实验室 1

要获取本实验的启动文件，请在您的目录中运行以下命令`labs`。

```
$ git pull starter master
```

如果您收到如下错误：

```
fatal: 'starter' does not appear to be a git repository
fatal: Could not read from remote repository.
```

确保按如下方式设置启动器遥控器：

```
$ git remote add starter https://github.com/61c-teach/fa20-lab-starter.git
```

并再次运行原始命令。

目标

- TSWBAT（“学生将能够”）**在**EECS 教学计算机上编译并运行 C 程序，并检查**C中不同**类型的控制**流**
- TSWB（“**学生将会**”）介绍 C 调试**器**并获得使用 gdb 调试 C 程序的**实践**经验
- TSWBAT 使用整数、字符、布尔表达式和按位运算符来模拟 C 中的正则表达式。

编译并运行 C

在本实验中，我们将使用命令行程序 gcc 来编译 C 语言程序。运行 gcc 的最简单方法如下：

这会将 program.c 编译为名为 a.out 的可执行文件。如果您学过 CS61B 或有 Java 经验，您可以将 gcc 视为 javac 的 C 版本。此文件可以使用以下命令运行：

可执行文件是`a.out`，那么 到底是干什么`./`用的呢？答案是：当你要执行可执行文件时，需要在前面添加文件路径，以便将你的命令与 python3 之类的命令区分开来。点 ( ) 指的是“当前目录”。顺便说一下，双点 (`..`) 指的是上一级目录。

`gcc`有各种命令行选项，欢迎您探索。不过，在本实验中，我们将仅使用`-o`，它用于指定`gcc`创建的可执行文件的名称。您可以使用以下命令编译`program.c`成名为 program 的程序，然后运行它。如果您不想所有可执行文件都命名为 ，这将非常有用`a.out`。

```
$ gcc -o program program.c
$ ./program
```

可选：C 编译器本地

通过运行安装 Xcode 命令行工具

然后，您可以通过运行以下命令来检查 gcc 是否安装成功

Ubuntu/

`build-essential`通过运行安装包

```
$ sudo apt install build-essential cgdb valgrind
```

这会安装许多软件包，例如`gcc`和`make`，以及`cgdb`和`valgrind`调试工具。如果您感兴趣，这里`build-essential`有一份截至 Ubuntu 18.04 的软件包详细说明。

对于 Windows 用户，如果您不想使用 Hive 并且想要在本地进行开发，我们建议您使用适用于 Linux 的 Windows 子系统。如果您还没有安装，请按照Piazza 上"Windows 用户设置"帖子中的说明进行操作。这些说明的一部分包括在 WSL 子系统中安装 C 编译器。

练习 1：看看你能做

在本练习中，我们将看到一个预处理器宏定义的示例。宏可能是一个比较复杂的话题，但通常情况下，它们的工作方式是：在编译 C 文件之前，所有`define`宏常量的名称都会被替换为它们所引用的值。在本练习中，我们将仅将宏定义用作全局常量。在这里，我们将其定义`CONSTANT_NAME`为引用`literal_value`（一个整型字面量）。请注意，名称和值之间只有一个空格分隔。

```
#define CONSTANT_NAME literal_value
```

现在，查看 中的代码`eccentric.c`。**注意**四个不同的基本 C 语言控制流示例。（想一想：它们是什么？）另外，你认出这些来自伯克利的古怪说法和人物了吗？:)

首先编译并运行程序，看看它的作用。尝试改变四个宏的常量值：V0 到 V3。观察改变**每个**宏的值会如何影响程序的输出。

行动

仅修改这四个值，使程序产生以下输出。

```
Berkeley eccentrics:
====================
Happy Happy Happy
Yoshua
Go BEARS!
```

实际上，有几种不同的宏组合可以产生这样的输出。本练习的挑战是：**考虑 V0 到 V3 中最少需要多少个*不同的*值才能保证它们仍然给出这个精确的输出。例如，当它们彼此不同时，理论上的最大值是 4。**

不知道如何运行程序？重新阅读一下简介。我们希望你将程序编译成一个名为 的可执行文件`eccentric`；你可以使用`-o`标志来执行此操作吗？

练习 2：捕捉那些虫子

顾名思义，调试器是一个专门设计用来帮助你查找代码中的 bug（逻辑错误和失误）的程序（附注：如果你想知道为什么错误被称为 bug，请看这里）。不同的调试器有不同的功能，但所有调试器都具备以下功能：

1. 在程序中设置断点。断点是代码中的特定行，你可以在此处停止程序的执行，以便查看附近发生的情况。
2. 逐行单步执行程序。代码只会逐行执行，但执行速度太快，我们根本无法确定哪些行导致了错误。能够逐行单步执行代码，可以让你准确地找到程序中导致错误的原因。

对于本练习，你会发现GDB 参考卡很有用。GDB 是"GNU De-Bugger"的缩写。`hello.c`使用以下`-g`标志进行编译：

```
$ gcc -g -o hello hello.c
```

这会导致 gcc 将信息存储在可执行程序中，以便`gdb`对其进行解读。现在启动我们的调试器 (c)gdb：

注意这个命令的作用！您正在对 gcc 生成的可执行文件 hello 运行 cgdb 程序。不要尝试对 hello.c 中的源代码运行 cgdb！它不知道该做什么。如果 cgdb 无法运行，您也可以使用 gdb 完成以下练习（使用 启动 gdb `gdb hello`）。cgdb 调试器仅安装在您的 cs61c-xxx 帐户中。请使用 Hive 机器或 27x Soda 中的一台计算机来运行 cgdb，因为我们的 cgdb 版本是为 Ubuntu 构建的。

注意：您可以在本地计算机上安装并运行 (c)gdb，但请注意，它可能无法在（已更新的）macOS 设备上安装。如果您遇到这种情况，可以使用 lldb，它也是另一个很棒的调试器。命令略有不同，但有一些很棒的指南（例如这个！）可以帮助您入门。不过，在本实验中，请使用实验设备以及 cgdb。

行动

通过执行以下操作逐步执行整个程序：

1. 在主程序中设置断点
2. 使用 gdb 的运行命令
3. 使用 gdb 的单步命令

在 gdb 中输入 help 来查找执行这些操作的命令，或者使用参考卡。

如果您看到类似 printf.c 的错误信息，请查看此处：没有该文件或目录。您可能单步执行到了 printf 函数！如果继续单步执行，您会感觉哪儿也去不了！CGDB 会报错，因为您没有定义 printf 的实际文件。这非常烦人。为了摆脱这个黑洞，请使用 finish 命令运行程序，直到当前帧返回（在本例中，直到 printf 执行完成）。**下次**，使用 next 命令跳过使用 printf 的那一行。

注意：cgdb 与

在本练习中，我们使用 cgdb 来调试程序。cgdb 与 gdb 完全相同，但它提供了一些额外的特性，使其在实际使用中更加便捷。参考手册中的所有命令都可以在 gdb 中使用。

在 cgdb 中，你可以按下 键`ESC`进入代码窗口（顶部），`i`按下 键返回命令窗口（底部）——类似于 vim。底部的命令窗口是你输入 gdb 命令的地方。

行动

学习这些命令对本实验的其余部分以及你的 C 编程生涯都将大有裨益。在名为 的文本文件中`gdb.txt`，回答以下问题。

1. 当您处于 gdb 会话中时，如何设置运行时传递给程序的**参数?**
2. 如何**创建断点**?
3. 如何在断点处停止后执行程序中的**下一行 C 代码?**
4. 如果下一行代码是函数调用，那么使用问题 3 的答案会立即执行整个函数调用。（如果不是，请考虑使用其他命令来执行问题 3！）如何告诉 GDB 你**想调试函数内部的代码**（即单步执行函数）？（如果你修改了问题 3 的答案，那么这个答案现在很可能也适用于这里。）
5. 在断点处停止后如何**继续执行程序?**
6. 如何在 gdb 中**打印变量的值**（甚至是像 1+2 这样的表达式）？
7. 如何配置 gdb 以便它**在每一步之后显示变量的值**?
8. 如何显示当前函数中**所有变量及其值的列表?**
9. 如何**退出**gdb?

练习 3：使用 YOU（ser 输入

让我们看看如果你的程序需要用户输入，并且你尝试用 GDB 来处理它，会发生什么。首先，运行由 定义的程序，`interactive_hello.c`与一个过于友好的程序进行对话。

```
$ gcc -g -o int_hello interactive_hello.c
$ ./int_hello
```

现在，我们将尝试调试它（即使实际上没有错误）。

当你尝试运行程序直至完成时会发生什么？我们将学习一个工具来帮助我们避免这种情况。本练习的目的是让你即使程序需要用户输入，也**能无所畏惧地**运行调试器。事实证明，你可以`stdin`直接从命令行向文件流（在这个愚蠢的程序中，由函数 fgets 读取）发送文本，其中包含一些特殊字符。

看看这个网站上的"重定向"部分，看看你能不能想出如何在程序运行时不显式地提供输入（你可能已经意识到，这会让你卡在 CGDB 里）。看看这篇 stackoverflow 帖子，获取更多灵感。

提示1：如果你正在创建一个包含输入的文本文件，那么你走对了路！提示2：记住，你**也可以在CGDB中使用命令行参数（包括重定向符号）来运行程序！**

希望您能理解重定向如何帮助您避免使用 CGDB 时出现的这种令人讨厌的情况。永远不要害怕调试器！我们知道它看起来有点麻烦，但它确实可以帮助您。

练习 4：Valgrind

即使使用调试器，我们也可能无法捕获所有 bug。有些 bug 我们称之为"bohrbug"，这意味着它们会在一组明确定义但可能未知的条件下可靠地出现。其他 bug 我们称之为"heisenbug"，它们并非确定性 bug，而是在人们尝试研究它们时会消失或改变其行为。**我们可以使用调试器检测到第一种 bug，但第二种 bug 可能会被我们忽略，因为它们（至少在 C 语言中）通常是由于内存管理不当造成的。**

请记住，与其他编程语言不同，C 语言要求你（程序员）手动管理内存。本周晚些时候我们会详细介绍这一点，但目前你只需要了解这些。

为了帮助捕获这些"海森堡漏洞"（heisenbug），我们将使用一个名为 Valgrind 的工具。Valgrind 是一个模拟 CPU 并跟踪内存访问的程序。这会降低进程的运行速度（这就是为什么我们不会总是在 Valgrind 中运行所有可执行文件），但也能暴露出一些仅在特定情况下才会显示可见错误行为的漏洞。

注意：Valgrind 可在大多数类 Unix 发行版以及 macOS 上安装，但许多学生报告了 Valgrind 与最新版本的 macOS 之间的兼容性问题。如果您无法在本地安装，我们建议您使用 Hive 机器进行 C 语言开发；这些工具在您完成更多 C 语言作业（项目 1 和 4！）时将非常有用。

在本练习中，我们将演示 Valgrind 输出的两个不同示例，并介绍每个示例的用途。

首先尝试构建两个新的可执行文件，`segfault_ex`from`segfault_ex.c`和`no_segfault_ex`from `no_segfault_ex.c` （使用`-o`之前的标志！）。此时，您应该能够`gcc`成功构建这两个可执行文件。

现在让我们尝试运行可执行文件。你观察到了什么输出？

让我们从……开始吧`segfault_ex`。你应该注意到了分段错误（segfault），它发生在程序因尝试访问不可用的内存而崩溃的时候（本课程后面会详细介绍；这实际上是早期内存设计的一个产物。今天我们使用的是"分页内存"而不是"分段内存"，但错误信息仍然存在！）。

这个 C 文件很小，所以您应该能够打开它并了解导致段错误的原因。现在就打开它，但不要更改文件。为什么会发生段错误？

现在让我们了解一下，如果文件很大，需要查找段错误时该怎么办。这时，Valgrind 就派上用场了。要在 Valgrind 中运行该程序，请使用以下命令：

```
$ valgrind ./segfault_ex
```

这应该会导致 Valgrind 输出非法访问发生的位置。将这些结果与您打开文件时确定的结果进行比较。Valgrind 如何帮助您将来解决段错误？现在尝试在 上运行 Valgrind `no_segfault_ex`。程序应该不会崩溃，但文件仍然存在问题。Valgrind 可以帮助我们找到（看似不可见的）问题。

不幸的是，你会发现 Valgrind 似乎无法准确地告诉你问题出在哪里。你可以使用 Valgrind 提供的消息来**确定哪个变量有未定义的行为，然后尝试推断发生了什么**（提示：什么是未初始化的值？）。

此时，我们并不期望您熟悉`sizeof`（下周就会介绍！），因此我们希望您从本节中获得一些关于问题可能发生位置的直觉。

希望在看完这个例子之后，您能够理解并回答以下问题：

- Valgrind 为何重要以及它有何用处？
- 如何在 Valgrind 中运行程序？
- 您如何解读这些错误信息？**不要害怕。尽力尝试并向我们寻求帮助。**
- 为什么未初始化的变量会导致“海森堡漏洞”？

我们现在向您介绍 Valgrind，因为它是一个非常重要的工具，您一旦开始编写 C 语言就一定会用到它。**然而，要真正理解它，我们需要学习 C 语言的内存模型，我们将在下周进行学习。**在课堂上学习完内存模型后，请回到本实验并尝试回答以下问题：

- 为什么程序**没有出现**`no_segfault_ex`段错误？
- 为什么会`no_segfault_ex`产生不一致的输出？
- 为什么`sizeof`不正确？如何才能继续使用，`sizeof`但又使代码正确？

练习 5：

这是一篇帮助你面试的文章。在 中`ll_cycle.c`，完成以下函数`ll_has_cycle()`，实现以下检查单链表是否存在循环的算法。

1. 从列表头部的两个指针开始。我们将第一个指针称为“乌龟”，第二个指针称为“兔子”。
2. 将兔子前进两个节点。如果由于空指针而无法前进，则表示我们找到了列表的末尾，因此该列表是非循环的。
3. 乌龟前进一个节点。（空指针检查是不必要的。为什么？）
4. 如果乌龟和兔子指向同一个节点，则该链表是循环的。否则，返回步骤 2。

如果要查看`node`结构的定义，请打开`ll_cycle.h`头文件。

行动

实现`ll_has_cycle()`。完成后，您可以执行以下命令来运行代码测试。如果您进行了**任何**更改，请确保按顺序重新运行以下所有命令。

```
$ gcc -g -o test_ll_cycle test_ll_cycle.c ll_cycle.c
$ ./test_ll_cycle
```

提示：学生通常用两种方式编写此函数。它们的区别在于对停止条件的编码方式。如果采用其中一种方式，则需要在一开始就考虑特殊情况。如果采用另一种方式，则需要进行一些额外的 NULL 检查，这没问题。前面两句话旨在提醒你不要过分强调函数的整洁性。如果它们对你没有帮助，请忽略它们。本练习的目的是确保你知道如何使用指针。

这里有一篇关于该算法及其工作原理的维基百科文章。如果你不完全理解，也不用担心。我们不会就此进行测试。

顺便说一下，指针被称为“乌龟”和“兔子”，因为乌龟指针增加得很慢（就像乌龟一样，移动得很慢），而兔子指针增加得很快（速度是乌龟的两倍；就像兔子一样，移动得很快）。

最后要说的是，龟兔赛跑的故事总是适用的，尤其是在 CS61C 中。缓慢而稳定地编写 C 程序，并使用像 CGDB 这样的调试程序，才能让你在比赛中获胜。

请提交**实验室自动评分器**作业（与上周相同！）。

请注意，gdb 答案的自动评分器非常简单。请确保：

1. 文件中的所有行都`gdb.txt`以答案编号开头（例如`1. answer`）。
2. 答案中只需填写命令**名称**`command arg`。因此，如果我们有一个带参数的命令，例如，则只需填写`command`文件即可`gdb.txt`。

为了核对，请确保你准备好回答关于你所做工作的问题。在核对之前，简要回顾一下你修改/编写的内容可能会有所帮助。