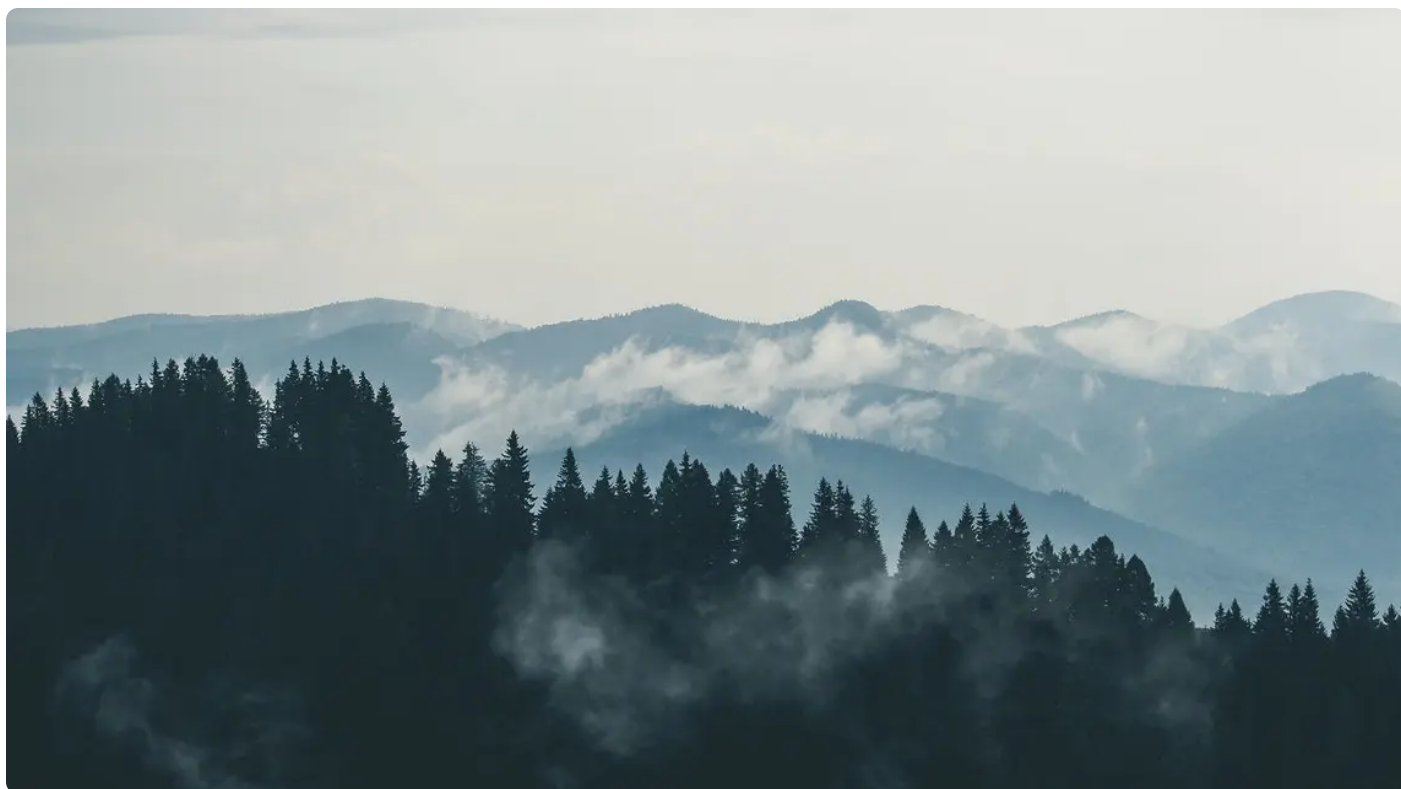


08 | 操控资源：指针是如何灵活使用内存的？

2021-12-24 于航

《深入C语言和程序运行原理》

课程介绍 >



讲述：于航

时长 13:30 大小 12.38M



你好，我是于航。

“指针”是 C 语言为我们提供的最为强大的武器之一。借助指针，我们可以更加灵活地使用应用程序所享有的内存。

不同于 Python、Java 等语言，C 语言为我们提供了这样一种能力：可以让程序员根据需要，主动选择使用“按值传递”或“按指针传递”这两种不同的数据引用方式。通常，按值传递会涉及原始数据的复制过程，因此在某些情况下，可能会引入额外的性能开销。而按指针传递则使程序内存中的“数据共享”成为了可能。

领资料

这一讲，就让我们来一起看下，在 C 语言中指针都有哪些使用方式，以及在语法背后，这些方式都是如何通过机器指令来实现的。

指针的基本使用

使用 C 语法定义变量时，通过为类型说明符添加额外的 “*” 符号，我们可以定义一个指向该类型数据的指针。不仅如此，通过添加额外的 `const` 关键字，我们还能够限制使用该指针变量时所能够进行的操作类型。

比如下面这个例子中，我们便定义了这样的一个指针。通过添加 `const` 关键字，编译器限制了对指针 `npA` 的使用，使得它自身无法被重新赋值，并且也无法通过它来修改所指向的数据。

```
1  #include <stdio.h>
2  int main(void) {
3      int n = 10;
4      const int* const npA = &n;
5      int* npB = &n;
6      *npB = 20;
7      printf("The value is: %d\n", n);
8      printf("Are they the same: %d", npA == npB);
9      return 0;
10 }
11
```

```
1  .LC0:
2      .string "The value is: %d\n"
3  .LC1:
4      .string "Are they the same: %d"
5  main:
6      push    rbp
7      mov     rbp, rsp
8      sub     rsp, 32
9      mov     DWORD PTR [rbp-20], 10
10     lea     rax, [rbp-20]
11     mov     QWORD PTR [rbp-8], rax
12     lea     rax, [rbp-20]
13     mov     QWORD PTR [rbp-16], rax
14     mov     rax, QWORD PTR [rbp-16]
15     mov     DWORD PTR [rax], 20
16     mov     eax, DWORD PTR [rbp-20]
17     mov     esi, eax
```

指针不仅在 C 语言中的使用方式很简单，它在机器指令层面的实现也十分简单。还记得我们在 03 讲 中最后介绍过的取地址 “&” 与解引用 “*” 运算符吗？通过使用这两个运算符，我们便能够完成对指针的最基本，也是最重要的两个操作，即取值与赋值。

观察上图中红框与蓝框内的 C 代码与汇编代码，我们来快速复习一下相关内容。取地址运算符可以用来获取内存中某个数据的所在地址，该过程一般会通过红框内的 `lea` 指令来实现，而解引用的过程正与此相反。如右侧蓝框内的第二行代码所示，直接通过 `mov` 指令，我们便可以按照所指向数据类型的固定大小（这里为 `DWORD`，即 32 位），来与对应内存地址上存放的数据值进行交互。

指针与数组

除了我们显式定义的各类指针变量外，指针与数组也有着千丝万缕的联系。数组是一块连续存放有相同类型数据的内存区域。在 C 语言中，数组有不同的使用方式，有些使用方式可能导致其被退化（`dacay`）为相应的指针类型。我们来看下面这个例子。



```

1 #include <stdio.h>
2 int sum(int arr[], size_t len) {
3     int total = 0;
4     for (; len > 0; len--)
5         total += *(arr + len - 1);
6     return total;
7 }
8 int main(void) {
9     int arr[] = { 1, 2, 3, 4 };
10    printf("%d", sum(arr, sizeof(arr) / sizeof(int)));
11    return 0;
12 }
13

```

```

1 > sum: ...
8 > .L3: ...
17 > .L2: ...
23 .LC0:
24     .string "%d"
25 main:
26     push    rbp
27     mov     rbp, rsp
28     sub     rsp, 16
29     mov     DWORD PTR [rbp-16], 1
30     mov     DWORD PTR [rbp-12], 2
31     mov     DWORD PTR [rbp-8], 3
32     mov     DWORD PTR [rbp-4], 4
33     lea     rax, [rbp-16]
34     mov     esi, 4
35     mov     rdi, rax
36     call    sum
37     mov     esi, eax
38     mov     edi, OFFSET FLAT:.LC0
39     mov     eax, 0
40     call    printf
41     mov     eax, 0
42     leave
43     ret

```

从上图左侧红框内的 C 代码中可以看到，我们在主函数内定义了一个包含有 4 个整型元素的数组 `arr`。在默认情况下，数组中的元素会以相邻的方式分配在连续的栈内存中。从右侧红框内的汇编代码中，我们可以验证这一点。

紧接着，通过调用名为 `sum` 的函数，我们可以求得数组内所有元素的累加和。该函数共接收两个参数，第一个为目标数组，第二个为该数组包含的元素个数。这里，我们直接将 `arr` 作为第一个参数传入。而此时，通过 `sizeof` 运算符，我们也能在编译时得到有关数组 `arr` 的大小信息，并动态计算出数组中元素的个数。

但当数组 `arr` 作为实参被传入函数 `sum` 后，事情发生了变化。从上图右侧蓝框内的汇编代码中可以看出，函数被调用前，`rdi` 寄存器内存放的是 `rbp-16`，也就是数组 `arr` 首个元素对应地址的值。因此，传递给函数 `sum` 的第一个参数实际上为一个指向 `int` 类型的指针，而有关数组 `arr` 的大小和类型的信息在此时已经全部丢失。对于这种情况，我们一般称其为“数组的退化”，即**数组类型退化为指针类型**。

指针的其他运算

在 C 语言中，除了可以对指针进行基本的解引用、赋值，甚至再次取地址的操作外，我们还可以对它进行算数与关系运算。但需要注意的是，指针的这两种运算不同于一般的数值类型。比如，对指针进行加法运算，就并不是将加数直接累加在对应的地址值上这么简单。你也可以再回顾一下上面讲解指针和数组时的示例代码，从函数 `sum` 的实现中，可以看到我们对退化指针 `arr` 的算数运算过程。



算数运算

总的来看，我们可以对指针类型进行这样几种算数运算：

- 单个指针与另一个整数相加 / 相减；
- 单个指针自增 / 自减；
- 两个指针求差。

指针在进行算数运算后，不能将其指向的、以固定长度字节作为整体的数据值“拆分”。因此，当我们对指针进行加法、减法、递增、递减运算时，编译器实际上是以当前指针所指向值对应的某个固定长度为单位，对指针中存放的地址值进行相应调整的。同样，对于指针之间的求差操作，求得的也并不是两个地址值之间以字节为单位的差，而是用这个差值除以上面提到的固定长度所得到的结果。

下面，让我们通过一个例子，来看看**编译器是如何在背后处理针对指针的算数运算的**。这里我介绍的是“单个指针与另一个整数相加”这种场景。由于其他指针算数运算的过程与此基本类似，相信理解了这一种，另外几种你也能融会贯通。

```
1  #include <stdio.h>
2  int main(void) {
3      int arr[][3] = {
4          { 1, 2, 3 },
5          { 4, 5, 6 } };
6      printf("%d\n", **(arr + 1)); // 4.
7      printf("%d\n", *(*arr + 2)); // 3.
8      return 0;
9  }
10
```

```
1  .LC0:
2      .string "%d\n"
3  main:
4      push    rbp
5      mov     rbp, rsp
6      sub     rsp, 32
7      mov     DWORD PTR [rbp-32], 1
8      mov     DWORD PTR [rbp-28], 2
9      mov     DWORD PTR [rbp-24], 3
10     mov     DWORD PTR [rbp-20], 4
11     mov     DWORD PTR [rbp-16], 5
12     mov     DWORD PTR [rbp-12], 6
13     lea     rax, [rbp-32]
14     add     rax, 12
15     mov     eax, DWORD PTR [rax]
16     mov     esi, eax
17     mov     edi, OFFSET FLAT:.LC0
18     mov     eax, 0
19     call    printf
20     lea     rax, [rbp-32]
21     add     rax, 8
22     mov     eax, DWORD PTR [rax]
23     mov     esi, eax
24     mov     edi, OFFSET FLAT:.LC0
25     mov     eax, 0
26     call    printf
```

领资料


这里在 `main` 函数的开始，我们定义了一个名为 `arr` 的，具有 2 行 3 列，共 6 个元素的二维数组。从右侧对应的汇编代码中，可以看到这个数组内部的数据是以地址连续的方式被存放在栈内存中的。对于这个存储方式，你可以将其理解为**编译器对 C 代码中的多维数组进行的扁平化（flatten）处理**。

在接下来的 C 代码中，我们通过指针的方式获取并打印了位于数组 `arr` 中两个不同位置上的值。其中，蓝框内的表达式首先对二级指针 `arr` 进行了加一操作，然后返回了对这个经过“累加”后的地址进行两次解引用的结果值。从右侧相应的汇编代码中可以看到，对指针 `arr` 的加一操作导致 `rax` 寄存器中的值被增加 12。而该寄存器中原先存放有数组中第 1 行第 1 列元素对应的地址值，因此在经过计算后，我们得到了一个指向元素 4 的二级指针。

也就是说，对变量 `arr` 进行加一操作，导致该指针向栈中的高地址方向移动了 12 个字节。之所以会有这样的变化，是因为 `arr` 作为一个二级指针，它在这里所直接指向的数据，实际上是二维数组中的每一个包含有 3 个整型元素的一维数组。而每一个一维数组的大小都为固定的 12 字节。因此，当对指针 `arr` 进行算数运算时，编译器便会以它所指向的一维数组的大小为单位，来进行地址上的调整。

同样地，对于黄框内的第二次数组元素访问，由于 `*arr` 作为一级指针（经过了一次解引用），直接指向的是二维数组内某个一维数组中的整型元素，因此，对它进行加法运算，将会以 4 字节作为单位来进行地址上的调整。

这里我给你留一个小问题：按照类似的计算方式，你能否直接推算出下面这行语句在执行后的输出结果？欢迎在评论区留下你的答案。

 复制代码

```
1 printf("%d\n", *((*(arr + 1) + 1)); // ?
```

最后，需要注意的是，**指针的算数运算在绝大多数情况下都只适用于数组相关的指针**。而在其他场景中，即使程序可以正常编译运行，但由于标准中可能并未要求编译器的具体求值规则，因此其行为是未定义的，程序的运行结果无法得到保障。

 领资料

关系运算

除了算数运算外，同一类型的不同指针之间还可以进行关系运算。

我已经在 [03 讲](#) 中介绍了关系运算符的机器指令实现方式。在大多数情况下，编译器会配合使用 `cmp` 与 `setg` 等指令来判断关系运算符两侧操作数的大小，并根据判断结果，进行相应的置位与复位操作，最终返回 0 或 1 作为结果。而对于指针之间的关系运算来说，其实现方式也是如此。

但需要注意一点：虽然在机器指令层面，指针的关系运算实际上是对指针内部所存放的地址值进行的大小判断，但从 C 语法的角度来看，具有实际意义的指针关系运算仅有为数不多的几种情况（你可以点击 [这个链接](#) 来详细了解）。除此之外，其他使用方式均会产生未定义行为（UB）。

堆内存指针

在我之前介绍的例子中，指针仅引用了位于栈内存中的数据。但实际上，指针还有另一个更重要的作用，那就是给予了我们**灵活操控堆内存中数据**的能力。

堆同栈类似，也是位于进程 VAS 中的一段专门用于存放数据的内存空间。栈中的数据随着函数的调用与返回，会被程序自动释放，而堆则有所不同。在堆中进行数据分配，需要借助特定的操作系统调用函数，并且被分配内存中的数据不会随着程序的运行而自动清除。因此，当这些数据不再被程序使用时，便需要显式地调用相应的系统函数，来将其释放。

幸运的是，C 标准库中已经为我们封装好了这样的一些函数。借助它们，我们可以方便地申请与释放堆内存，并享受堆分配算法带来的性能保障。这里我先带你回顾一下这些函数的使用方式，而它们的具体内容，我会在 15 讲 中再为你详细介绍。

通过下面这段示例代码，我们可以快速回顾一下标准库函数 `malloc` 与 `free` 的使用方式。对于其中的关键语句，你可以参考它们上方的注释。

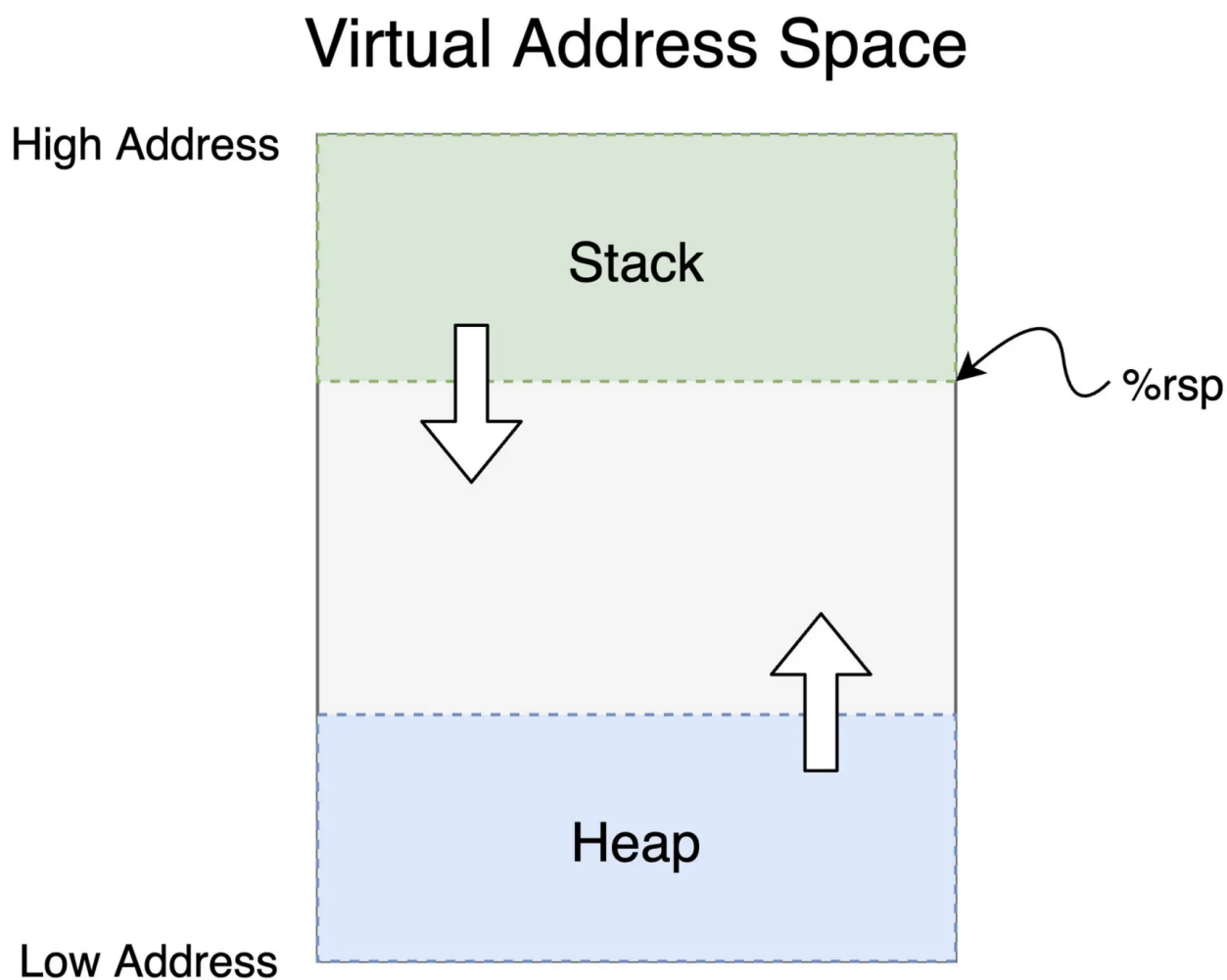
 复制代码

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #define N 5
5 int main(void) {
6     int arr[] = { 1, 2, 3, 4, 5 };
7     // 分配用于存放 N 个整数的堆内存;
8     int* p = (int*) malloc(sizeof(int) * N);
9     // 将数组 arr 中的元素复制到分配的堆内存中;
10    memcpy(p, arr, sizeof(int) * N);
11    for (int i = 0; i < N; ++i) {
```

领资料

```
12 // 通过指针遍历堆空间中的数据；
13 printf("%d\n", *(p + i));
14 }
15 // 释放先前分配的堆空间，让操作系统可以回收内存；
16 free(p);
17 return 0;
18 }
```

在 VAS 中，堆内存的位置处于栈内存的“下方”，即低地址方向。与栈内存相反的是，堆内存的占用区域将随着程序的不使用从低地址向高地址逐渐增长，如下图所示：



看到这里，你可能会有这样的疑问：我平时写的 C 程序只需要临时变量就够用了，这些变量的值会被分配在栈内存中，那我们为什么还需要堆呢？

因为栈上的数据在函数返回时就会被释放，因此我们只能通过不断拷贝的方式保持其“存活”。而全局变量和静态变量的生存期虽然与整个程序保持一致，但也并没有办法在程序的运行过程

中动态生成，且缺乏一定表现力。

而堆内存则可以很好地解决这些问题。存放在其内部的数据能够由程序动态地创建，而且可以保持与程序相同的最大生存期。不仅如此，和全局变量、静态变量这两种将值完全暴露给所有程序代码使用的方式相比，使用堆内存可以将数据的使用，限制在其所需要的最小范围内，这无疑加强了程序对内存资源的精细化管理程度。

使用指针的注意事项

借助指针，我们可以灵活地使用程序存放在堆内存与栈内存中的数据，但不当的指针使用方式也可能导致程序出现难以调试、甚至是难以复现的 **BUG**。其中，你需要特别注意避免下面这些操作，因为它们会导致程序出现无法预测的未定义行为：

- 解引用未初始化的指针；
- 函数返回指向其内部局部变量的指针；
- 非指向同一数组内元素的两个指针之间的减法操作；
-

除此之外，对堆指针进行有效的生命周期管理，也是我们在构建程序时需要注意的问题。由于同一个堆指针可能会在程序的不同函数中被使用，因此就要特别注意：我们应该通过 **free** 函数及时清理堆内存，以防止内存泄露；同时，又不应该去释放一块已经被释放过的堆内存（重复释放会产生异常）。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

这一讲，我主要介绍了 C 语言中有关指针的一些话题，包括指针在 C 语言中的基本使用方式、指针与数组的关系、指针的算数与关系运算，以及它们在机器指令层面的实现细节。同时，我还介绍了堆内存指针，并和你简单探讨了在使用 C 指针时需要注意的一些问题。



在 C 代码中，通过添加特定的 “*” 符号，我们可以声明所定义变量为一个指针类型。而与指针有关的两个常用操作符为取地址操作符 “&” 与解引用操作符 “*”，它们一般可以通过 **lea** 指令与 **mov** 指令来实现。

指针与数组也有着密不可分的联系。在某些特定的使用方式下，编译器会将数组类型退化为指针类型，导致其丧失了有关数组的类型与大小等信息。

除此之外，指针类型还可以参与算数与关系运算。其中，算数运算主要涉及指针与整数的加 / 减运算、指针的自增 / 自减运算，以及两个同类型指针间的求差运算。而关系运算则同数值类型保持一致。但需要注意的是，标准中仅规定了上述运算类型对于指针的有限使用方式，而规定之外的使用方式则属于未定义行为。

同时，我还介绍了可以引用堆上数据的指针。堆是除栈之外的又一个重要的数据存放“容器”。相较于栈上数据，以及全局变量或静态变量中的数据，位于堆中的数据具有更加灵活的生存期，并且能够在程序运行过程中动态生成。

最后，我总结了在使用 C 指针时需要注意的问题。对指针的不当使用会使程序产生标准中未定义的行为。对于堆指针来说，除了未定义操作外，没有及时对相关资源进行清理，或重复清理，都会导致程序的运行产生异常。而这些都是我们在设计 C 程序结构时，需要特别注意的问题。毕竟，再强大的武器也是一把双刃剑。

思考题

下面这段代码可以正常编译吗？为什么？

 复制代码


```
1 #include <stdio.h>
2 int main(void) {
3     int arr[] = { 1, 2, 3, 4 };
4     printf("%d", arr[3] == 3[arr]);
5     return 0;
6 }
```

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

 领资料

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

上一篇 07 | 整合数据：枚举、结构与联合是如何实现的？

下一篇 09 | 编译准备：预处理器是怎样处理程序代码的？

更多课程推荐

操作系统实战 45 讲

从 0 到 1, 实现自己的操作系统

彭东
网名 LMOS
Intel 傲腾项目关键开发者



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言 (9)

💬 写留言

📁 领资料



sky

2021-12-25

有个疑问，算数运算一节，右侧汇编代码样例中`sub rbp 32`，说明栈分配了32字节，但二维数组实际只占 $4*6=24$ 字节，剩余8字节是做什么用的？

作者回复: 这是一个好问题！实际上这是由于 SysV 规范要求函数调用前，栈顶需要在 16 字节的边界对齐（我们曾在第 05 讲中提到过）。因此由于这里在栈中需要存放 6 个元素，需要至少 24 字节的空间。所以为了对齐到 16 字节，`rsp` 的值要减少至少 32 字节。



=

2022-01-07

`printf("%d\n", *(*arr + 1) + 1);` // 结果为5

`arr+1`后，二级指针指向`arr[1]`

`*(arr+1)`后，一级指针指向`arr[1][0]`

`*(arr + 1) + 1`后，一级指针指向`arr[1][1]`

`*(*(arr + 1) + 1)`后，解引用出结果5

文末的代码可以编译通过，而且`arr[3]`



👍 1



qinsi

2021-12-25

因为 $a[i] = *(a + i)$

所以 $x[y[i]]$

$= x[* (y + i)] = x[* (i + y)] = x[i[y]]$

$= *(x + i[y]) = *(i[y] + x) = i[y][x]$

在用到多级间接下标时可以避免嵌套中括号



👍 1



l

2021-12-24

答案是5，`arr+1`为`arr1 0`，然后再加一为`1 1`，所以为5

作者回复: 答案正确!



zxk

2022-01-22

老师，关于“在某些特定的使用方式下，编译器会将数组类型退化为指针类型”，那这个是属于使用不当，还是一种优化手段？不太明白这个行为会有什么后果及应用场景。

领资料

作者回复: 好问题。我们这里讲到的“数组退化到指针”实际上是 C 语言在设计上的一种性质，并不是使用不当或者优化手段。这个性质出现的原因要追溯到 C 语言的前身 B 语言中的类似设计上。在大多数场景下，需要注意问题就是变量性质（数组大小、数组类型）的丢失。





Victor

2021-12-26

虽然编译过了，但这个可读性太差了

作者回复: 哈哈是的，所以正常项目中不要使用哈。



Geekim

2021-12-25

原文中的

使用指针的注意事项

借助指针，我们可以灵活地使用程序存放在堆内存与栈内存中的数据，但不当的指针使用方式也可能导致程序
但不当应该是当不但

共 2 条评论 >



无双BaOY_WHA

2021-12-24

`arr[3]` 和 `3[arr]` 映射到汇编代码上是一样的。但没见过 `3[arr]` 这种写法...



pedro

2021-12-24

竟然真的可以编译，从汇编上看 `3[arr]` 的值是从编译期就确定的，活久见，目前还未见过这种数组取值方式。

共 2 条评论 >



领资料

