03｜计算单元：运算符是如何工作的？

2021-12-10 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航
时长 15:48 大小 14.47M

你好，我是于航。

运算符（operator）、表达式（expression）和语句（statement）是组成 C 程序的三个最基本的语法结构。在 C 语言中，这三种概念之间一般呈“包含”关系，即表达式中通常含有运算符，而语句中也可以包含有表达式。最终，众多的语句便组成了一个完整的 C 程序。

可见，一个完整的 C 程序是由不同粒度的语法单元自下而上一层层构建出来的。而在这层语法形式之下，运算符、表达式和语句究竟是怎样被编译，并通过机器指令表达的呢？

作为 C 语言中用于提供计算能力的核心语法结构，运算符在支持应用核心功能的构建过程中，起着不可或缺的作用。那么这一讲，我们就先来看看，C 语言中的运算符究竟是如何

被编译器实现的。

C 运算符的分类

在目前最新的 C17 标准中，C 语言一共有 48 个运算符。按照这些运算符功能的不同，我们可以将它们分为七类（分类方式并不唯一），如下表所示：

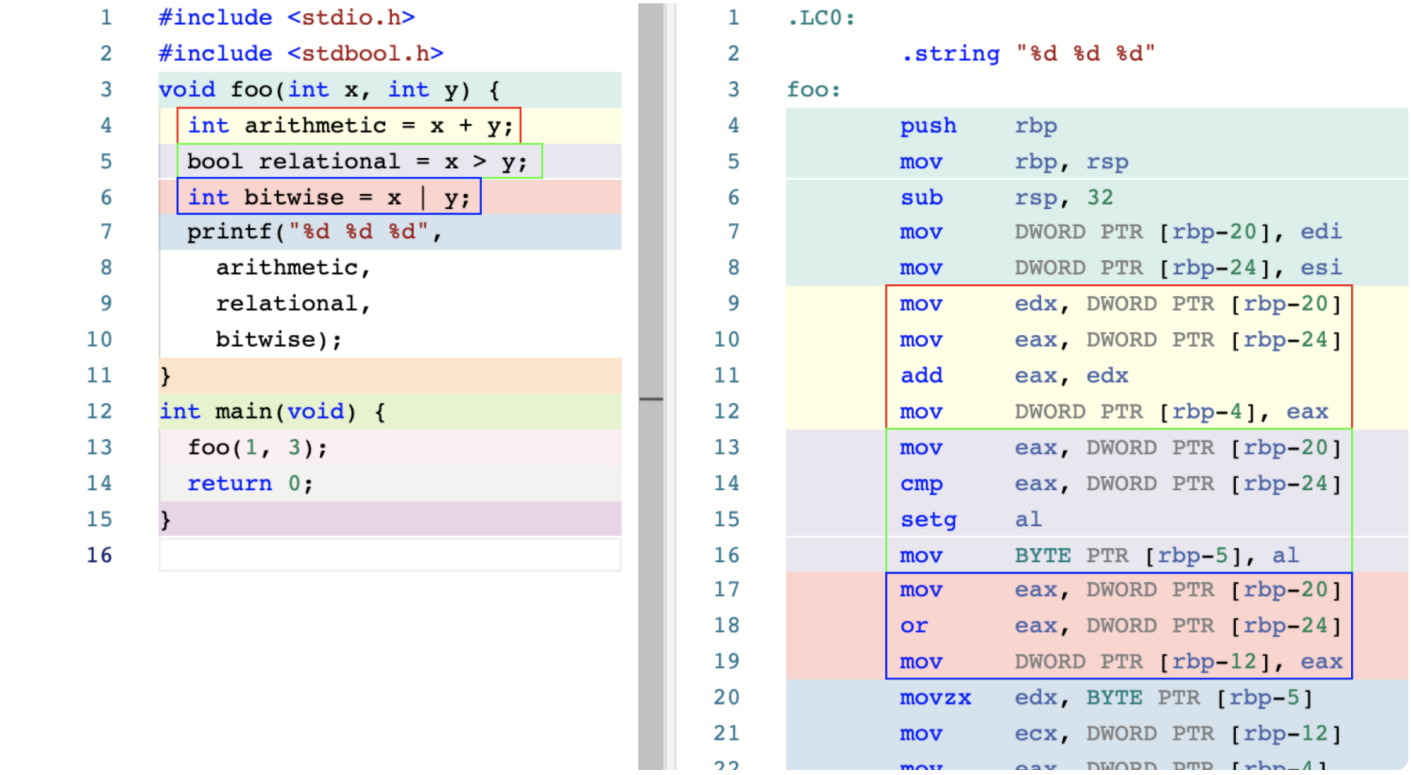
类型	包含的运算符（仅列出一部分）
算数运算符	+a、-a、a * b、a / b、a % b、++a、a--
关系运算符	a == b、a != b、a > b、a < b、a >= b、a <= b
位运算符	a & b、a | b、a ^ b、~a、a << b、a >> b
赋值运算符	a = b、a += b、a -= b、a *= b、a /= b
逻辑运算符	a && b、a || b、!a
成员访问运算符	a[b]、a.b、a->b、&a、*a
其他运算符	sizeof、?:、(type) a、a(...)、a, b

这七类运算符在功能上均有所不同，因此，使用机器指令进行表达的具体方式和复杂程度也不同。其中，算数、关系、位与赋值运算符由于功能较为基础，可以与某些机器指令一一对应，因此我们会放在一起进行介绍。而逻辑运算符、成员访问运算符及其他运算符，由于实现相对复杂，我会分开讲解。

算数、关系、位、赋值运算符

算数、关系、位、赋值这四类运算符在经过编译器处理后，一般都可以**直接对应到由目标平台上相应机器指令组成的简单计算逻辑**。

比如，在下面这段示例代码中，我们在 foo 函数的内部使用到了加法运算符 “+” 、大于运算符 “>” ，以及按位或运算符 “|” 。你可以通过右侧的输出内容，查看默认情况下（即不使用任何编译优化）左侧 C 源代码对应的汇编结果。



```c
#include <stdio.h>
#include <stdbool.h>
void foo(int x, int y) {
  int arithmetic = x + y;
  bool relational = x > y;
  int bitwise = x | y;
  printf("%d %d %d",
    arithmetic,
    relational,
    bitwise);
}
int main(void) {
  foo(1, 3);
  return 0;
}
```

```
.LC0:
        .string "%d %d %d"
foo:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-24], esi
        mov     edx, DWORD PTR [rbp-20]
        mov     eax, DWORD PTR [rbp-24]
        add     eax, edx
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-20]
        cmp     eax, DWORD PTR [rbp-24]
        setg    al
        mov     BYTE PTR [rbp-5], al
        mov     eax, DWORD PTR [rbp-20]
        or      eax, DWORD PTR [rbp-24]
        mov     DWORD PTR [rbp-12], eax
        movzx   edx, BYTE PTR [rbp-5]
        mov     ecx, DWORD PTR [rbp-12]
```

这里为了便于观察，我将代码中关键的三行 C 语句，以及它们对应的汇编代码用相同颜色的框标注了出来。**下面我们来分别看看它们的实现方式。**

先来看左侧代码中用红色框标注的内容，这是一个使用了加法运算符 “+” 的 C 语句。在它对应的汇编代码中，前两行代码分别从栈内存中将变量 x 与 y 的值放入到了寄存器 *edx* 与 *eax* 里。紧接着，程序通过 add 机器指令计算这两个寄存器中的数字之和。随后，通过 mov 指令，程序将计算得到的结果值从寄存器 *eax* 移动到了局部变量 arithmetic 对应的栈内存中。至此，这行 C 代码便执行结束了。

实际上，这行 C 语句同时包含了对算数运算符 “+” 和赋值运算符 “=” 的使用过程。其中，汇编指令 add 直接对应于 C 代码中加法运算符的操作。而 mov 指令则对应于等号赋值运算符的操作。

在这门课后面的内容中，你还会多次看到编译器对 mov 指令的使用。这里我们采用的是 Intel 的汇编代码格式，因此，当该指令的源或目的操作数中涉及到某个具体的内存位置时，汇编代码中会出现类似 “DWORD PTR [rbp-8]” 的参数形式。

对此，你可以这样解读：将寄存器 *rbp* 中的值减去 8 得到的结果作为一个地址，然后在这个地址上读取 / 写入大小为 DWORD 的值。在 Intel 体系中，一个 WORD 表示 16 位，一个 DWORD 为 32 位，而一个 QWORD 表示 64 位。

接下来，我们继续观察另外两条 C 语句对应的汇编代码，你可以得到类似的结论。其中，绿色框标注的关系运算符大于号 “>” 对应汇编指令 cmp。这个指令在被执行时，会首先比较变量 x 与 y 值的大小，并根据比较结果，动态调整 CPU 上 FLAGS 寄存器中的相应位。

为了帮你深入理解接下来的内容，我们先来熟悉一些相关概念。这里提到的 **FLAGS 寄存器是一组用于反映程序当前运行状态的标志寄存器**。许多机器指令在执行完毕时，都会同时调整 FLAGS 寄存器中对应位的值，以响应程序状态的变化。

比如，在上面这段代码中，汇编指令 cmp 的下一条汇编指令 setg 便会通过查看 FLAGS 寄存器中的 ZF 位是否为 0，且 SF 与 OF 位的值是否相等，来决定将寄存器 *al* 中的值置 1，还是置 0。而该寄存器中存放的数字值，便为变量 relational 的最终结果。

那么，ZF 位、SF 位、OF 位又是什么呢？它们其实都是 FLAGS 寄存器中的常用标志位，用来反映当前指令执行后引起的 CPU 状态变化。我把 FLAGS 寄存器中的常用标志位整理在了下面的表格中，你可以参考。

标志位名称	位	全称	什么情况下置位（即变更为值 1）
CF	0	Carry	指令执行引起了进位或借位
PF	2	Parity	指令执行结果的最低有效字节中值为 1 的位个数为偶数
ZF	6	Zero	指令执行结果为 0
SF	7	Sign	指令执行结果的最高有效位为 1
OF	11	Overflow	当操作数被当做有符号数时，指令的执行产生了溢出

极客时间

理解了 FLAGS 寄存器和标志位的概念后，再来看 cmp 指令的具体执行流程，你应该会更加清楚了。

举一个简单的例子：假设这里函数 add 在调用时传入的值 x 为 3，y 为 2。那么，当 cmp 指令执行时，它首先会在 CPU 内部对这两个操作数进行隐式的减法运算，运算后得到结果 1。而 ZF、SF、OF 在这里都将被复位，而复位则代表着标志位所表示的状态为假。因此，

FLAGS 寄存器的状态满足指令 `setg` 的置位条件（ZF=0 且 SF=OF），*al* 寄存器的值将被置 1。

到这里，我相信基本原理你已经理解了，不过还是建议你自己动手实践下。我给你留下了两个小问题，你可以在评论区和我讨论：

> 当改变值 x 与 y 的大小以及正负性时，`cmp` 指令的执行会对 FLAGS 寄存器有何影响？
>
> 此时， FLAGS 寄存器中标志位的值变化是否符合指令 `setg` 的置位条件？

针对这两个问题，你需要注意下我在 02 讲 中提过的这一点：**负值在寄存器中是以补码的形式存放的，而计算机在进行加减运算时，不需要区分操作数的符号性**。

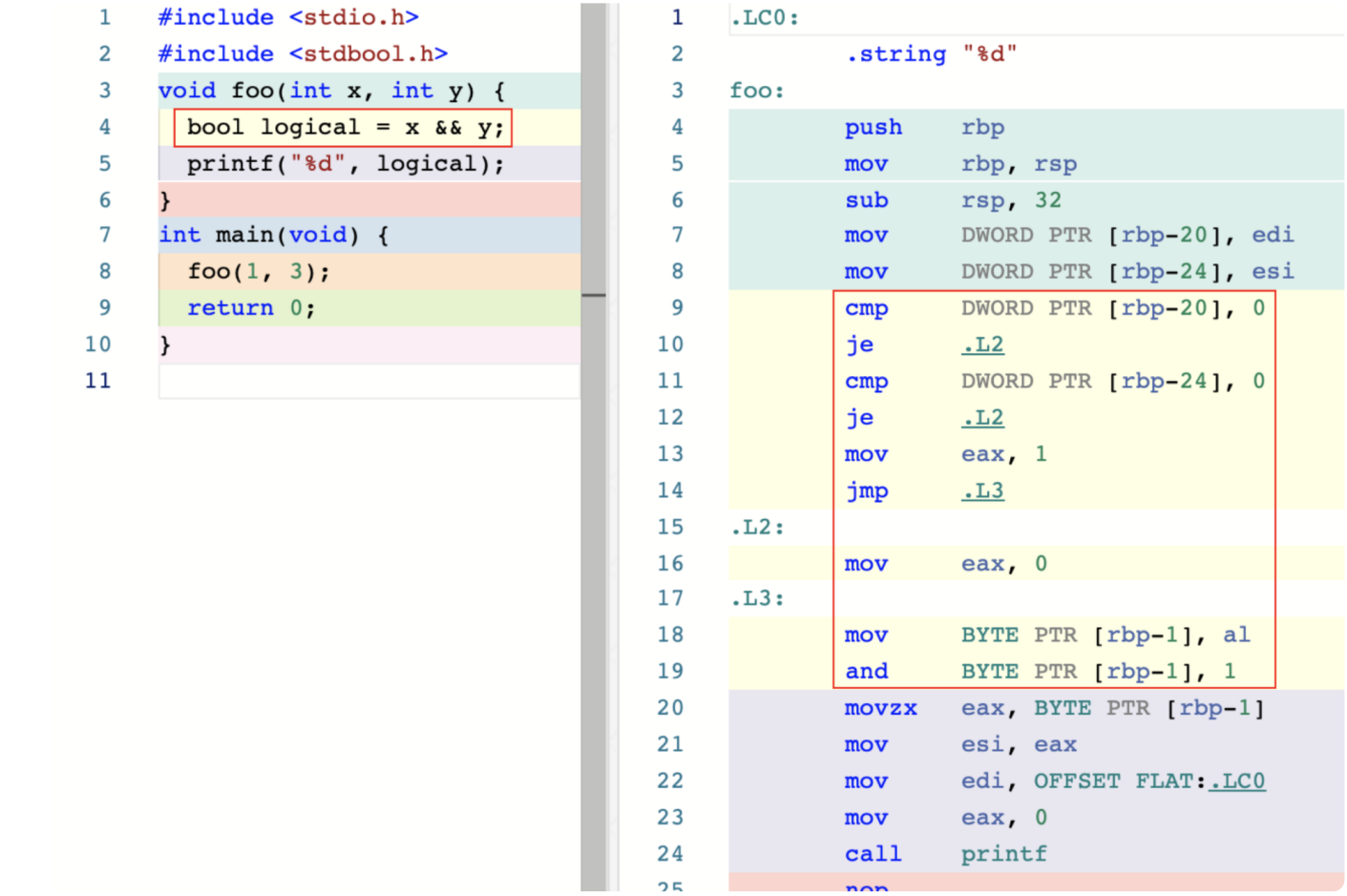
最后，再来看下这段代码中蓝色框标注的或运算符 “|” 。可以很直观地看到，它所对应的汇编指令是 `or`。

针对这几类运算符，值得一提的是，即便是在开启了最高编译优化等级的情况下，编译器实现上述这些运算符的基本逻辑仍然不变，只不过会相对减少通过栈内存访问函数传入参数的过程，而在某些情况下会选用寄存器传值。

逻辑运算符

说完这四种较为直观的运算符，让我们再来看看逻辑运算符。这里，我们直接以逻辑与运算符 “&&” 为例，来看看编译器是如何实现它的。该类别下的其他运算符，实现方式与其类似，区别仅在于使用的具体指令可能有所不同。

首先，我们还是来看一段示例代码：



```
#include <stdio.h>
#include <stdbool.h>
void foo(int x, int y) {
  bool logical = x && y;
  printf("%d", logical);
}
int main(void) {
  foo(1, 3);
  return 0;
}
```

```
.LC0:
        .string "%d"
foo:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-24], esi
        cmp     DWORD PTR [rbp-20], 0
        je      .L2
        cmp     DWORD PTR [rbp-24], 0
        je      .L2
        mov     eax, 1
        jmp     .L3
.L2:
        mov     eax, 0
.L3:
        mov     BYTE PTR [rbp-1], al
        and     BYTE PTR [rbp-1], 1
        movzx   eax, BYTE PTR [rbp-1]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        nop
```

在 C 标准中，逻辑与运算符 “&&” 的语义是：如果它左右两侧的操作数都具有非零值，则产生计算结果值 1。而如果任一操作数为 0，则计算结果为 0。

不仅如此，标准还规定了该运算符在执行模型中的求值规则：如果通过逻辑与运算符左侧第一个操作数的求值结果就能确定表达式的值，就不再需要对第二个操作数进行求值了，这也就是我们常说的“短路与”。那么在汇编中，这个运算符是如何实现的呢？

逻辑与运算符并没有可与之直接对应的汇编指令。并且，为了满足“短路”要求，编译器在非优化的实现中通常会使用条件跳转指令，比如 `je`。`je` 指令会判断当前 FLAGS 寄存器中的标志位 ZF 是否为 0。若为 0，则会将程序执行直接跳转到给定标签所在地址上的指令。

上图中右侧输出的汇编代码里，程序会按顺序将位于栈内存中的变量 x 和 y 的值与数值 0 进行比较。若其中的某个比较结果相等，程序执行将会直接跳转到标签 “.L2” 的所在位置。在这里，值 0 会被直接放入寄存器 *eax*。而若变量 x 与 y 的值判断均不成立，则值 1 会被放入该寄存器。紧接着，标签 “.L3” 中的指令将接着执行。

到这里，寄存器 *eax* 中的值将会被作为最终结果，赋值给变量 logical。这里我再给你留个小问题：标签 “.L3” 中前两条汇编语句的作用是什么？欢迎在评论区留下你的看法。

当然，就逻辑与运算符来说，在使用高编译优化等级时，编译器还可能会采用下面这种方式来实现该运算符。这里，我们看到了新的汇编指令： `test` 、`setne` 和 `movzx` 。

复制代码

```
1 test    edi, edi  ; edi <- x.
2 setne   al
3 test    esi, esi  ; esi <- y.
4 setne   sil
5 movzx   esi, sil
6 and     esi, eax
```

`test` 指令的执行方式与 `cmp` 类似，只不过它会对传入的两个操作数做隐式的“与”操作，而非减法操作。在操作完成后，根据计算结果，指令会相应地修改 FLAGS 寄存器上的 SF、ZF 以及 PF 标志位。另外的 `setne` 指令则与 `setg` 指令类似，该指令将在 ZF 为 0 时把传入的寄存器置位，否则将其复位。最后的指令 `movzx` 实际上是 `mov` 指令的一种变体。这个指令将数据从源位置移动到目标位置后，会同时对目标位置上的数据进行零扩展（Zero Extension）。

了解了这些，我们就可以来尝试理解**编译器在高优化等级下对逻辑与运算符的实现方式**：首先，通过 `test` 指令，程序可以判断参数 x 与 y 是否为非零值。若为非零值，则相应的寄存器会被指令 `setne` 置位。在这种情况下，寄存器 *al* 与 *sil* 中便存放有用于表示变量 x 与 y 是否为零的布尔数字值 0 或 1。接下来，通过数据移动指令，寄存器 *sil* 中的值被移动到寄存器 *esi* 中。最后的 `and` 指令又会对 x 与 y 的布尔数字值再次进行与操作，得到的最终结果将被存放在目的寄存器，即 *esi* 中。

这种实现方式大量减少了对栈内存以及条件跳转指令的使用，使得程序减少了访问内存时产生的延迟，以及由于分支预测失败而导致的 CPU 周期浪费，从而执行性能得到了提升。

可以看到的是，在使用高编译优化等级的情况下，C 标准中逻辑与操作符的“短路”特性并没有体现出来，程序实际上同时对参数 x 与 y 的值进行了判断。而这也正是因为 **C 语言的 “as-if” 性质给予了编译器更多的优化空间**。C 标准中规定，除去几种特殊的情况外，

在不影响一个程序的外部可观测行为的情况下，编译器可以不遵循 C 标准中对执行模型的规定，而是采用其特定的实现方式，优化程序的性能。

在非优化版本的实现中，编译器使用了 je 条件跳转指令。其实，现代流水线 CPU 通常会采用一种名为“**投机执行**”的方式来优化条件跳转指令的执行。所谓投机执行，是指 CPU 会通过分析历史的分支执行情况，来推测条件跳转指令将会执行的分支，并提前处理所预测分支上的指令。而等到 CPU 发现之前所预测的分支是错误的时候，它将不得不丢弃这个分支上指令的所有中间处理结果，并将执行流程转移到正确的分支上。很明显，这样就会浪费较多的时钟周期。

更多优化 C 程序性能的技巧，我会在“ C 工程实战篇”那个模块中详细讲解。相信在对 C 核心语法的实现细节有了更深入的理解后，你在学习这些知识时也会更容易、理解得更透彻。

成员访问运算符

接下来，让我们继续来看看成员访问运算符。这里我以取地址运算符 “&” 、解引用运算符 “*” 为例，来介绍编译器对它们的实现细节。来看下面这段代码：

```c
#include <stdio.h>
void foo(void) {
  int n = 10;
  int* n_ptr = &n;
  int m = *n_ptr;
  printf("%d %p %d", n, n_ptr, m);
}
int main(void) {
  foo();
  return 0;
}
```

```
.LC0:
        .string "%d %p %d"
foo:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-16], 10
        lea     rax, [rbp-16]
        mov     QWORD PTR [rbp-8], rax
        mov     rax, QWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rax]
        mov     DWORD PTR [rbp-12], eax
        mov     eax, DWORD PTR [rbp-16]
        mov     ecx, DWORD PTR [rbp-12]
        mov     rdx, QWORD PTR [rbp-8]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        nop
```

如上图中红色框对应的 C 代码和汇编代码所示，对于取地址运算符 “&” ，实际上它一般会直接对应到名为 lea 的汇编指令。这个指令的全称为 “Load Effective Address” ，顾名思义，它主要用来将操作数作为地址，并将这个地址以值的形式传送到其他位置。比

如，上面代码中的 lea 指令将寄存器 *rbp* 中的值减去 16 后，直接存放到寄存器 *rax* 中，而此时该寄存器中的值就是局部变量 n 在栈上的地址。

而解引用运算符 “*” 的行为与取地址运算符完全相反。当需要对位于某个地址上的值进行传送时，我们可以直接使用 mov 指令。上图中，在蓝色框的汇编代码里，第一条 mov 指令将变量 n_ptr 的值传送到了寄存器 *rax* 中。随后，第二条 mov 指令将 *rax* 寄存器中的值作为地址，并将该地址上的值以 DWORD，即 32 位值（对应 int 类型）的形式传送到 *eax* 寄存器中。最后，第三条 mov 指令将此时 *eax* 寄存器中的结果值传送到了变量 m 在栈内存上的存储位置。

至于该类别下的其他运算符，因为它们的本质都是访问位于某个内存地址上的数据，因此实现方式大同小异。这里我就不展开介绍了，建议你试着自己探索下，毕竟“实践出真知”。如果有问题，可以在评论区提出来，我们一起交流。

其他运算符

最后，让我们来看看除了上面那六类运算符之外的其他运算符，这里我主要介绍 sizeof 运算符和强制类型转换运算符 “(type) a”。至于函数调用运算符，由于内容较多，我会在后续的课程再单独为你介绍。我们还是通过一段示例代码，观察它们在默认情况下的汇编实现：

```
1  #include <stdio.h>
2  void foo(void) {
3    size_t n = sizeof(int);
4    short f = (short) n;
5    printf("%zd %d", n, f);
6  }
7  int main(void) {
8    foo();
9    return 0;
10 }
11
```

```
1  .LC0:
2          .string "%zd %d"
3  foo:
4          push    rbp
5          mov     rbp, rsp
6          sub     rsp, 16
7          mov     QWORD PTR [rbp-8], 4
8          mov     rax, QWORD PTR [rbp-8]
9          mov     WORD PTR [rbp-10], ax
10         movsx   edx, WORD PTR [rbp-10]
11         mov     rax, QWORD PTR [rbp-8]
12         mov     rsi, rax
13         mov     edi, OFFSET FLAT:.LC0
14         mov     eax, 0
15         call    printf
16         nop
17         leave
```

其中，sizeof 运算符是一个编译期运算符，这意味着编译器仅通过静态分析就能够将给定参数的大小计算出来。因此，在最终生成的汇编代码中，我们不会看到 sizeof 运算符对应于任何汇编指令。相反，运算符在编译过程中得到的计算结果值，将会以字面量值的

形式直接“嵌入”到汇编代码中使用。你可以从上图中右侧红框内的汇编代码看到，C 代码 `sizeof(int)` 的计算结果 4 直接作为了 `mov` 指令的一个操作数。

至于强制类型转换运算符呢，其实也很好理解。这里，我们将变量 n 的值类型由原来的 `size_t` 转换为了 `short`。你可以从上图中蓝框内的汇编代码里看到，当 `mov` 指令将变量 n 的值移动到变量 f 所在的内存区域时，它仅移动了这个值从低位开始一个 WORD，即 16 位大小的部分。至于其他类型之间的转换过程，你可以简单理解成**对同一块数据在不同机器指令下的不同“解读”方式**。

在高编译优化等级下，上面介绍的成员访问运算符与强制类型转换运算符的实现方式并没有发生本质的变化。而与其他运算符类似的是，编译器会减少对栈内存的使用。同时，基于更强的静态分析能力，编译器甚至可以提前推算出某些变量的取值，并省去在程序运行过程中再进行类型转换的过程，从而进一步提升程序的运行时性能。

总结

好了，讲到这里，今天的内容也就基本结束了，我来给你总结一下。

今天我主要围绕 C 语言中的基本“计算单元”，运算符，讲解了 C 语言中的几类不同运算符是如何被编译器实现的。具体总结如下：

- 通常来说，算数、关系、位、赋值运算符的实现在大多数情况下，都会直接一一对应到特定的汇编指令上；
- 逻辑运算符的实现方式则有些不同，它会首先借助 `test` 、 `cmp` 等指令，来判断操作数的状态，并在此基础上再进行相应的数值转换过程；
- 在成员访问运算符中，取地址运算符一般对应于汇编指令 `lea` ，解引用运算符则可直接使用 `mov` 指令来实现；
- 对于其他运算符，`sizeof` 运算符会在编译时进行求值，强制类型转换运算符则直接对应于不同指令对同一块数据的不同处理方式。

思考题

除了我在上面提到的三个问题外，我再给你留个思考题：编译器是通过哪类指令来实现三元运算符 “?:” 的？你可以自己动手实践，并在评论区给出你的答案。

今天的课程就结束了，希望可以帮助到你，也期待你在下方的留言区和我一起讨论。同时，欢迎把这节课分享给你的朋友或同事，我们一起交流。

上一篇 02 | 程序基石：数据与量值是如何被组织的？

精选留言 (1)

写留言

pedro

2021-12-10

实践了一下，三目运算符实际就是一个简化的分支跳转指令tips，如下：

```
cmp DWORD PTR [rbp-4], 1
jle .L2
mov eax, 2
jmp .L3...
```

展开

1