Lecture 2

(注：下文中C指代ctrl，F指代fn，W指代windows，A指代alt)

> 这是一个Markdown写成的讲义，你可以将它线上转化为pdf，或者使用vscode的Markdown侧边预览进行查看。如果你想获得更美观的视觉体验还可以试试MarkText

> ChatGPT比你想象的强大。如果你觉得查手册太费时间而你只是稍微好奇某个东西可以先尝试问问AI

课前准备

```
sudo apt install vim gcc valgrind gdb
```

引子

还记得我们第一节课中的提示吗？C本质上是UNIX的副产物。没有UNIX，我们对C的学习与认识就只能限制在一个抽象层上点点点。
C程序是怎么变成可执行文件的，C还有什么不为我们所熟知的点？这些东西不在UNIX上操作是没办法讲清楚的。Let's make our hands dirty, enjoy hacking!

vim

nano并不是一个把快捷键用到炉火纯青的编辑器，还有比它更狠的，那就是它的祖师爷vim
你可以和nano一样，用vim hello.c打开一个文件，或者在这个文件不存在的时候创造这个文件。打开之后，你处于vim的浏览模式

```
命令模式
   ^| esc
  :|V
浏览模式 <-你现在在这儿
   ^| i
esc|V
编辑模式
```

使用i进入编辑模式，键入以下程序：

```c
#include<stdio.h>
int main()
{
    printf("Hello(again), C!\n");
    return 0;
}
```

使用esc回到浏览模式，使用h j k l进行光标的左、上、右、下方向的移动（j像一个向下的箭头，h和l分别在最左和最右，剩下一个k是向上）
在浏览模式，你可以像在man page里一样使用/配合通配符进行查找，这个方法在大文件中很有用
输入:，这时终端的左下角出现了一个冒号等待你，你可以先输入w，回车，这样你的改动就写入了hello.c，再输入:，键入q，这样你就退出了vim
这些简易的vim命令还可以进行组合。比如我可以使用:wq!来进入命令模式(:)，写入文件的改动(w)并退出(q)且使以上命令强制执行(!)
好了，你已经基本学会了vim的使用方法，可以用笨办法和vim进行交互了。你可以在课后空出半个小时，在命令行中键入vimtutor阅读一下，或者打印出vim的快捷键列表放在手头进行参考。

Hello, world的一生

写完了，你应该找一个编译器将hello.c编译为可执行文件。你有两种选择，由GNU维护的gcc和由llvm维护的clang。gcc的使用范围更广一些。

```
gcc hello.c
./a.out
```

当然，你还可以使用-o来重命名输出的可执行文件

```
gcc hello.c -o hello
```

当你的程序出现错误时编译器会在初期进行发现并提示，其中warning不影响编译，但保留warning会导致程序兼容性不佳甚至有逻辑错误，而error会导致程序不通过编译。
gcc可以为你发现语法错误甚至是程序中不严谨的地方，这里涉及到了一些flag：

```
-Wall #一些更加细致的flag的集合，将所有的容易修改的warning全部显示
-Werror #把所有的warning全部转化成error
-Wextra #极其严格，甚至不允许指针和(int) 0进行比较
-w #反其道而行，连warning都不显示了
```

日常我们只要-Wall -Werror就行

```
gcc -Wall -Werror hello.c -o hello
```

不过话说回来，你不好奇从.c到可执行文件这中间发生了什么吗，不是说CPU只认识0和1吗？

预处理

预处理是先于编译的一个步骤，这一步中所有以#开头的预处理语句会进行展开。
我盲猜你对宏没什么好感，你觉得这玩意儿展开太麻烦，还得加一对括号。gcc也不是没意识到这件事，于是提供了一个flag进行宏展开：

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

当我们使用vim打开hello.i时可以发现stdio.h被直接展开塞进了我们原本的C程序。

编译

编译是将C程序转化为汇编程序的过程。

```
gcc -S hello.i -o hello.S
```

一般的，汇编文件习惯使用.S或者.asm后缀进行表示。当你使用vim打开汇编文件时可以发现一些汇编指令，大概长这样：

```
leaq    .LC0(%rip), %rax
movq    %rax, %rdi
call    puts@PLT
movl    $0, %eax
popq    %rbp
```

汇编语言规定了一些段和跳转点，除此之外的每一句汇编指令都对应着硬件层面的一个小操作，比如在寄存器之间挪动值，自增自减，或者是管理栈。

汇编语言本质上是机器语言的助记符号，机器语言是0和1的串。恰当地规划汇编指令可以达到优化程序性能的作用。
gcc提供了一系列优化程序性能的flag，按照优化的激进程度可以分为-O1 -O2 -O3 -Ofast

```
gcc -S hello.c -O1 -o hello_O1.S
gcc -S hello.c -O2 -o hello_O2.S
gcc -S hello.c -O3 -o hello_O3.S
gcc -S hello.c -Ofast -o hello_Ofast.S
vim <asm_file_name>.S
```

或者你也可以通过

```
diff -u <file1_name> <file2_name>
```

进行两个文件之间的内容比较

汇编

汇编文件通过汇编器转换为可重定位目标程序，一般使用.o作为后缀。

```
gcc -c hello.S -o hello.o
```

如果你打开hello.o会发现里面是乱码，但是我们依然有办法大概看看它是什么。
objdump是一个反汇编软件，它可以将机器语言反向转译为汇编语言。-d为反汇编(disassemble)，-S为指定源文件(Source File)：

```
objdump -dS hello.o
```

或者还可以使用hexdump将文件中的数据转化为十六进制数进行显示：

```
hexdump hello.o
```

注意文件本身的表示内容，hexdump hello.S得到的是汇编语言的十六进制ascii，而hexdump hello.o得到的是机器码的十六进制表示
我们还可以尝试修改一下这个文件.....

```
vim hello.o
# 修改可以看清的部分
```

链接

最后，hello.o会和printf.o进行链接，形成可执行文件。特殊的，由于我们的printf.o是标准库的一部分，所以我们可以不加任何flag直接进行使用

```
gcc hello.o -o hello
```

这里便形成了可执行文件。

你好奇printf.o在哪儿吗？大致的方法是读取目标程序的符号表(symbol table)，再把输出进行模式匹配

```
cd /lib/x86_64-linux-gnu
ls libc.so.6
nm -D libc.so.6 | grep printf
```

看起来上面的内容逻辑自洽的不得了，不是吗？然而，我们的讲述某种意义上还是宏观层面的。如果你真的想深入编译器的每一处细节，你可以用这个：

```

```

```
gcc -fdump-tree-all hello.c
```

debug

当你遇到bug的时候你会怎么做，直接盯着看吗？这种debug的方法显然是不科学的。我们使用计算机的初衷就是使用自动化的手段来减轻人类劳动的，但是眼睛直接看的方法显然是加重了人类劳动的。一种最朴素的debug方法是使用`printf`打印临时变量的值，但这种方法建立在一种对"某个地方一定有问题"的怀疑之上。如果你第一反应没感觉到哪儿出问题了，你就应该借助一些更加强大的工具了。

一般的语法错误可以在warning和error中发现，逻辑错误可以通过结果进行倒推。但是如果是段错误呢？C中最致命的错误无疑是段错误(Segmentation Fault)，段错误会在任何非法内存访问时（如野指针，数组越界，二次free等）产生且一般没有有效的提示信息。科学的工具在这时就显得很重要了。

valgrind

你可以直接使用

```
valgrind ./<executable>
```

来检查可执行文件有没有内存泄漏问题，但是它只负责检查不负责改，而且输出的可读性比较差。修改的工作要交给gdb完成。

gdb

在正式使用之前请将目标程序带上 `-g` flag重新编译以产生符号表

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

我们先看第一个示范程序，它计算1到100的和

```
#include<stdio.h>
int main()
{
   int sum;
   for(int i = 1; i <= 100; i++)
      sum += i;
   printf("sum = %d\n", sum);
   return 0;
}
```

但是当我们输入`.sum`时显示了`sum = -192810638`，这显然是不正常的
我们使用

```
gdb ./sum
```

来启动调试器
使用start启动程序，再使用layout src进行源码查看。
在gdb中有大量的指令都有缩写，比如单步执行的step可以缩写为s，单步执行在见到函数的时候会直接进入函数内部。你可以在gdb中敲回车来重复上一条指令
当i = 1时sum理论上是1，但是当我们使用print sum或者p sum来查看变量的值时会发现它是一个垃圾值，初步判断是变量未经初始化就直接使用的问题，调试结束，使用exit或者quit退出gdb。

```
int sum = 0;
```

我们改过来这个程序，然后使用不同等级的优化，再进gdb看看

```
gcc sum.c -g -o sum_no
gcc sum.c -O2 -g sum_2
```

使用gdb打开第一个程序，

```
start
p sum
```

这时候声明语句还没执行，sum是一个垃圾值，但是

```
gdb sum_2
start
s
```

却直接跳进了printf。
开启优化的一个很可能的情况是你的算术运算在编译的过程中被优化为了直接赋值，gdb单步执行的时候把中间的运算步骤全部省略了，然后跳进了printf，这对debug是很不利的。这时候你就明白为什么你使用Visual Studio敲代码的时候很多帖子会建议你使用 debug mode 而不是 release mode ，release mode本质上就是包了一层图形界面的优化选项，你应该在debug完毕之后再使用release mode进行提速

第二个示范程序：

```
#include<stdlib.h>

int main()
{
  int *p = NULL;
```

```
  p++;
  *p = 114514;
  return 0;
}
```

这里的程序在编译后运行直接产生了Segmentation Fault，我们老样子：

```
gdb ./segment
start
layout src
```

使用单步调试，在第七行的时候程序出现了段错误，初步猜测是指针陷阱。
重新使用start开始程序，但是我们在第7行打个断点，然后让程序不间断运行，直到断点处停止

```
break 7
continue
```

我们看看p里头到底发生了什么，鉴于这是一个指针，我们使用十六进制(hexadecimal)表示法进行打印

```
p/x p
p/x *p
```

第一个结果是0x4，第二个结果是一个错误信息，p出现了非法的内存访问，调试结束

第三个示范程序：

```
#include <stdio.h>

int main()
{
    int sum = 0, i = 0;
    char input[5];

    scanf("%s", input);
    for (i = 0; input[i] != '\0'; i++) {
        if (input[i] < '0' || input[i] > '9') {
            printf("Invalid input!\n");
            sum = -1;
            break;
        }
        sum = sum*10 + input[i] - '0';
    }
    printf("input=%d\n", sum);
```

```
    return 0;
}
```

这个示范程序不仅在输入非数字时会报错，还会在接受过长的字符串时报错，我们老样子：

```
gdb ./buffer
start
layout src
```

这里有大量的标准库函数。它们常常涉及到从一些较为底层的操作，如果使用单步执行会浪费大量时间。我们使用next或者n在单步执行的时候将函数当成单个语句进行跳过，当然，如果你不小心使用了单步执行也可以使用finish跳出函数

```
n
#...
# stack smashing detected
# program received SIGABRT
```

程序返回了一个报错，甚至于在调试过程中gdb的显示也出现了一点异常。在网络上查询报错信息得知这是缓冲区溢出(buffer overflow)错误，接下来检查input[]的内容。我们依然输入一个过长的字符串，不过在输入后我们直接查看字符串的内容：

```
enulvrneiava
p input
```

还没意识到问题在哪儿吗？如果没意识到，重新开始程序，输入一个能够容纳进input的字符串，再打印字符串的内容。没错！\0这个表示字符串结束的空字符不见了！
缓冲区溢出是计算机中最为普遍且致命的错误之一。Windows系统的"永恒之蓝"漏洞本质上就是一个缓冲区溢出。试想一下，当示范程序的循环变量并没有找到空字符而使得input[i]按照顺序乱跑会发生什么结果，而如果恰好这个乱跑的路径上包含一个恶意脚本的起始地址呢？
不说别的，我们在练习题中放置了另一个程序，看起来只是将我们的示例程序打包成模块化的函数，但是当这个练习程序发生缓冲区溢出时甚至可以使gdb崩溃。
在日常的编程活动中我们应当留出足够的缓冲区空间保证字符串的正常储存，还应当避免使用不安全的库函数，比如将不进行缓冲区溢出检查的gets()（已于C11标准中被禁用）更换为fgets()。
我们进入最后一个示例程序吧

```
void f1();
void f2();
void f1()
{
   f2();
}
```

```
void f2()
{
   f1();
}

int main()
{
   f1();
   return 0;
}
```

老样子：

```
gdb ./stack
start
layout src
```

直接用n，报段错误，直接用s，发现除了f1()函数调用和f2()函数调用之外没有任何其他的有效信息。但是当我们使用了backtrace进行栈回溯的时候发现栈回溯记录上多了大量且重复的函数名称，合理怀疑是无穷递归导致了栈溢出。
每次函数被调用时会创建一个栈帧(stack frame)，在栈帧中存储函数的返回地址，局部变量等信息。而当函数返回时会跳转进入返回地址，将栈中的信息恢复并销毁这个栈帧。如果函数一直进行递归，那么栈空间不断增长最终会导致栈溢出。这里涉及到了程序的机器级表示，简单总结就是"递归过深会导致栈溢出"。如果你想了解更多关于程序的机器级表示的内容可以参考《深入理解计算机系统》的第三章。

大型工程的构建

你会怎么构建一个大型工程？把所有的函数都写进一个.c里头，占了一千多行，还是#include一个.c文件？那头文件是啥？
Linux其实本质上就是一个大型的C程序，涉及到的源文件就有一万多个。这个超大型的工程又是如何进行组织的呢？

头文件

还记得预处理步骤吗？如果你仔细观察hello.i文件就会发现展开结果只有大量的函数声明。如果将函数的内部结构也直接放入头文件会大大降低预处理的效率。头文件就是为了解决这样一个问题诞生的。

我们需要两个文件，一个是.h，另一个是与它同名的.c。在.h中放入函数的定义，类型别名，结构体，全局变量和一些宏的定义，在.c中#include这个.h并写入这些函数的实现。为了简化问题，我们一个文件中只声明一个函数

```
//gcd.h
int gcd(int a, int b);
```

```c
//gcd.c
#include"gcd.h"
int gcd(int a, int b)
{
   while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

```c
//main.c
#include<stdio.h>
#include"gcd.h"

int main()
{
    printf("gcd 16, 21: %d\n", gcd(16, 21));
    return 0;
}
```

include后头的<>和""分别指代了首先从系统搜索头文件和首先从本文件夹搜索头文件。
还记得链接吗？我们的gcd显然不是标准库的一部分，为了使gcd.o和main.o建立链接，我们的编译指令应该写成：

```
gcc main.c gcd.c -o main
```

那如果我们又写了lcm呢？

```c
//lcm.h
int lcm(int a, int b);
```

```c
//lcm.c
#include"gcd.h"
#include"lcm.h"

int lcm(int a, int b)
{
   return (a * b) / gcd(a, b);
}
```

```
//main.c
...
#include"lcm.h"
...
   printf("lcm 16, 21: %d\n", lcm(16, 21));
   return 0;
```

如法炮制：

```
gcc main.c gcd.c lcm.c -o main
```

这里的包含关系都是一层，但如果是两层呢？两层及以上的时候你的头文件就可能会被多次包含……

```
//score.h
#define N 100

typedef struct {
    int x, y;
} Point;

extern int score;
extern int a;
```

```
//score.c
int score = 85;
int a;
```

```
//gcd.h
#include"score.h"
```

```
//lcm.h
#include"score.h"
```

然后……

```
gcc main.c gcd.c lcm.c score.c -o main
```

报错了！错误原因是score被重复声明了。展开一下可以看出int score = 100在gcd.h 和 lcm.h 中都出现了。显然，我们应该在预处理的时候添加一个flag来避免重复包含。

其实很简单，只要把整个头文件装在#ifndef块中就行了

```
#ifndef __GCD_H__
#define __GCD_H__
//头文件正文
#endif
```

另一个问题，你不觉得刚才的gcc命令很啰嗦吗？对于Linux这种有一万多个源文件的工程来说这种单句的指令是灾难性的！我们需要定义一套规则进行编译。

Makefile

使用vim makefile编写makefile，编写每个源文件生成中间文件时的依赖文件，比如main显然依赖于main.o, gcd.o, lcm.o和score.o的编译产物进行链接：

```
main: main.o gcd.o lcm.o score.o
   gcc main.o gcd.o lcm.o score.o -o main
main.o: main.c gcd.h lcm.h
   gcc -c main.c -o main.o
gcd.o: gcd.c score.h
   gcc -c gcd.c -o gcd.o
lcm.o: lcm.c gcd.h score.h
   gcc -c lcm.c -o lcm.o
score.o: score.c
   gcc -c score.c -o score.o
```

保存之后，你就可以直接使用make进行编译了。（友情提示：makefile只认识tab缩进不认识空格缩进）

你还可以尝试这时候修改某个源文件，然后再make，你的新增内容会被自动补充进编译产物中。
但是这还不够。如果仅仅是上面这样的写法那还不如直接一句gcc呢。make的强大之处在于它某种意义上也是一种编程语言，刚才的makefile就可以改写成这样：

```
CC=gcc
CFLAGS=-std=c11 -w
OBJ=main.o\
    gcd.o\
    lcm.o\
    score.o
HEADER=gcd.h\
       lcm.h\
       score.h

main: $(OBJ)
       $(CC) $(CFLAGS) $(OBJ) -o main
```

```
main.o: $(HEADER)
        $(CC) $(CFLAGS) -c main.c -o main.o
gcd.o: $(HEADER)
        $(CC) $(CFLAGS) -c gcd.c -o gcd.o
lcm.o: $(HEADER)
        $(CC) $(CFLAGS) -c lcm.c -o lcm.o
score.o:
        $(CC) $(CFLAGS) -c score.c -o score.o

.PHONY: clean
clean:
        rm -rf $(OBJ) main
```

开头我们定义了变量，在规则中我们可以使用`$(variable)`的方式将变量的内容进行展开，还有一个伪目标`clean`，你可以使用`make clean`调用它来清理编译产物只留下源码。
想想这样有什么好处——以后我们如果再新添加了什么源文件只要新写一条规则，然后在变量中补充源文件的名称就行了。
关于makefile更详细的讲解可以看看拓展阅读中的《跟我一起写Makefile》进行学习。

尾声

事实上讲C并不是一件容易的事情，gcc的强大还是超乎了我的想象，许多的flag我也是在写讲义的时候才第一次知道。
C的另一块硬骨头是它和系统底层的极其紧密的联系。或者说，C是一门伪装成高级语言的汇编语言。再激进一点儿，没有对计算机系统全方位的认识，你写不出高质量的C代码。关于C，乃至计算机系统的全貌，可以参考拓展阅读中的《深入理解计算机系统》和《Linux C编程一站式学习》两本书。
如果你发现讲义中有什么问题欢迎与我联系，你将有机会列入Contributors列表。两份讲义对应的Github repo将在下课后更改为所有人可见。
祝大家学有所成，也希望大家多多水群，踊跃发言，我们下一次社团公开课再见。

公开课群号：(TODO)
电脑协会2群：760543989

练习题

不止是C

(1)emacs是另一个将快捷键使用到炉火纯青的编辑器，和vim不同，emacs追求极致的扩展性，甚至为了扩展性它的本体就要占据117MB的空间。如果有兴趣可以尝试一下emacs
(2)尝试触发这个程序的缓冲区溢出漏洞：

```
#include <stdio.h>
char input[5];

int sum_all()
{
    int sum = 0, i = 0;
    for (i = 0; input[i] != '\0'; i++) {
```

```
        if (input[i] < '0' || input[i] > '9') {
            printf("Invalid input!\n");
            sum = -1;
            break;
      }
        sum = sum*10 + input[i] - '0';
    }
    return sum;
}

int main()
{
    scanf("%s", input);
    int sum = sum_all();
    printf("input=%d\n", sum);
    return 0;
}
```

你的gdb还好吗？由此请思考这样一个问题：

> 为什么大型工程很少使用文件级全局变量？

(3)展开示例的makefile，和你输入make之后产生的命令比对一下看看你的展开是不是正确的
(?)大型练习：你在大型项目中总有可能会有思路走偏或者bug藏在层层函数和抽象之后的情况。如果有那么一个软件可以帮助你回到最近的没出错的那一刻重新开发，那该多好！基于这个思想，版本管理系统诞生了。查找资料，了解git的使用方法。（我并不想在课上讲解git，很多时候git的命令乃至使用命令的顺序都是高度公式化的，如果我讲的话大概率也和念官方文档差不多）

互联网

(1)Hello和Hello(again)分别致敬了两场著名的科技发布会，你知道它们分别是谁吗？
(2)在*The C Programming Language*中作者使用的编译器叫`cc`，但我们只知道`gcc`，当你在命令行中输入`which cc`和`which gcc`的时候出现了相同的结果。那么这两者之间有什么联系呢？
(3)事实上`gcc`默认使用的C标准并不是我们所熟知的`ANSI C`，而是`GNU C`。在早期出版的*The C Programming Language*中使用的C标准是`K&R C`，这三者谁是谁的拓展，它们又和国际通用的`ISO C`标准之间有什么联系？`gcc`有什么flag可以使得你的代码使用`ANSI C`编译？
(?)你察觉到了吗？我们熟知的计算机世界建立在"绝对可信"的代码之上。我们享受着"绝对可信"的软件并自由地使用它们。然而Linux内核bug光2024年就产生了将近三千个，Windows更新后产生致命问题也不是什么新鲜事。哪怕是gdb，在某些刁钻的bug前还是会破功，出现逃脱了层层错误处理之后还活下来的段错误。你对此有什么感想？

拓展阅读

什么是机器语言？
The C Programming Language 确切来说这本书不只是C，更像是《如何在Ken Thompson出生之前手搓UNIX标准库的一个小部分》
Computer Systems: A Programmer's Perspective 借助C，了解计算机系统的全貌
Linux C一站式学习 一本堪称仁至义尽的中文C教材
跟我一起写Makefile

实用链接

gcc online documentation
gdb documantation
CERT C Coding Standard 怎么编写可靠的C程序？你需要它！