

知识挖掘机

通过 C 语言入门计算机科学

通过 C 语言你可以实践一下操作系统、数据结构、甚至硬件



韩大 2024-10-26

本文系用户投稿，不代表机核网观点



收听本文 09:32



未经授权 禁止转载

前言

以前刚刚接触计算机的时候，对于很多计算机的问题很感兴趣，譬如为什么有一个硬件叫“内存”，而且还那么贵；为什么会有一个叫硬盘的设备，和内存又有什么区别等等。后来看了

很多科普的文章和书籍，得到的大多数是一些“比喻”的说明，虽然有一点概念，但总觉得不是非常彻底。后来也看了网上的公开课，譬如哈佛的《计算机导论》，发现讲的非常有趣，但还是不够“直白”，我希望能可以直接用某些方法，去“搞”一下计算机，而 C 语言，就是这种工具。当你用过 C 语言之后，对于计算机上因为各种 bug 不经意露出来的某些“信息”，你就能大概了解是什么意思。虽然知道这个也不代表能怎么样，但能满足自己的好奇心也是很不错的。

从 HelloWorld 说起



现在很多手机、电脑用户，会发表对于“系统”的看法：譬如 Mac 系统比 Windows 高级；IOS 用起来比安卓舒服……而作为程序员，我对于“操作系统”的看法，往往和使用界面完全无关，那个只是一个叫资源管理器的程序。我真正在意的是——我写的程序，能如何“使用”这个操作系统。



大家多多少少都可能见过的 Hello, World! 程序：

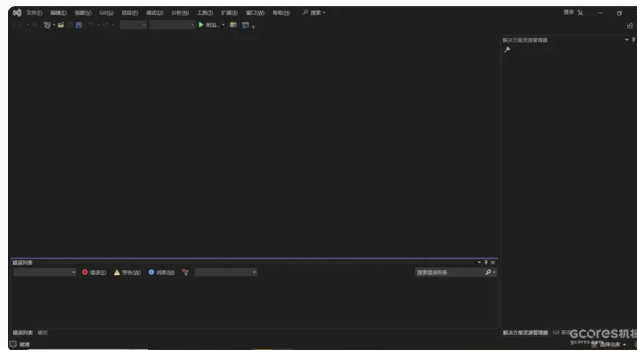
```
#include <stdio.h>

int main() {
    printf("Hello, World!");
}
```

一般的教程，到这里要么长篇累牍的要求你安装一套 C 语言的开发工具，要么直接不管你死活，直接开始用一个不知道哪里来的程序开始编译了。其实，开发工具这个环节，是一个很有价值的知识点。不过我也不打算说的太详细，因为后面会一点点了解。

因为我用的是 Windows 的笔记本，所以我会安装一套能编译 C 语言的开发工具，我选择的是现在最时髦的编译器是 LLVM；以及我被迫选择，不安装就没法编译的 MicroSoft Visual Studio 2022 社区版（免费）：

1. LLVM 的安装方法：请搜索“llvm windows 安装”，如果装好了，在 cmd 或者 powershell 里面就可以用到命令：clang
2. MicroSoft Visual Studio 2022 社区版的安装方法：<https://visualstudio.microsoft.com/zh-hans/vs/community/>。需要注意一定安装其中的“**Windows11 SDK**”这个组件（是否 win11 根据你的电脑上的 windows 版本来定），因为所有操作系统的标准库都在这个组件里，没有的话最后无法生成可执行的 exe 文件。



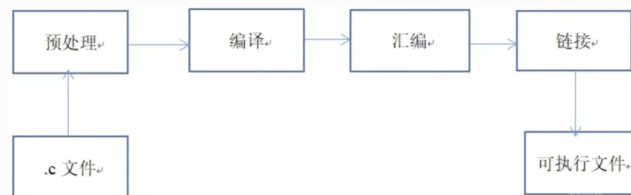
装好了你可以启动这个软件，但其实我一次都不会打开它

一旦装好了上面的东西，就可以把上面的代码保存为 hello.c，然后用命令行（譬如 cmd 或者 powershell）属于这个“神奇”的命令：

```
clang hello.c
```

然后你就得到了一个叫 a.exe 的可执行程序。运行这个程序 `.\hello.exe`，就能看到打印了那行字 Hello, World!。

如果你很顺利的走到这一步，就会觉得 C 语言也没啥特别嘛，但其实上面那行 `clang hello.c`，其实包含了非常复杂的步骤。一个文本文件，变成一个可以在 windows 运行的 exe 文件，需要经过四大步骤，而这四个步骤，我们都可以通过编译器 clang 一步步的呈现出来。



预处理

如果你仔细看 hello.c 的源码，你会发现这一行代码：

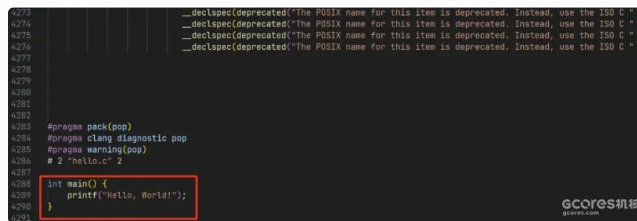
```
#include <stdio.h>
```

这里的文字，其实不算 C 语言的内容。这是编译器提供的一个“预处理”功能：把“系统库”的 `stdio.h` 这个文件，复制到此处。我们可以通过下面这个命令，来得到第一步“预处理”之

后的文件。

```
clang -E hello.c -o hello.i
```

这个命令的参数 -E 表示的就是只做预处理的意思，-o 参数表示结果文件名为后面的 hello.i，当你运行完这个命令，你就会得到一个 hello.i 文件，打开这个文件，你会发现有大量的文字，而在文件最后，才是你写的 hello world 源码：



```
4273      __declspec(deprecated("The POSIX name for this item is deprecated. Instead, use the ISO C "
4274      __declspec(deprecated("The POSIX name for this item is deprecated. Instead, use the ISO C "
4275      __declspec(deprecated("The POSIX name for this item is deprecated. Instead, use the ISO C "
4276      __declspec(deprecated("The POSIX name for this item is deprecated. Instead, use the ISO C "
4277
4278
4279
4280
4281
4282
4283      #pragma pack(pop)
4284      #pragma clang diagnostic pop
4285      #pragma warning(pop)
4286      # 2 "hello.c" 2
4287
4288      int main() {
4289          printf("Hello, World!");
4290      }
4291
```

这个文件前面 4000 多行，都是 stdio.h 的内容，只有最后三行是你写的源代码。这就是预处理的工作，这个时候你可能会问，为啥需要预处理呢？这个问题很好，我们后面可以慢慢说，特别是 #include 这个预处理功能。当然预处理功能除了包含文件，还有其他很多，但最常见的是这个。

编译

第二步是把预处理之后的文本文件 hello.i，变成机器能理解代码。cmd 窗口输入以下：

```
clang -S hello.i -o hello.s
```

这里的 -S 表示进行编译，生成“汇编文件”，同样，-o 命令指定结果文件保存在 hello.s 里。如果我们打开 hello.s，你会看到还是一对文本，但是这些文本都是“汇编语言”的。

```
230      movq    %r8, 120(%rsp)
231      movq    %rdx, 112(%rsp)
232      movq    %rcx, 104(%rsp)
233      movq    176(%rsp), %rax
234      movq    %rax, 88(%rsp)           # 8-byte Spill
235      movq    128(%rsp), %rax
236      movq    %rax, 80(%rsp)           # 8-byte Spill
237      movq    120(%rsp), %rax
238      movq    %rax, 72(%rsp)           # 8-byte Spill
239      movq    112(%rsp), %rax
240      movq    %rax, 64(%rsp)           # 8-byte Spill
241      movq    104(%rsp), %rax
242      movq    %rax, 56(%rsp)           # 8-byte Spill
243      callq   __local_stdio_printf_options
244      movq    56(%rsp), %rdx           # 8-byte Reload
245      movq    64(%rsp), %r8           # 8-byte Reload
246      movq    72(%rsp), %r9           # 8-byte Reload
247      movq    80(%rsp), %r10          # 8-byte Reload
```

“汇编语言”是一种方便人类阅读机器码的格式，这里面每一行文字，都代表了一条真正的机器码指令。由于机器码指令是完全以数字表达的，人很难分辨，所以设计了一些类似英语单词的文本指令来对应。总而言之，这一步的 hello.s 文件，里面已经包含了每个机器代码的信息。这些机器代码将可以输入给计算机的 CPU 来做一些事情。

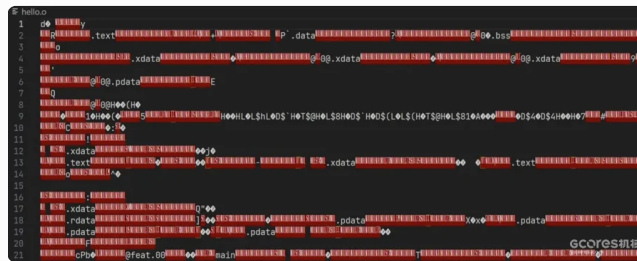
汇编

接下来我们可以把 hello.s 的汇编内容，转换成真正的机器码，也叫二进制格式（实际上我不喜欢“二进制”这个说法，因为计算机里所有文件都是二进制的）。命令如下，-c 表示编译出二进制“目标文件”：

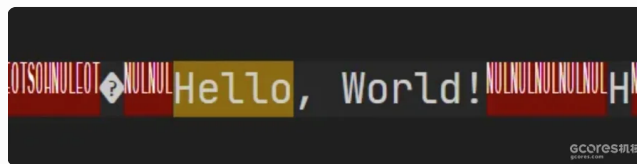
```
clang -c hello.s -o hello.o
```

这一步的产物 hello.o 里面，真正的就是全部“二进制”的内容了，一般的文本编辑器是无法

打开和显示里面的内容的。



但是如果你锲而不舍的看，还是能看到你要打印的字符的：



但是到了这一步，还没有生成 exe 文件。你可能会怀疑，不是已经生成机器码了吗？为啥还不是可执行的文件呢？这也是之前我的疑问。

我们看看生成的文件大小：4,413 字节，而之前编译的 a.exe 文件大小是：142,848 字节。也就是说还有很多内容没有包含进来，而这些没有包含进来的内容，就是你写的 main() 这个函数之所以可以在 windows 上启动的程序，以及 printf() 这个函数真正的功能所在的程序。

hello.o 实际是 hello.c 所编写的程序，但启动这个程序，并且在屏幕上显示字符这些功能，都是 windows 操作系统提供的功能。只有把你的写的程序和 windows 提供的程序连接起来，这个程序才能真正的被执行。所以我们需要下面这个步骤——链接

链接

最后这步，我们终于生成了可执行文件：

```
clang hello.o -o hello.exe
```

这里的不再需要什么别的参数，只用 -o 参数指定生成的 exe 文件名就好了。

刚刚我们说了，我们自己写的程序，需要和操作系统 windows 的功能连接起来，那么到底是“链接”了哪一部分呢？我们可以用下面的代码来查看(powershell)：

```
&"C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.38.33130\bin\Hostx64\x64\dumpbin.exe" /dependents .\hello.exe
```

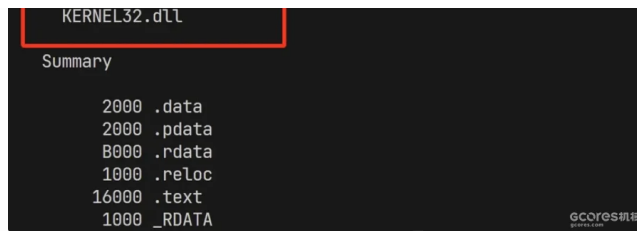
这里使用了 Microsoft Visual Studio 2022 社区版所附带的一个工具 dumpbin.exe，它可以告诉我们一个 exe 文件链接了一些什么“库”文件。而这些“库”文件，就有 windows 所提供的。下面是命令执行的结果：

```
Microsoft (R) COFF/PE Dumper Version 14.38.33134.0
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file hello.exe

File Type: EXECUTABLE IMAGE

Image has the following dependencies:
```

这里的这个 KERNEL32.dll，就是 windows 提供的一个库文件。提供了程序启动、打印文字到屏幕等等很多功能。我们写的 hello.exe 依赖这个库文件来实现功能。这个 dll 文件，就放在 `c:\windows\system32\` 目录下，也就是操作系统安装的目录。因此操作系统其实有很多功能，其实是为了提供各种程序运行的，而远不止是我们常用的安装、管理各种软件的图形界面。



如果你没有在 Microsoft Visual Studio 中安装 Windows11 SDK 这个组件的话，你就会得到一个错误信息：`LINK : fatal error LNK1104: 无法打开文件“kernel32.lib”`，这个错误信息说明了这个 exe 程序，需要和 kernel32 这个库进行链接。



在这最后一步，我们接触了 dll 文件，也就是“库”文件。C 语言学习和使用中，最让人头秃的问题之一，就包括和“库”相关的各种问题。下一篇，我会仔细说说这个“库”到底是怎么回事。



C 语言
8 作品



科技

C语言

👍 117

📖 150



韩大

643 人关注

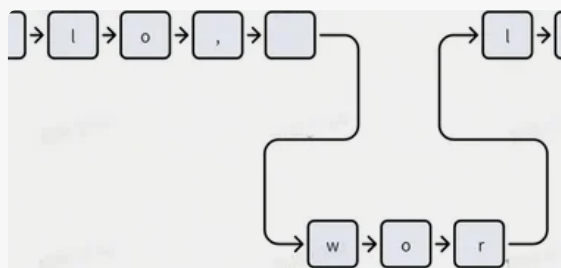
关注



知识挖掘机

39810 人关注

订阅



知识挖掘机

通过C语言入门计算机科学7: 数据结构



韩大

2 天前

👍 29 💬 5

评论区



请先登录再发表评论

发送

共 17 条评论

热门 最新



project痛 · 2024-10-28

工作打开app摸鱼，又看到工作内容

👍 11 🗨️ 回复 · 举报

作者喜欢



cameLcAsE · 2024-10-26

感叹，要是每个大学教编程都像这样从如何用编译器工具链就不会有那么多到毕业都不分不清编译器和编辑器和IDE的人了😂

👍 6 🗨️ 回复 · 举报

作者喜欢



拍照莫比剪刀手 · 2024-10-28

读完后一点感触：我是非科班程序员（电气），工作后才转的软件工程师，看这篇文章觉得很有感触。我平时开发嵌入式C都是在芯片厂家提供的IDE工具里面进行的，IDE工具集成好了很多编译工具链和代码调试的小工具方便开发人员，但也有坏处：我对于代码从C文件到最后编译生成出来的ELF/HEX文件中间的过程是一头雾水且一知半解；在工作中我又偶尔会涉及到软件测试、内存分配等环节不得不关心C语言的编译过程...

博主的文章写的思路很清晰，请继续加油，早日更新下一篇文章

👍 4 🗨️ 回复 · 举报

作者喜欢

已全部加载



Gcores 机核



机核从2010年开始一直致力于分享游戏玩家的生活，以及深入探讨游戏相关的文化。我们开发创作者。

我们坚信游戏不止是游戏，游戏中包含的科学，文化，历史等各个层面的知识和故事，它们同热爱游戏的您。

知乎 微博 微信 播客 吉考斯工业 核市奇谭 RSS

营业执照

增值电信业务经营许可证 京B2-20191060

京ICP备17068232号-1

网络文化经营许可证京网文[2024]1733-082号

 京公网安备 11010502036937号

食品经营许可证 JY11105052461621

出版物经营许可证 新出发京零字第亦230031号

联系我们 / CONTACT US 投稿须知 用户协议 隐私政策 社区规 工作:

Copyright © 2009 - 2024 GAMF rved

App内打开



通过 C 语言入门计算机科学 2：链接和库 | 机核 GCORES

你是不是曾经怀疑，我们电脑上大量的 dll 文件，里面到底是啥？

在刚刚上面的例子中，我们发现了一个简单的 HelloWorld 程序，也是需要和操作系统发生关系的。这种关系在编译过程里，主要就是通过“链接”操作系统的“库”来体现。那么下一个需要了解的问题就是：编译器软件是通过什么规则来把“库”和你的程序“链接”起来的呢？

可能很多人对于代码里的那行 `#include <stdio.h>` 印象深刻，觉得“链接”这个过程和这行代码一定有极大的关系。而事实上，确实有一点点关系，但关系不大。我们可以来做一个试验，把我们的 HelloWorld 文件 `hello.c` 改成下面这样：

```
int printf(const char *format, ...);
int main()
{
    printf("Hello, World!\n");
}
```

可以看到这样的代码，已经没有那行“神奇”的 `include` 代码了。我们可以尝试编译一下：

```
clang hello.c
```

你会发现，这样照样可以生成 `a.exe`，并且运行结果还是一样的正常！这就说明了，`stdio.h` 里面并没有什么神奇的东西，对于我们这个程序有用的，也就是一行普通的“函数声明”而已！

```
int printf(const char *format, ...);
```

所谓 函数声明：一行用来说明某个函数的调用方式的代码，编译器会根据这行代码，来判断后面的代码，对于此函数的使用格式是否正确。

编译器并不会纠结当前要编译的这个文件里，是否包含每个用到的函数的完整代码（所谓函数的“定义”），只要是能和“函数声明”对的上，就可以编译成功。关于要调用的函数到底在哪里，如何处理，那会留给“链接”步骤去处理。

在“链接”阶段，编译器会从 `hello.o`（中间产物，目标文件）中搜索得到 `printf()` 这个函数没有被“满足”，便会自己能找到的“库”里面搜索，如果找到了能满足 `printf()` 这个声明的“函数定义”，便会把这个库的 `printf()` 函数“链接”到最终产物 `a.exe` 文件里面去。因此最后 `a.exe` 就是一个可以运行的正常程序了。

编译器在默认的情况下，会自己去搜索一系列的“标准库”和“系统库”。而 `printf()` 正是属于这些“标准库”的其中一个函数。而保存了这个函数的代码，就是在 `kernel32.dll` 这个操作系统附带的文件中。需要注意的是，同样这个 `printf` 函数的代码，在 Linux 和 MacOS 上，是在不同名字的库文件里的。

不管是 Windows 还是 Linux，都以 C 语言库的形式，提供了很多功能，以便开发者可以通过编程来使用操作系统的功能。从这个角度来说，操作系统的用户还包含了程序开发者。

从上面的例子中，我们可以感觉到一个事实：C 语言中的“函数”，是基本的程序模块单元。

设想一下，如果我们需要用 C 语言来开发一个大型的程序。这个工作需要很多人参与，并且可能持续一个比较长的开发时间。我们的程序代码应该如何进行划分呢？显然让所有人都把代码加到一个 .c 文件里面，是不可行的。最自然的想法，是能让程序分成很多不同的“模块”，各自单独开发和测试，然后再通过某种机制“组装”在一起。对于 C 语言来说，就是可以把不同的模块，定义为不同的“函数”，写入到不同的 .c 文件中，最后我们可以把这些 .c 文件编译成不同的“库”文件，用链接的方式，生成最终的可执行程序。

很多现代编程语言，都已经把“模块管理”放到语言本身里面的设计了。如 Java 和 Go 都有 `import` 关键字，C# 有 `using` 关键字。唯独 C 语言是缺乏这方面的设计的，所以我们需要额外了解这一套特别的机制。

首先，我们新建一个源文件 `add.c`，里面定义一个我们自己写的函数 `add()`，这个函数功能就是做一个简单的加法。

```
int add(int a, int b)
{
    return a + b;
}
```

C 语言的函数和数学的函数有点像，都有输入参数和返回值：

一个函数的“返回值、名字、参数列表”这一行，就是这个函数的“声明”。如果其他的 .c 文件想调用这个函数，就需要把函数的声明部分写到调用方的源文件里。因此我们在 `hello.c` 里面写上 `add()` 的声明，并且进行调用：

```
int printf(const char *format, ...); // 声明
int add(int a, int b);              // 声明

int main()
{
```

```
int c = add(1, 2);
printf("Hello, World! %d\n", c);
}
```

在 `main()` 中，我们以 `int c = add(1,2);` 这句调用了 `add()` 函数，然后把返回值放在变量 `c` 里面。并且在下面的 `printf()` 函数中，以 `%d` 这个定位符，把变量 `c` 的内容以十进制的格式显示出来。

现在，我们可以尝试编译 `hello.c`，不出意外，编译器找不到 `add()` 函数 在哪里，所以没法链接成一个可执行程序。（编译器也把函数和变量这些名字统称为“符号”）

```
clang hello.c -o hello.exe

hello-834397.o : error LNK2019: 无法解析的外部符号 add，函数 main 中引用了该符号
hello.exe : fatal error LNK1120: 1 个无法解析的外部命令
clang: error: linker command failed with exit code 1120 (use -v to see invocation)
```

为了让编译器能把 `add()` 函数链接成功，我们需要把 `a.c` 文件编译成一个“库”。多个包含了机器指令”的目标文件，可以放入一个“库”文件中，以便一起提供给调用者。我们这次需要先准备好装入“库”文件的唯一一个目标文件，把 `add.c` 编译成 `add.o`：

```
clang add.c -c -o add.o
```

然后用命令 `llvm-lib` 把 `add.o` 文件放入库文件 `add.lib` 中：

```
llvm-lib add.o /out:add.lib
```

在 Linux 下使用 `ar` 命令来生成（静态）库，库文件一般是 `libadd.a`

```
clang hello.c -o hello.exe -ladd
```

注意这里我们的链接命令中，增加了一个 `-ladd` 的参数，意思就是：链接一个叫 `add` 的库。这个参数会让编译器，在当前目录寻找 `add.lib` 这个库文件，然后尝试链接所有未能找到定义的函数。（当然默认的标准库也会一起尝试进行链接）

当我们运行 `hello.exe` 之后，我们可以看到屏幕上显示了 `Hello, World! 3` 这行字，表示确实调用了 `add()` 这个函数。

我们这里生成的 `add.lib` 文件，被称为“静态库”。对于静态库来说的链接，称为“静态链接”，这种链接方式，并不会需要生成出来的 `hello.exe` 还依赖 `add.lib` 文件；而是会把 `add()` 函数的程序代码，直接复制到 `hello.exe` 的内部。——这和我们对于 `printf()` 的链接是不一样的，`printf()` 的代码并没有复制到 `hello.exe` 中，而是继续呆在 `kernel32.dll` 里面，在运行 `hello.exe` 再去调用 `kernel32.dll` 的内容。`printf()` 这种链接方式，我们称为“动态链接”，而动态链接库文件，一般都是用 `.dll` 的名字来命名。

我们下面可以尝试把 `add()` 这个函数，放入一个动态链接库中，让 `hello.c` 来进行链接，看看有什么不一样。

在 windows 上，为了让我们的函数可以被动态链接，我们需要遵守 windows 的规则，修改一下 `add.c` 的代码，主要就是加上个标记 `__declspec(dllexport)`：

```
__declspec(dllexport) int add(int a, int b)
{
    return a + b;
}
```

在 Linux 下，是不需要加这个标记的。

改了 `add.c` 后，我们的 `hello.c` 对应的 `add()` 函数声明也得加上这个标记：

```
int printf(const char *format, ...);
__declspec(dllexport) int add(int a, int b);

int main()
{
    int c = add(1, 2);
    printf("Hello, World! %d\n", c);
}
```

```
clang add.c -c -o add.o
```

然后就可以编译动态库了，注意这次用的是 `clang` 而不是 `llvm-lib` 来生成：

```
clang add.o -shared -o add.dll
```

```
正在创建库 add.lib 和对象 add.exp
```

`add.dll` 包含了运行时 `add()` 函数的程序，`.exe` 文件的运行，依赖这个文件

`add.lib` “导入” 文件，提供给想要调用 `add()` 的程序进行链接使用

`add.exp` “导出” 文件，一般不需要额外关注

```
clang hello.c -o hello.exe -ladd
```

在 Windows 下，链接 `add.dll` 的时候，必须要有 `add.lib`；在 Linux 下则只需要动态库文件本身这一个文件（一般文件名会是 `libadd.so`）就可以了，不像 windows 需要有两个文件。

运行了 `hello.exe` 后，结果和原来链接的静态库效果是一样的。但是我们可以用工具来看到确实是动态链接了。


```
&"C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.38.33130\bin\Hostx64\x64\dumpbin.exe" /dependents .\hello.exe
```

动态链接和静态链接不同之处，就在于运行程序时，dll 文件必须存在。

```
> .\hello.exe  
> Hello, World! 3
```

如果我们删除了 add.dll，然后运行 hello.exe，你就会发现之前显示的 "Hello, World! 3" 消失了！

```
> mv add.dll add.d  
> .\hello.exe  
>
```

动态链接库更加有用的地方，在于如果我们修改了 dll 文件的内容，只要替换文件，exe 文件不需要修改，就能使用修改过之后的功能。譬如我们可以把 add() 函数的功能稍微修改一下，让计算的结果加上 100：

```
_declspec(dllexport) int add(int a, int b)  
{  
    return a + b + 100;  
}
```

```
clang add.c -c -o add.o  
clang add.o -shared -o add.dll
```

我们无需重新编译 hello.exe，直接运行，就能发现显示的内容已经不同了：从显示 “3” 到显示 “103”。

```
> .\hello.exe  
> Hello, World! 103
```

动态链接库的 dll 文件，作为一种可以拷贝就能用的程序模块，可以被多个不同的 exe 程序所共用，因此操作系统的很多功能，都是通过 dll 来提供的。另外，我们的程序如果需要更新某些功能，也可以让用户下载并覆盖那些有修改的 dll 文件，就能拥有新的功能。

大家在电脑使用中，是不是也有碰到过 dll 文件找不到的错误呢？其实这就是有些软件开发者，认为使用者的电脑中“应该”有，而实际上使用者并没有这些库文件而导致的。以前我最常见的是 DirectX 组件找不到的问题，譬如 DDraw.dll 找不到。DirectX 是游戏开发者常用的库，因此想要运行使用了这些库的游戏，就必须要有这批 dll 文件才行。而且很多不同的游戏可能会用到同一套 DirectX 库文件，所以没必要每个游戏都复制一份这些 dll，让玩家去下载安装一次就好了。

在上面的例子中，我们需要在 hello.c 里面，写上 add.c 里面定义的 add() 函数的声明，才能编译通过。写 hello.c 的人，在没有 add.c 的源代码的情况下，其实是可以链接 add.dll/add.lib 的。但是实际

上，`add()` 函数和 `hello.c` 往往是不同的人开发的。`hello.c` 的开发者，在没有 `add.c` 源码，要怎么知道 `add()` 函数该如何写声明呢？

所以一般来说，编写 `add()` 函数和库的作者，除了编译好 `add.dll/add.lib` 以外，还会编写一份名字为 `add.h` 的“头文件”，附带在库文件 `add.dll/add.lib` 上，一起提供给使用者。而使用者只需要在 `hello.c` 里面，写上 `#include "add.h"` 就可以代替具体的 `add()` 函数声明了。

```
#ifndef ADD_H_
#define ADD_H_

__declspec(dllexport) int add(int a, int b);

#endif /* ADD_H_ */
```

这样的头文件，相当于函数库的“使用手册”，帮使用者对库里面的函数进行调用。我们在最早的 `HelloWorld` 程序中，看到的 `#include <stdio.h>` 中的 `stdio.h`，其实也是“标准库”提供给我们的头文件，方便我们能使用操作系统的标准库的。

```
#include <stdio.h>
#include "add.h"

int main()
{
    int c = add(1, 2);
    printf("Hello, World! %d\n", c);
}
```

在 `add.h` 的内容中，除了函数声明，还有 3 行其他的代码：`#ifndef #define #endif`。它们并不属于严格意义上的 C 语言代码，而是“预处理”的命令，作用是防止 `.h` 文件被多次 `#include` 后发生“重复声明”的编译错误。这种特点体现了 C 语言，在模块管理的设计上确实考虑的不多，需要开发者配合编译器，通过“额外补丁”的方法来解决问题。在后续的大多数语言中，都不需要这些东西了。现在的 C++20 方案中，就加入了模块管理的功能，也是可以告别预处理指令了。

C 语言的函数，不仅仅是编程语言角度上一个可以复用的代码块，而且是最基本的链接单元。操作系统和编译器，使用函数的声明，作为不同的库文件进行链接的接口格式。正是因为大量的操作系统功能，都以 C 函数库的规则，来供开发者链接，所以直接写 C 语言代码来使用操作系统，反而是最方便的做法。网上很多开发者宣称 C 语言适合做“底层开发”，这种和操作系统的适应性，就是其中的理由之一。

通过 C 语言入门计算机科学：变量类型 | 机核 GCORES

C 语言借鉴了数学符号，提供了操作内存一套简洁的方法；也奠定了用数字表达整数、浮点数、字符的计算机方案。这些设计直接影响了后来大多数的其他编程语言。

所有的 C 语言教程，都会介绍这种语言的各种变量类型，譬如 `char` 用来放字符，`int` 用来放整数，`float` 用来放浮点数等等，后面还会说指针也是一种类型。但是在我看来，C 语言的变量，其实只有以下简单的含义：

每个变量，都代表了计算机的一块内存。

不管变量是什么类型的，内存里面放的都是数字

表达式里的变量名，表示读出这块内存的内容

通过 `=` 操作符把数字写入内存

变量的类型，代表了对应的运算规则

变量类型决定了变量对应的内存的长度

运算符根据类型进行不同规则的数学计算

C 语言里，一切内容都是数字；一切操作都是数学计算

`char` 类型，是 C 语言最常用的变量类型。因为一个 `char` 类型的变量，就是一个字节（Byte）的内存。一个字节由 8 个二进制数组成，可以存放的正整数范围是 0~255。虽然这个类型，从名字上看，应该是用来存一个“字符”的，但实际上只是存的一个数字而已。我们可以写个程序来理解下：

```
#include <stdio.h>
int main()
{
    char a = 'z';
    char b = 'A';
    char c = a - b;
    printf("a: %c-%d, b: %c-%d, c: %c-%d", a,a, b,b, c,c);
}
```

这个程序编译运行之后，显示的结果是 a: z-122, b: A-65, c: 9-57

看这个代码之前，我先介绍一下 `printf()` 的用法：

`printf()` 的第一个参数，表示要显示一个什么格式的字符串。其中由变量替换的地方，用 `'%'` 跟一个字母代表

`printf()` 的第二个，以及后续的参数，会按顺序，根据第一个参数字符串中的 `'%'` 定位符，依次把内容显示出去

这里用的 `'%c'` 表示，把对应的变量，以字符的方式显示。而 `'%d'` 表示，把对应的变量，以十进制数字的方式显示。

```
char a = 'z';  
char b = 'A';
```

回到代码，我们定义了两个字符类型的变量 `a` 和 `b`，里面分别存了小写字符 `'z'` 和大写字符 `'A'`。实际上，计算机分别找了两块内存，各自都是 1 个字节的，然后在里面分别写了数字 122 和数字 65。为什么是这两个数字呢？因为 ASCII 编码表，规定了 122 代表 `z`，而 65 代表 `A`。

ASCII 全称为 American Standard Code for Information Interchange，就是“美国信息交换标准代码”，最早规定了怎么用数字来表示英文字符的一套编码表

然后我们就可以用数学运算来对这两个变量进行运算。`char c = a - b;` 这行代码，用了减法这个数学运算。从类型“字符”来看，是不是很奇怪呢？两个字符居然可以相减！实际上 C 语言只是把存在变量 `a, b` 里面的数字相减了而已。减完的结果就是 57，存到变量 `c` 里面了。而 57 这个数字，在 ASCII 编码表里面，对应的是字符 9，所以按字符打印的时候，就打印出来一个“9”了。回顾开头提到的“变量”的含义，是不是表明了，所有存在变量里的，不管是什么类型，其实都是数字呢？而且我们用 `=` 等号操作符进行了写入，用 `-` 减号操作符进行了数学计算。

这个例子，粗粗一看，似乎没有什么实际含义，但实际上，不管是图形还是声音，对于计算机来说都是数字，而计算机程序要做的绝大多数操作，就是把这些数字在不同的内存之间搬过来搬过去。譬如显示一个照片，就是把图片文件的数字，读取到内存里，然后再把这些内存里的数字，按照图形压缩的计算方法，进行加加减减的数学运算，解压成可以直接显示的图形数据（也是一堆数字），然后送到显卡的内存上。所以即便是最复杂的程序，都离不开这个例子中的做法：用变量来存放数字，用运算符来计算数字。

`int` 这种“用来表示整数”的类型，其变量的长度，一般是 4 个字节。刚刚我们讨论过的 `char` 是 1 个字节，那么，1 个 `int` 变量，能不能和 4 个 `char` 变量等同起来使用呢？完全是可以的！我们可以写一个下面的程序来说明：

```
#include <stdio.h>  
  
int main()
```

```

{
char s[4] = {0, 0, 0, 0};
int i = 1145258561;
char *p = (char *)&i;
s[0] = p[0];
s[1] = p[1];
s[2] = p[2];
s[3] = p[3];
printf("%c %c %c %c\n", s[0], s[1], s[2], s[3]);
}

```

这个程序运行后显示：A B C D

这里的第一行 `char s[4] = {0,0,0,0};` 定义了一个叫 `s` 的字符数组，意思就是连续 4 个字符类型变量。数组可以用下标，来获得对应位置的变量内容，如 `s[1]` 表示数组里面顺序第二的变量。（经典笑话：程序员数数从 0 开始数）这里的 `s` 字符数组，一开始里面四个字符变量放的都是数字 0。

第二行 `int i = 1145258561;` 定义了一个 `int` 整数类型的变量，然后写了一个很大的数字进去。为什么是这个数字呢？这个数字不会乱写的。后面会解释。

第三行 `char *p = (char *)&i;` 定义了一个字符指针 `p`，然后把 `i` 的指针，强行转换并赋值给了 `p`。这里虽然有点绕，但简单的理解，就是把变量 `i`，强行变成一个字符数组 `p` 来使用。这里的 `&i` 表示读取变量 `i` 的内存地址，这个地址也是一个数字，然后以 `char *` 字符指针的形式，复制写入变量 `p` 的内存里去。

后面四行比较简单，就好像两个字符数组变量的复制，把 `p` 这个数组的元素，依次 1、2、3、4 的写入 `s` 数组中去。实际上就是把整数变量 `i` 的内容，按照内存顺序，一个个的放到四个字符变量中去。

最后打印了四个字符变量的内容，正好是 "A B C D"，因为整数 1145258561 放在四个字节长度的内存里面，刚好是：“65 66 67 68”，而这四个字节内存里面的数字，就是 ASCII 编码表里面的 "A B C D" 的对应编码

整数 1145258561 在内存中存放的一个是 32 位二进制数 01000100 01000011 01000010 01000001，我们按照这个二进制数，分别去看四块一个字节的内存，里面就是十进制数 68 67 66 65，如果写成 16 进制，则是 44 43 42 41

对于 C 语言来说，内存中的数字到底是什么含义，完全可以让程序员去决定。一个 `int` 变量完全可以被当作 4 个 `char` 变量进行操作，反过来也是可以的。我们需要关注的是：哪种类型的变量，以及这种变量的运算符，能最快的满足我们自己的需求。在实际开发中，经常会用 `char` 数组，来代表对“任何数据”的内存进行处理。因为 `char` 的长度是 1 个字节，非常便于我们对于内存中的数据进行自己的处理和拷贝。

对于整数 `int` 和字符 `char` 来说，一般的运算符如加减乘除，计算规则其实都一样的。`char` 你可以认为

是一个字节的整数。但是浮点数 `float` 类型的数字，在内存中的存放是比较复杂的。如果你使用加法 + 加上 1 进行运算，并不是和整数一样处理：最低位反转一下。我们可以写一段程序来窥探一下，`float` 类型变量的数学运算，反应在内存上到底是什么一个样子。

```
#include <stdio.h>

int main()
{
    float f = 3.1415926535;
    char *p = (char *)&f;
    printf("before: %u %u %u %u\n", p[3], p[2], p[1], p[0]);

    f = f + 1;
    printf("after: %u %u %u %u\n", p[3], p[2], p[1], p[0]);
}
```

输出内容为： before: 64 73 15 4294967259 after: 64 4294967172 4294967175 4294967278

上面的程序，先建立了一个 `float` 浮点数变量 `f`，初始化给的内容是圆周率（一部分），是个小数。然后让这个数字加上了整数 1。代码通过 `char` 数组的方式，把前后两次 `f` 变量内存里面的内容打印了出来。

我们根据 +1 前后的内存情况看，根本看不出有任何规律。为啥这个变量只是相差了整数 1，内存里面就差别那么大呢？原因就是浮点数，在内存中的表示，是根据一种特殊的规则来存放的，这套规则的名字叫 **IEEE-754**。这套规则比整数存放在内存里的规则可复杂多了。所以即便只是相差了整数 1，在内存里面的记录的信息可以相差非常远。有兴趣的读者可以专门去网上搜索“浮点数 存储”一类的文章了解。

同样是数学的加法，C 语言在实现整数加法，和浮点数加法上，其处理方法都是很不一样的。我们也不能简单的认为：C 语言就是提供了一种“像数学符号”的工具来操作内存。起码对于 `float` 类型变量来说，C 语言还是帮程序员们做了一些很复杂的事情。

我们没有讨论另外一个重要部分，就是“布尔运算”，或者叫“逻辑运算”。在 C 语言中，数字 0 代表了 `false`，所有其他数字都代表了 `true`。`if` 关键字基本上就是一个“判断是否 0”的选择器，这部分比较接近日常的“是、否”逻辑，所以没有专门提及。

C 语言提供了一套和数学很像的操作符，给程序员们去操作计算机的内存，譬如 `=` 等号表示写入内存，`+-*/` 等数学运算符，会根据变量类型进行运算。变量就代表了内存，变量类型就是内存的长度，内存里面存的就是数字。C 语言允许程序员以任何自己喜欢的方法，去读写这些数字。这一套处理数字的办法，不但是 C 语言，很多后来发明的计算机语言，基本都沿用了 C 语言对于整数、浮点数的处理方法。这套操作内存和数字运算的方法，基本上已经可以完成一切计算机要处理的任务了。除此之外，C 语言已经没有更多其他的“功能”或者内容了——真的是非常的简洁！

有了这一篇的基本认识，下一篇可以真正的开始讲 C 语言最有特色的内容：指针

通过 C 语言入门计算机科学 4：数组和指针 | 机核 GCORES

指针，是 C 语言中最 “著名” 的东西。有人说是最复杂的最难学的部分，也有人说是最危险但是最强大的部分。

我个人认为，C 语言指针的难以掌握的原因有以下几个：

指针的声明符号——* 星号同时也是 “解引用” 操作符。在不同的语句中的含义是不一样的。

指针是一种复合类型，它需要指定自己是属于 “何种变量” 的指针

指针是可以运算的，其数值的变化和指针类型有关。

程序员可以使用指针读写任何地址的内存，如果不了解程序对于内存的运作机制，可能会访问了一些无法预知结果的地址，造成各种进程退出导致的故障

```
#include <stdio.h>

int main()
{
    int i = 10;
    int *p;
    p = &i;
    printf("var i addr: %llu, *p content: %d\n", p, *p);

    *p = 20;
    printf("var i now: %d\n", i);
}
```

var i addr: 272537222788, *p content: 10 var i now: 20

指针这种类型的变量，含义是 “指向某个变量地址的变量”，也就是说，指针变量的内容，是另外一个变量的内存地址。内存地址也是一个数字，在 64 位操作系统中，长度是 8 个字节。

注意指针是一个复合类型变量，也就是说，在声明的时候，除了 * 号外，必须在前面带上一个基本变量的类型，表示指针 “指向” 的变量是什么类型的。你可以选择 char\int\float 等，如果有一些情况下无法确定 “指针的类型”，可以用 void * 来声明一个 “通用类型” 的指针。

在上面的代码里，int *p; 声明了一个 “int 类型的指针” p 变量。而 p = &i 的功能是，把变量 i 的内存

地址的值，写入到变量 `p` 里面去。这里的 `&` 符号的功能是“取地址操作符”，意思就是把某个变量的地址读取出来。

在后面的代码 `printf()` 代码中，我们尝试打印了指针变量 `p` 和 `*p` 的内容：

我用了 `%llu` 作为占位符打印 `p` 的内容，因为 `p` 作为指针变量，长度是 64 位（8 个字节），比之前我们用的 `int` 类型变量的 4 个字节要大，所以使用了 `%llu` 意思是按 `long long unsigned` 长无符号整数（8 个字节）的格式来打印 `p` 的内容。`clang` 编译的时候，会警告我们：变量 `p` 是一个指针类型变量，不是一个“长无符号整数”类型，怀疑我们可以写错了。另外一种常见的打印指针的占位符是 `%p`，如果你用 `%p` 来打印指针，`clang` 就不会抱怨了。

我也用了 `%d` 来打印 `*p` 的值。在这里 `*` 星号的作用是“解引用操作符”，功能是把后面的 `p` 的值，作为一个内存地址，然后去读取此内存地址中的值。由于 `p` 是“`int` 类型的指针”，所以 `*p` 的计算结果，就是一个 `int` 类型的值。于是可以用 `%d` 来打印。

解引用操作符 `*` 是操作指针指向内存的内容的符号，不但可以用来读取指定内存的值，同样可以用来把数据写入这块内存。后面的代码 `*p = 20;` 这一行，就是对 `p` 指向的内存，写入了数值 20。最随后的 `printf()` 代码中，我们可以看到变量 `i` 的值确实被改成了 20。

`&`：取地址操作符，表示取一个变量的地址。我们往往把这个操作的结果赋值给一个指针变量。

`*`：解引用操作符，表示获取一个指针（地址）所指向的内存的变量。我们常常用来读取指针指向变量的值，或者修改这个变量的值。需要注意的是：`*` 在变量声明的语句中表示“指针类型”而不是“解引用”的意思。

如果我们只是用 `&` 和 `*` 来操作一个变量的指针，那么指针本身作用和变量没多少差别，看起来也没什么用处。但是，如果我们需要处理的不是一个变量，而是“一批”变量，指针就是一个很有用的工具了。而所谓的“一批变量”，最常见的就是字符串了。

在 C 语言当中，是没有“字符串”这个变量类型的。而绝大多数的其他编程语言，都有“字符串”类型，为什么 C 语言没有呢？我认为最基本的原因是：

C 语言的变量类型代表了一个特定的内存长度，字符串的长度是不固定的，所以没法给“字符串类型”一个长度设定。

那么 C 语言要处理的最常见的文件应该如何办呢？最简单的办法就是使用“数组”。所谓“数组”，其实就是一批变量，这些变量一般是连续存在的，也就是说在内存里面互相挨着的。我们要存放一段字符串“abc”的话，就声明三个 `char` 类型变量组成的数组，依次在数组中放入'a' 'b' 'c' 三个值就好了。C 语言为数组提供了“下标”的方法来代表里面的每一个变量（又叫做元素），`str[0]` 表示第一个变量（元素），`str[1]` 表示第二个变量，依次类推。

然后，用“字符数组”的方式来存放字符串，还有一个问题需要解决：就是“字符串”长度该如何

表示？由于我们可以声明一个 10 个元素的字符数组，仅仅存放 5 个字符的内容，如果没有办法知道，内容在第 6 个字符处结束，那么可能会让程序处理一些完全不需要的内容。C 语言为了解决这个问题，设计了一个直观但又坑爹的方法：

把字符变量中的数字 0，认为是字符串结束的符号。数字 0 在 ASCII 编码表当中是 NUL 符号，被认为是“没有字符”的意思。

比如我们有一个字符数组 `str[10]`，长度是 10 个字节，我们如果想存放字符串 "abc"，那么我们需要把 `str[0]` 写入 'a'，`str[1]` 写入 'b'，`str[2]` 写入 'c'，`str[4]` 写入 0。这样我们从字符数组 `str` 的下标 0 开始读内容，一直读到第一个内容为 0 的元素为止，就是这个字符串的全部内容了，如果数组第 4 个元素以后还有内容，也被看成是没有意义的数，直接被忽略掉。`printf()` 的占位符 `%s` 就是根据这个规矩进行工作的。下面这个例子就说明 C 字符串是怎么使用的，所以程序中打印的结果，只有“abc”，而后面的“defghi”都不会显示：

```
#include <stdio.h>

int main()
{
    char str[10] = {'a', 'b', 'c', 0, 'd', 'e', 'f', 'g', 'h', 'i'};
    printf("str: %s ptr: %p\n", str, str);
}
```

运行结果：str: abc ptr: 0000004C584FFECE

上面说了 C 字符串的方案，但是和“指针”又有什么关系呢？是不是作者逐渐忘记了标题？——并不是，在上面的 `printf()` 代码中，有一个 `%p` 的占位符，打印的是字符数组变量 `str`，可以看到打印出来一个地址信息。实际上一个数组变量，如果不带下标的话，就是这个数组的第一个元素的指针。

之所以 C 语言如此设计，正是因为指针往往用来操作一个数组。如果我们有两块代码，都需要使用同一个数组，那么使用这个数组的第一个元素的地址，比起把数组复制来复制去要效率搞的多。下面我们可以看一下数组的写法，和指针的写法是如何对应的。

```
#include <stdio.h>

int main()
{
    char *str = "abc";
    printf("str: %s ptr: %p\n", str, str);
    for (int i = 0; i < 4; i++)
    {
        printf("str[%d]: %c | %c\n", i, str[i], *(str+i));
    }
}
```

```
str: abc ptr: 00007FF78EC07320
str[0]: a | a
str[1]: b | b
str[2]: c | c
```

```
str[3]: |
```

这段程序的 `char* str = "abc";` 这一行，其中 `"abc"` 定义了一个字符数组，内容是 `{'a', 'b', 'c', 0}`，而 `char* str` 的指针内容，指向了 `"abc"` 这个字符数组的第一个元素。

程序后面的 `for` 循环，运行了四次，每次都通过下标操作，显示了 `str[0]...str[1]` 的内容，这里可以看到，就算 `str` 不是一个字符数组变量，但还是能通过下标 `[i]` 的写法来读取数组的元素内容。

而后面的 `*(str+i)` 这段，表示的是：用 `str` 的值（地址）加上 `i`，这样就指向了数组第 `i` 个元素的内存，然后用 `*` 解引用操作符读取了对应内存的值。上面的这个过程，其实和用下标的写法 `str[i]` 是等价的。C 语言中的所有数组的功能，是可以使用指针来完成所有操作的。

C 语言的变量就是一块内存，其实数组也是一块内存，数组的长度表示这块内存可以分成多少段（元素）来访问，数组的类型表示每段（元素）的长度是多少。

通过 C 语言入门计算机科学 5：指针运算 | 机核 GCORES

指针这种类型的变量，其数学运算符有自己的特别规定。

在上面的例子中，我们处理的数据都是一个定义好了的变量内容。一般我们实际工作中，往往是处理一些会变化的数据。譬如处理用户的输入、数据文件内容等等。在处理这种变化的数据时，指针成为非常关键的工具。下面举一个例子，用来处理“命令行”参数参数。

命令行参数：在以命令行启动的程序里，是可以读取到启动程序的后续输入参数的。譬如我们前面用的编译命令 clang，我们就把要编译的文件名以“命令行参数”的形式输入到程序里面的

为了获得命令行参数，我们可以通过 main() 函数的参数来获得。之前的例子代码中，我们都忽略了命令行参数，所以使用的最简单的函数形式。下面是例子代码：

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("argv: %p\n", argv);
    for (int i = 0; i < argc; i++)
    {
        printf("argv[%d]: %s %p\n", i, *(argv + i), *(argv + i));
    }
}
```

```
> .\arg.exe ab cde fghi
argv: 0000025BE4E27A20
argv[0]: C:\Users\Admin\Documents\learningC\arg.exe 0000025BE4E27A48
argv[1]: ab 0000025BE4E27A73
argv[2]: cde 0000025BE4E27A76
argv[3]: fghi 0000025BE4E27A7A
```

上面的代码中，我们修改了 main() 的参数，其中 int argc 表示会有几个参数，char **argv 指针参数指向了存放所有命令行参数的内存。然后把命令行参数的内容，以及存放这些内容的地址（以 16 进制数）都打印了出来。

命令行参数的第一个参数是启动程序自己，所以 argc 最少是 1

存放命令行参数的内存块，需要存放多个字符串，所以需要多个字符指针，而所有的这些字符指针，

又需要按照参数的顺序存放起来，所以就需要一个“指向字符指针的指针”变量。——有人会把这种关系描述成一个“字符的二维数组”，对于每个字符，我们可以通过 `argv[i][j]` 的方式读取。但是我认为“指针的指针”比较好理解，对于指针来说，它可以被看成“指针的指针的指针……的指针”，这里有多少层嵌套，变量声明的地方就有多个 `*` 号。

内存中指针的情况

上图还有一个特别值得注意的地方，就是每个 `argv[i]` 的值，刚好相差的就是命令行参数的参数加一（末尾有个 `\0`），譬如 `argv[1]` 和 `argv[2]` 这两个指针的内容分别是 `0000025BE4E27A73` 和 `0000025BE4E27A76`，这两个地址相差 3，刚好是 `ab\0` 这个命令行参数 "C 字符串" 的内存长度。也就是说字符串指针每次增加一，指针变量中的数值也增加一，简单的就好像 $1 + 1 = 2$ 一样。不过这可是 `char` 指针的特例。下面我们用一个例子，来看看指针运算中的 $1 + 1 \neq 1$ 。

之前我们用的例子都是字符数组 `char*`，其实任何类型都可以构造数组，譬如 `int`。

```
#include <stdio.h>

int main()
{
    int int_arr[3] = {1, 2, 3};
    int *p = int_arr;

    int *last_p = p;
    unsigned long long offset = 0;

    for (int i = 0; i < 3; i++)
    {
        if (i != 0)
        {
            offset = (unsigned long long)p - (unsigned long long)last_p;
            last_p = p;
        }
        printf("int arr[%d]: %d, addr: %p, offset: %llu\n", i, *p, p, offset);
        p++;
    }
}
```

```
int arr[0]: 1, addr: 0000009F1D6FFB68, offset: 0
int arr[1]: 2, addr: 0000009F1D6FFB6C, offset: 4
int arr[2]: 3, addr: 0000009F1D6FFB70, offset: 4
```

上面这段代码，主要需要注意的是，把 `int_arr` 这个数组的里面，每个元素的地址打印了出来。在上面的代码 `offset = (unsigned long long)p - (unsigned long long)last_p` 中，我强行把 `p` 和 `last_p` 从整数指针类型，转换成无符号整数计算，这样就可以获得其内部数值的真正的差。如果从地址的数字看，每次 `p++` 运行之后，数值实际上是增加了 4，而不是 1。实际上，每个指针变量，如果你对其进行数学运算，譬如加法或者减法，实际上都是以指针的类型，所代表的内存长度，作为“步长”进行运算的，而并不是简单的数学运算。

譬如对于整数指针 `int* p` 来说，每次用 `p = p + 1` 增加的值，实际上增加了 `1*4`，因为 `int` 整数类型的

长度是 4。之所以字符指针每次加一，地址也是加了一，完全是因为字符 `char` 类型的长度是 1 导致的。

减法也是类似的，譬如上面例程中的代码，如果 `offset` 的计算改成：

```
offset = (unsigned long long)(p - last_p);
```

那么 `offset` 的值就会是 1，因为作为两个整数指针变量 `p` 和 `last_p`，两者做减法得出的答案，也是结果的有几个 4 的倍数作为结果的。

上面我们看了 `int` 指针的计算，按 `int` 的长度作为单位。那么如果是 `int**` 这种“指针的指针”类型，计算的长度又是什么呢？其实只要是指针的指针，不管有多少级，不管最后的类型是 `int` 还是 `char`，都按“指针”这种类型算，“指针”的长度统一是 8 个字节（64 位系统）。

前面我们分析过，C 语言的变量是内存，数组就是内存块，指针是操作内存块的工具。所以我们完全可以对内存进行最底层的操作，譬如复制或者改写，而不必真的知道内存里面存的具体是什么。最常见的就是用 `char*` 字节指针来操作一切数据。我们可以做一个内存复制的例程，来表现 C 语言常常怎么处理真实的数据：我们经常需要把文件内容的内存数据复制出来处理，或者把一整段数据写入到网络端口的缓冲区去。

```
#include <stdio.h>

void copy(char *src, char *dst, int len)
{
    for (int i = 0; i < len; i++)
    {
        dst[i] = src[i];
    }
}

int main()
{
    char *str = "abc";
    char dst[4] = {0};
    copy(str, dst, 4);
    printf("dst: %s\n", dst);

    int int_arr[3] = {1, 2, 3};
    int dst_int[3] = {0};
    copy((char *)int_arr, (char *)dst_int, 3 * sizeof(int));
    for (int i = 0; i < 3; i++)
    {
        printf("dst_int[%d]: %d\n", i, dst_int[i]);
    }
}
```

程序里面出现了 `sizeof()` 关键字，这个关键字作用是返回某个变量或者类型的长度。这里的 `sizeof(int)` 返回值为 4

```
dst: abc
```

```
dst_int[0]: 1  
dst_int[1]: 2  
dst_int[2]: 3
```

在这个程序里面，我们写了一个 `copy0` 函数，这个函数以 `char` 数组的形式，逐个字节对目标数组进行赋值。不管我们传入的是字符数组，还是整数数组（需要把指针类型强行转换），这个函数都会严格的把内存的数据复制过去。在很多处理真实数据的程序里，都是这样通过 `char` 指针，一个个字节进行处理的。

上面这个 `copy0` 函数，隐藏着一个经典的 bug，这也是一些公司面试题里面常见的一条。这个 bug 的原因是，如果 `src` 和 `dst` 两个缓冲区的范围有互相覆盖，按照上面的写法，就会有问题了。譬如 `src + 2` 等于 `dst` 的地址的情况下（更接近于把数据往后移动一段举例），复制就会丢失一部分数据。接近上面这个 bug 的思路，就是判断一下如果可能出现两块内存有重叠区域，就选择从尾往头进行复制，而不是从头往尾复制。在标准库里面，`memcpy0` 函数就有缓冲区重叠的问题，但是另外一个 `memmove0` 就没有这个问题。

本篇我们通过命令行参数，处理过一些变长的内存。实际上 C 语言可以支持我们随意创造变长的内容，我们的程序可以在运行时动态的改变要占用的内存大小。下一篇介绍 C 语言如何自由的控制内存。

通过 C 语言入门计算机科学 6：内存管理 | 机核 GCORES

C 语言指针，被认为是强大而又危险的原因，正是因为其可以自由读写内存。计算机程序运行的全部状态，几乎都存放在内存中。越是自由，越容易造成问题。

上一篇我们介绍了指针的运算，这个操作实际上隐藏着相当大的危险性。因为通过指针的运算，你可以让指针指向任何一块内存，然后去读写里面的信息。为了了解这些危险性，我们需要知道 C 语言大概是如何使用内存的。

假如我们需要编写一个 C 语言函数，用来返回一个“字符串”，那么我们应该如何写呢？根据之前的知识，C 字符串就是一个字符数组，我们似乎返回一个字符数组的指针就可以了。

```
#include <stdio.h>
char *test()
{
    char arr[12] = {'a', 'b', 'c', 'd', 'e', 0};
    return arr;
}

int main()
{
    char *p = test();
    printf("%s\n", p);
}
```

```
mem.c:7:12: warning: address of stack memory associated with local variable 'arr' returned [-Wreturn-stack-address]
   7 |     return arr;
     |           ^~~
1 warning generated.
```

abcd,s 薊?

看起来字符串的前 4 个字节的内容是对的，但是后面的内容就看不懂了，显然这么写这个程序是有问题的。我们先来看看那条编译报警信息，它的意思是：在代码的第 7 行：返回了栈 (stack) 内存的地址，这个地址是一个局部变量 arr 的。这个“栈内存”又是啥意思呢？为了知道这里的意思，我们需要了解一下 C 语言在调用函数时，是怎么使用内存的。

C 语言在每个函数开始调用时，都会分配一块内存，用于存放函数的输入参数、返回值、程序返回指令地址（危险）、函数内局部变量等。函数里面调用新的函数，内存会“向下”新增一块区域，放上被调用的函数参数、返回值、返回指令地址、局部变量。随着函数的层层嵌套调用，这个内存块的使

用会持续的“往下”延申使用。

当一个函数调用结束，进行返回的时候，此函数所使用的内存块会被“释放”。上层调用者函数如果再次调用另外一个函数，刚才被“释放”的这块内存又会被新的函数所使用。——这样一块内存，随着函数的调用、返回而不断变化使用长度，就被称为“栈内存”（stack memory）。存放在“栈内存”中的数据，随着程序运行的情况，会不断的变化。同一个地址的内存，可能在不同时候，被不同的函数写上各自的内容。这就是上面那个例程出问题的原因。

函数调用过程中内存的变化

当 main() 调用 test() 的时候，计算机分配了一块内存供 test() 使用，test() 的局部变量 arr 数组也存放在其中。当 test() 函数完成运行返回时，这块内存就被计算机“回收”了，准备用作其他的功能。然而，曾经使用这块已经被回收的内存的变量 arr 的地址，却被 main() 的 p 指针变量记录了起来。当 main() 函数调用 printf() 函数时，之前被 test() 使用的内存，又被分配给 printf() 使用。而 printf() 在运行的时候，并不会在意之前 test() 的 arr 变量用了什么位置，直接就按自己的需要写入这块内存。因此 arr 曾经使用的内存，从第 5 个字节处开始，刚好被 printf() 给“覆盖”了，写上了他自己的数据（我们光从代码看，并没法预测是什么内容）。因此我们看到，arr 变量的内容，就从第 5 个字节开始变得奇怪了。

由于函数的局部变量所在的内存区域，是会随着函数的调用和返回被分配和回收，进行重复使用的，所以记录下这块内存的地址，留着在上层调用者函数中使用，就是一种明显的“错误”，因此编译器也会“警告”我们。

那么，问题来了，如果我要写一个函数，返回一个数组的内容，到底应该怎么做才是对的呢？答案有两个：

使用输出参数

使用堆

上面说的第一种方法“使用输出参数”，其原理就是：在调用函数前，先把需要返回的数据所存放的内存准备好，然后把这块内存指针传入函数，让函数把内容写到这个指针指向的内存上。这有点像我们去食堂打饭，需要先拿个盘子，让服务员把食物放到我们选的盘子里，然后我们再从这个盘子里面拿食物。代码的示例如下：

```
#include <stdio.h>

void test(char *output)
{
    char arr[12] = {'a', 'b', 'c', 'd', 'e', 0}; // 字符常量
    for (int i = 0; i < 12; i++)
    {
        output[i] = arr[i];
    }
}
```

```
int main()
{
    char p[12] = {0};
    test(p);
    printf("%s\n", p);
}
```

程序运行显示：abcde

在上面的例子中，test() 的参数 char *output 就是所谓的 “输出参数”，而这个指针参数所指向的内存，就是上面举例的那个 “盘子”。在 test() 函数中，我们通过一个循环对输出结果内存进行赋值，当 test() 返回后，main 的 p 数组里面就存好了函数的结果了。然而，上面的这个写法，其实是有一些漏洞甚至安全隐患的，我们稍微修改一下上面的代码。

```
#include <stdio.h>

void test(char *output)
{
    char arr[12] = {'a', 'b', 'c', 'd', 'e', 0}; // 字符常量
    for (int i = 0; i < 12; i++)
    {
        output[i] = arr[i];
    }
}

int main()
{
    char q[12] = {'x', 'y', 'z', 'r', 's', 't', 'u', 'v', 'w', 0};
    char p[3] = {0};
    test(p);
    printf("p: %s\n", p);
    printf("q: %s\n", q);
}
```

```
p: abcde
q: de
```

这个结果比较奇怪的有两个地方，一个是我们的数组 p 命名只有 3 个字节的长度，打印出来却有 6 个字节的内容（abcde\0）；第二个问题是，我们定义的数组 q 本来没有参与 test() 的调用，最后结果从 "xyzrstuvw" 变成了 "de"。造成这个问题的原因，是因为在 test() 函数内部，并没有管输出参数 output，或者叫 p 数组，它的长度是不是有 12 个元素，而是直接去赋值了。当 test() 的 for 循环，运行到 i>2 的时候，其实已经超过了 p 数组的内存限定空间了，这个问题叫做 “指针越界”。在 p 数组限定的长度以外的内存，到底是用来做什么的，一般是难以预料的。在这个例子里，p 数组的内存后面，就是数组 q 的内存，所以这次越界，就把 q 数组的内容给写错了。

指针越界之后的内存情况

从这个例子来看，指针越界造成了数组 q 被意料之外的改写了，这当然会造成程序运行的错误

（bug），但其实指针越界有可能造成更危险的问题：在栈内存中，除了存放了函数中用到的局部变量以外，还有很多其他的重要数据，包括当前函数返回之后，应该执行的程序的地址信息。如果有个程序，会从网络或者其他外部输入数据，这些数据又被写入内存的时候越界了，那么这些数据就有可能被写入栈内存里存放了程序执行地址的内存。这样就提供了一个“漏洞”，让恶意使用者，通过构造专门的输入数据，改写程序执行地址——当前函数返回后，就会去执行这个恶意写入的程序地址，从而让计算机去执行开发者预料之外的代码。这种内存越界的 bug，也是很多程序的安全漏洞的原因。具体更多关于这类漏洞的知识，本文不作过多介绍，只是告诉大家这类漏洞非常容易出现。

对数组 arr 写越界可能造成安全漏洞

最后，关于输出参数的正确写法，应该是如下例程：对于输出参数，除了有指针外，还需要再输入一个参数来表示输出参数内存长度。并且用返回值来告诉调用者，输出的内容的具体长度，避免调用者错误使用。

```
#include <stdio.h>

int test(char *output, int len)
{
    char arr[12] = {'a', 'b', 'c', 'd', 'e', 0}; // 字符常量
    int size = len > 12 ? 12 : len;           // 计算应该返回的长度
    for (int i = 0; i < size; i++)
    {
        output[i] = arr[i];
    }
    return size;
}

int main()
{
    char q[12] = {'x', 'y', 'z', 'r', 's', 't', 'u', 'v', 'w', 0};
    char p[3] = {0};
    int len = test(p, 3);
    printf("p: %.*s\n", len, p);
    printf("q: %s\n", q);
}
```

```
p: abc
q: xyzrstuvw
```

上面的程序中 test() 增加了参数 len 和返回值 int，在循环赋值之前，先判断了一下输出参数所指向的内存（缓冲区）的长度，如果小于返回数值的长度，就最多写满这个缓冲区的长度，不会越界写到其他地方去，然后返回写了的数据长度给调用者。

在 main() 中，我们把输出参数内存的长度 3 传入 test()，然后从 test() 的返回值获得输出缓冲区中数据的长度，然后用了 printf() 的一个特殊占位符 %.*s 来打印字符串，这个占位符的意思是，根据一个长度和一个指针，来打印最多这个长度的数据，而不是一直读到碰到内存中的字节数字 0 为止。这样就避免了打印超过预定数据长度以外的数据。

输出参数需要在调用前，就要预设返回结果的内存长度，但是如果没法预测，是不是有其他方法也能完成任务呢？也是有的，就是使用堆内存，动态的申请内存使用。

所谓堆 (heap) 内存，就是一类不通过直接写变量名来使用的内存。而是通过一对特殊的函数，来分配和回收的内存。这些内存不会使用上面所说的栈 (stack) 内存的空间。同样是返回一个字符数组，我们可以用堆内存如下实现：

```
#include <stdio.h>
#include <stdlib.h>

char *test()
{
    char arr[12] = {'a', 'b', 'c', 'd', 'e', 0}; // 字符常量
    char *output = malloc(12);
    for (int i = 0; i < 12; i++)
    {
        output[i] = arr[i];
    }
    return output;
}

int main()
{
    char *p = test();
    printf("p: %s\n", p);
    free(p);
}
```

程序输出：abcde

在上面的例程中，出现了两个新的函数 `malloc()` 和 `free()`，这两个函数是来自 C 的标准库的，所以需要 `#include <stdlib.h>`。其中 `malloc()` 的参数表示要申请多少个字节的内存，返回值就是申请后的内存指针。注意申请多少就用多少，如果写入数据的时候，超过了申请内存的长度，有可能会把程序里其他变量给写“坏”了。而 `free()` 是释放通过 `malloc()` 申请的内存，传入指针即可，至于要释放的内存有多长，其实是存在申请的内存指针前面的某个位置（头部内存），所以千万要记得释放 `malloc()` 返回的数值，而不是随便在堆内存中选择一个地址用来释放。

在堆内存上分配的空间，C 语言是不会像栈内存一样自动跟随函数的调用来释放的。需要程序员主动调用 `free()` 进行释放，否则程序就会占用内存越来越多，形成了所谓的“内存泄漏”。当然这种问题在实际程序中非常常见，所以我们偶尔也会碰到，一个程序运行的时候，占内存越来越多，电脑越来越卡的情况。同样的，对于释放过的堆内存重复释放，也可能存在问题，通常会导致进程在这里退出（崩溃），或者是导致下一次 `malloc()` 的时候，错误的使用了被其他代码使用的内存而产生问题。所以，确保堆内存的 `malloc()` 和 `free()` 能正确的，一对一的运行，是一个非常考验程序员的需求。这也是后来很多编程语言，都设计了所谓内存自动管理的机制，避免让程序员来处理这个事情。

当我们掌握了指针，了解了栈内存和堆内存，我们才能比较完整的了解 C 语言的能力。下一篇，我们

尝试用之前说的这些能力，做一点真正有用的东西，譬如说用起来没有那么麻烦的“字符串”，而不是以 0 结尾的字符数组。

通过 C 语言入门计算机科学 7: 数据结构 | 机核 GCORES

使用 C 语言编写数据结构是最接近算法原理的手段

当我们掌握了指针，以及栈内存、堆内存的概念，我们就可以用这些工具，来编写一些功能更强大的代码，譬如说“字符串”。前面我们了解的“C 字符串”，实际上只是一块内存，每个字节放一个数字代表字符，数字 0 表示结束。这种表达字符串的方法，在使用上有个缺点：如果要拼接两个或者更多的字符串，我们必须重新申请内存，然后复制内容到新内存上，既消耗内存又浪费时间；同样的，如果要删除字符串其中一部分，或者是在某个位置插入另外一个字符串到本字符串，都是需要重新申请内存然后复制。

所以，我们可以开始自己做一个简单的“字符串”，它要能快速的拼接、插入、删除部分字符串。为了实现这个目标，我们可以用一个叫做“链表”的概念，来完成上述任务。

所谓的链表，就是一条由多个节点构造成的结构。每个“节点”的内存块中，都有一个指针，指向下一个“节点”。如果需要往这个链表中加入一个或者多个节点，只要修改这些节点的指针就可以了。

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    char val;
    struct node *next;
};
```

链表 "Hello, world" 在内存中的情况

这段代码的 `struct node{...}` 表示定义了一个“结构体”，类型叫 `node`，也就是定义了一种新的变量类型 `node`。这种类型的变量，内部包含两个部分的数据，一个是字符，另外一个是指针。代码上可以用 `xx.val` 和 `xx.node` 来使用这里面的数据，如果你获得的是一个 `struct` 的指针，则用 `xx->val` 和 `xx->node` 来使用。在这里，我们会用 `NULL` 这个字眼来代替数字 0，表示指针没有指向有意义的内存，实际上指针内存里面存的就是 0。

1. 在字符串的某个位置插入其他字符串，包含追加和新建字符串

```
// 插入一个字符串到链表中，在第 pos 个位置插入
// 如果 str 为 NULL，则在第 0 个位置插入
// 如果 pos 为 -1，则在最后一个位置插入
// val 是一个 C 字符串，以 \0 结尾
struct node *insert(struct node *str, int pos, char *val)
```

```

{
    // 遍历链表，找到第 pos 个位置
    struct node *p = str;
    int cur = 0;
    while (str != NULL && (pos < 0 || cur < pos) && p->next != NULL)
    {
        p = p->next;
        cur++;
    }

    // 插入字符串
    for (int i = 0; val[i] != 0; i++)
    {
        struct node *new_node = malloc(sizeof(struct node));
        new_node->val = val[i];
        new_node->next = NULL;

        if (p == NULL)
        {
            str = new_node;
        }
        else
        {
            new_node->next = p->next;
            p->next = new_node;
        }

        p = new_node;
    }

    return str;
}

```

插入链表的操作

```

// 删除链表中第 pos 个位置的 len 个字符
struct node *delete(struct node *str, int pos, int len)
{
    if (str == NULL)
        return NULL;

    // 遍历链表，找到第 pos 个位置
    struct node *prev = str;
    struct node *p = str;
    int cur = 0;
    while (cur < pos && p->next != NULL)
    {
        prev = p;
        p = p->next;
        cur++;
    }

    // 删除 len 个字符
    for (int i = 0; i < len; i++)
    {
        if (p == NULL)
            break;

        struct node *next = p->next;
        if (prev == p) // 如果是第一个节点

```

```

    {
        str = next;
        prev = next;
    }
    else
    {
        prev->next = next;
    }
    free(p);
    p = next;
}

return str;
}

```

链表中删除节点

3. 把这个字符串的内容以 C 字符数组的形式返回

// 返回值需要是一个 C 字符串，以 \0 结尾，需要自己释放内存

```

char *get(struct node *str)
{
    if (str == NULL)
        return NULL;

    // 遍历链表，找到看看有多少个字符
    int len = 1;
    struct node *p = str;
    while (p->next != NULL)
    {
        p = p->next;
        len++;
    }

    // 分配内存
    char *output = malloc(len + 1);

    // 遍历链表，复制字符
    p = str;
    int i = 0;
    while (p != NULL)
    {
        output[i] = p->val;
        p = p->next;
        i++;
    }
    output[i] = 0;
    return output;
}

```

```

int main()
{
    struct node *str = insert(NULL, 0, "Hello, world!");
    str = insert(str, -1, " from C.");
    char *output = get(str);
    printf("output: %s\n", output);
    free(output);
}

```



```
str = delete (str, 0, 7);  
output = get(str);  
printf("output: %s\n", output);  
free(output);  
}
```

```
output: Hello, world! from C.
```

```
output: world! from C.
```

上面这段代码，显然距离一个完整的“字符串”功能来说还很不够，而且性能也不是特别好，`get()` 返回堆内存指针本身也不够安全……但是通过这个例子，我们完成了一个最基本，也是最常见的数据结构的雏形：链表。这种通过指针互相关联起来的内存块所构造的数据结构，才是真正解决大量现实问题的常用工具。链表本身也有单向链表、双向链表、数组链表等很多不同的设计，用来解决各种不同的用途。链表以外，还有诸如哈希表、二叉树、跳跃表等等很多数据结构。计算机课程有一门基础课叫《算法导论》或者《数据结构》，讲的就是如何设计各种数据结构和算法，来解决一些常见的问题。对于有指针，可以随意控制内存的 C 语言来说，是更适合构造这些数据结构的语言。

通过 C 语言入门计算机科学 8：文件 | 机核 GCORES

抽象是计算机科学的基本原理

在 windows 上，我们可以看到我们的电脑，是由很多不同图标的设备组成的。每个不同的设备，都有不同的图标，双击后会有不同的菜单。这非常直观。但是作为程序员，我们却希望所有的电脑设备，都能用类似的一种方法来操作，而不希望换一种设备就是一种编程方法。所以，对于 C 语言来说，会把很多设备都抽象为“文件”：键盘是文件、屏幕是文件、打印机是文件、磁盘上的信息也是文件……

windows 上的各种设备

我们可以通过一个简单的程序，来看看键盘和屏幕是怎么被抽象成文件来使用的。

```
#include <stdio.h>

int main()
{
    char str[5] = {0};
    fscanf(stdin, "%s", str);
    fprintf(stdout, "Hello, World!\n%s\n", str);
}
```

这个程序一运行，就会停住，然后显示一个光标，如果你输入了字符 abc 然后回车。你的屏幕上就会显示：

```
Hello, World!
abc
```

在这个程序里，我们看到一个有点熟悉的函数 `fprintf()`，这和之前我们在屏幕上打印数据的 `printf()` 用法几乎一样，只不过需要在第一个参数那里，传入一个 `FILE*` 类型的变量。我们这里传入的是 `stdout`，意思是“标准输出”——一般程序的标准输出，就是屏幕了。

另外，这个程序里面还有一个函数，和 `fprintf()` 是一对儿的，叫 `fscanf()`，这个函数的功能是从文件中，按格式读取内容到变量里面。这个“格式”的写法，和我们之前的 `printf()` 的写法一模一样。`fscanf()` 的第一个参数就是读取的文件，这个文件传入的是 `stdin`，意思是“标准输入”——也就是键盘了。

上面这个程序，编译的时候会有个报警：`fscanf()` 是被弃用的，应该用 `fscanf_s()` 来代替。其原因之前讲过，由于读取文件输入的内容，是可能超过参数 `str` 的内存长度的，那么超过的部分，有可能会覆盖掉栈内存中非常危险的“返回程序地址”部分，从而导致安全漏洞。这种漏洞是黑客们最常见的入侵门路。而 `fscanf_s()` 可以把每个指针变量，都增加一个新的参数，表示最多只读入多少个字节，这样就不

会有覆盖掉危险内存地址的风险了。譬如我们这个程序可以写为：

```
fscanf_s(stdin, "%s", str, 5); // 最多读入 5 个字节的数据到 str 中
```

在一般的 C 程序启动的时候，会自动“打开”三个文件，分别是 `stdin`, `stdout`, `stderr`，这三个文件我们可以直接用。`stderr` 是标准错误的意思，用来输出程序的错误信息的，有些操作系统会收集这个“文件”的内容作为日志。这三个文件，都不是那种常见意义上的磁盘文件，所以你不需要去寻找到底文件名是啥，存在哪里了。在 Linux 上，`stdin/stdout/stderr` 常常被重定向到具体的文件，或者通过“管道”把两个程序的 `stdout` 和 `stdin` 连接起来，这样两个程序就可以直接通信了。以前我写 CGI 程序的时候，就可以通过 Apache 网络服务器的 `stdout`，把用户浏览器发送的 HTTP 请求报文，传给我们自己写的 CGI 程序的 `stdin`；通过我们代码的解析和处理后，把 HTTP 响应报文，譬如 HTML 结果，通过 `stdout` 发到 Apache 进程的 `stdin`，最终通过网络发到用户的浏览器上，从而实现能通过浏览器执行的程序。需要注意的是，如果我们写了一个读取 `stdin` 的带漏洞的程序，而这个程序被用来作为 CGI 进程，那么黑客就有可能通过网络入侵你的服务器了。

相对于 `stdin/stdout`，我们当然可以操作传统的文件。我们可以写个例子实现一个简单的文件复制。

```
#include <stdio.h>

int main()
{
    char *src_file = "a.txt";
    char *dst_file = "b.txt";

    char buf[1024] = {0};
    FILE *sf = fopen(src_file, "r");
    int len = fread(buf, 1, 1024, sf);
    fclose(sf);

    FILE *df = fopen(dst_file, "w");
    len = fwrite(buf, 1, len, df);
    fclose(df);

    printf("Copy file size: %d bytes\n", len);
}
```

然后我们在这个程序的所在目录，存下一个 `a.txt` 文件，里面写上 `Hello, World!`。在运行这个程序之后，屏幕显示：

同时，在同一个目录，会生成一个叫 `b.txt` 的文件，打开之后，里面就是 `Hello, World!`。

上面这个程序，用了一个叫 `fopen()` 的函数用来打开文件，第一个参数是文件路径，第二个参数“读写模式”。这个函数编译的时候会报警说不安全，但是我们用作例子是可以的。我们的程序以“读”模式打开了 `a.txt` 后，把文件内容读出，放到内存（变量 `buf`）中。由于我们设置的 `buf` 数组长度是 1024 个字节，所以这个文件最多只能被拷贝出 1024 个字节的内容。

然后我们用“写”模式打开了 `b.txt` 文件，把 `buf` 的内容写入进去。两个打开的文件指针对象 `sf` `df`，都

需要调用 `fclose()` 来关闭，否则也是一种资源泄漏。操作系统对于一个进程最多打开的文件数量是有限制的，打开太多也会额外消耗内存。所以用完了文件指针，需要关闭掉。

读者可以自己考虑一下改造这个程序，譬如用命令行参数输入要拷贝的文件路径和要生成的文件路径，以及处理超过 1024 字节的内容，让不管多大的文件，都可以被拷贝成功。

当我们使用 C 语言来操作电脑的外部设备，不管是网卡、打印机还是键盘、屏幕，都可以用同一种方式来进行读取、写入，这就是按照文件操作。事实上上面的程序，我们用 `fscanf()`/`fprintf()` 是一样可以完成的，只不过限制了拷贝的文件只能是文本文件而已。把外部设备作为“文件”来处理，不止是 C 语言的风格，其他的编程语言也是类似的。这些文件操作的功能，是操作系统提供的，所以不管什么语言都是这样处理，不过用 C 语言可以非常简单的用 `char` 数组来作为任何数据的缓冲区，对于编写底层的外部设备操作程序，是特别方便，而且性能更好的。

整个系列写到这里终于要收尾了。看着每一篇的阅读量呈等差数列下降，心里还是有点难过的，不过这也是我自己对于这门知识的一次总结。回想起自己学习 C 语言的过程，从第一次拿起教程，到真正的写出能实用的程序，期间足足过去了十年。这十年间，我写了大量的 PHP/JAVA 等各种语言的代码。而 C 语言一直都不能如同这些语言一样容易上手，我觉得最核心的原因有三：

这门语言没有很多功能的标准库，所以看起来好像并不能做什么事情，所以让人学习的兴趣比起其他语言要小的多

由于缺乏模块管理等工程设计，让编译链接过程显得非常复杂，而且 C 语言教程往往又不包含这些，所以真正要实用起来，会碰到大量的问题，显得非常困难

大部分的教程习惯于从抽象的数据类型、数组等角度去介绍语言的功能，没有把“操作内存”作为最核心的用法来介绍，导致指针、堆内存、栈内存非常让人费解，加上动辄 `core dump` 的特性，显得这门语言非常难学。在绕了无数的弯路后，我现在觉得 C 语言并不是一门复杂的语言，而且有很多的语法完全可以用另外一种写法，一样可以成立，譬如数组和指针的操作很多就是等价的。只是规矩越少的东西，灵活性越大，要掌握的好也会显得越困难，如同围棋比象棋更“复杂”一样。因此我希望把这些简单但重要的知识记录下来，以便帮助到愿意踏上这条看似困难的学习之路的同行者。