朱浩泽 1911530
计算机系统设计平时作业三
March 11, 2022

# 平时作业三

**Question 1.** 用代码的方式反汇编 i++ 和 ++i，了解和解释其背后的原理。

我们利用 linux 自带工具 objdump 对代码进行反汇编操作，其指令如下源代码如下：

```
g++ -c -o 1.o 1.cpp
objdump -s -d 1.o > 1.o.txt
```

```
int main()
{
    int i = i;
    i++;
}
```

Listing 1. i++ 源代码

```
int main()
{
    int i = 1;
    ++i;
}
```

Listing 2. ++i 源代码

进行反汇编后的代码如下：

```
1.o:     文件格式 elf64-x86-64

Contents of section .text:
 0000 f30f1efa 554889e5 c745fc01 00000083  ....UH...E......
 0010 45fc01b8 00000000 5dc3               E.......].
Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520392e  .GCC: (Ubuntu 9.
 0010 332e302d 31377562 756e7475 317e3230  3.0-17ubuntu1~20
 0020 2e303429 20392e33 2e3000             .04) 9.3.0.
Contents of section .note.gnu.property:
 0000 04000000 10000000 05000000 474e5500  ............GNU.
 0010 020000c0 04000000 03000000 00000000  ................
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 1a000000 00450e10 8602430d  .........E....C.
 0030 06510c07 08000000                    .Q......

Disassembly of section .text:

0000000000000000 <main>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
   8: c7 45 fc 01 00 00 00  movl   $0x1,-0x4(%rbp)
   f: 83 45 fc 01           addl   $0x1,-0x4(%rbp)
  13: b8 00 00 00 00        mov    $0x0,%eax
```

```
  18: 5d                    pop    %rbp
  19: c3                    retq
```

Listing 3. i++ 反汇编代码

```
2.o:     文件格式 elf64-x86-64

Contents of section .text:
 0000 f30f1efa 554889e5 c745fc01 00000083  ....UH...E......
 0010 45fc01b8 00000000 5dc3               E.......].
Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520392e  .GCC: (Ubuntu 9.
 0010 332e302d 31377562 756e7475 317e3230  3.0-17ubuntu1~20
 0020 2e303429 20392e33 2e3000             .04) 9.3.0.
Contents of section .note.gnu.property:
 0000 04000000 10000000 05000000 474e5500  ............GNU.
 0010 020000c0 04000000 03000000 00000000  ................
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 1a000000 00450e10 8602430d  .........E....C.
 0030 06510c07 08000000                    .Q......

Disassembly of section .text:

0000000000000000 <main>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
   8: c7 45 fc 01 00 00 00  movl   $0x1,-0x4(%rbp)
   f: 83 45 fc 01           addl   $0x1,-0x4(%rbp)
  13: b8 00 00 00 00        mov    $0x0,%eax
  18: 5d                    pop    %rbp
  19: c3                    retq
```

Listing 4. ++i 反汇编代码

可以看出，以上代码并没有任何区别，所以我们将单纯的 $i++$ 更改为 $int\ a = i++$ ($++i$ 更改为 $int\ a = ++i$) 再做尝试，可以得到反汇编代码如下：

```
0000000000000000 <main>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
   8: c7 45 f8 01 00 00 00  movl   $0x1,-0x8(%rbp)
   f: 8b 45 f8              mov    -0x8(%rbp),%eax
  12: 8d 50 01              lea    0x1(%rax),%edx
  15: 89 55 f8              mov    %edx,-0x8(%rbp)
  18: 89 45 fc              mov    %eax,-0x4(%rbp)
  1b: b8 00 00 00 00        mov    $0x0,%eax
  20: 5d                    pop    %rbp
  21: c3                    retq
```

Listing 5. i++ 反汇编代码

```
0000000000000000 <main>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
```

```
   8: c7 45 f8 01 00 00 00  movl   $0x1,-0x8(%rbp)
   f: 83 45 f8 01           addl   $0x1,-0x8(%rbp)
  13: 8b 45 f8              mov    -0x8(%rbp),%eax
  16: 89 45 fc              mov    %eax,-0x4(%rbp)
  19: b8 00 00 00 00        mov    $0x0,%eax
  1e: 5d                    pop    %rbp
  1f: c3                    retq
```

Listing 6. ++i 反汇编代码

从上述代码便可以看出这两者之间的区别，首先从宏观上来看，如果是 $int\ a = i++$ 的话，最终的值得到的是 1，即 a 获得的是 i 的原值；如果是 $int\ a = ++i$ 的话，最终得到的值是 2，即 a 获得的是 i 更新后的值。然后对反汇编生成的代码进行分析，可以看出首先在栈上为变量 i 开辟一个空间，对 i 进行赋值操作，然后便出现了区别，i++ 是先取出 i 到操作数栈、然后再对局部变量表中的 i 做 ++，而 ++i 是先对 i 进行加一操作，然后才取出 i 到操作数栈。对于上面代码具体来说便是 $int\ a = i++$ 的过程是先将 i 的原值 1 复制到操作数栈，对 a 进行赋值即弹出操作数栈顶的值 1，然后对局部变量表中 i 的原始值进行加 1 的操作，即使得 i 由 1 变为 2，然后将这个值再赋给 i 更新 i 的值；而 $int\ a = ++i$ 的过程是先对局部变量表中 i 的原始值进行加 1 的操作，即使得 i 由 1 变为 2，然后将 i 的值复制到操作数栈，最后赋值即弹出操作数栈顶的值。

**Question 2.** 用代码解释预处理的作用，背后的原因。

首先我们先找出一段基准代码如下

```cpp
#include <iostream>
#define N 100
using namespace std;
int main()
{
    int a, b, i, t;
    a = 0;
    b = N;
    i = 1;
    cout << a << endl;
    cout << b << endl;
    while(i < (15))
    {
        t = b;
        b = a +b ;
        cout << b << endl;
        a = t;
        i = i + 1;
    }
    return 0;
}
```

Listing 7. 源文件

我们利用 g++ helloworld.cpp −E −o helloworld.i 生成预处理后的文件如下

```cpp
…… 此处省略替换的头文件
using namespace std;
int main()
{
    int a, b, i, t;
```

```
    a = 0;
    b = 100;
    i = 1;
    cout << a << endl;
    cout << b << endl;
    while(i < (15))
    {
        t = b;
        b = a +b ;
        cout << b << endl;
        a = t;
        i = i + 1;
    }
    return 0;
}
```

Listing 8. 预处理后的文件 (节选)

首先观察 helloworld.i 文件长度，在 Linux 系统下生成的 helloworld.i 文件由原本的 22 行代码增长至 18189 行，而 macOS 系统下则增至 41496 行代码。然后我们将引入头文件的语句删除，再进行一次该步骤，可以发现代码长度没有显著变化。可以很直观的看出，代码长度暴涨的原因是因为源文件中的 #include <iostream> 即引入头文件的语句，被源文件的内容直接替代。由此可见，预处理器的一项工作便是引用头文件的语句直接替换为源文件的内容。而不同平台虽然都使用了 g++ 编译器，但预处理后的代码也存在着巨大的差异，这是因为不同平台下，由于机器码等存在着差异，导致标准库头文件存在着差异，预处理阶段可以很好的降低这一差异所带来的相同代码在不同平台上所产生的各种问题。接下来对去除头文件后剩下的部分进行分析，可以看出源文件中的注释被自动消去了，宏定义的语句也被取代；而在 main 函数中，宏定义的参数直接被宏体做了简单的替换，由此可见，在我们写程序时，应将宏体用括号封装，以免出现应预处理器仅进行宏的简单替换而导致程序出错的问题。预处理器还有条件编译的功能。条件编译是根据实际定义宏（某类条件）进行代码静态编译的手段，可根据表达式的值或某个特定宏是否被定义来确定编译条件。预处理阶段的条件编译可以有效的防止重复包含头文件的宏。