == 1;
        }
        else{
            return f(a/b, b, t+1);
        }
    }
}
```

当 b==2 时，可以通过位运算来节省时间，但是 b==3 时貌似不行。

bool f2(uint a) { return n & (n-1) == 0; }