C 语言拾遗 (1): 机制

蒋炎岩

南京大学　　计算机科学与技术系　　计算机软件研究所

本讲概述

在 IDE 里，为什么按一个键，就能编译运行？

- 编译、链接
  - .c → 预编译 → .i → 编译 → .s → 汇编 → .o → 链接 → a.out
- 加载执行
  - `./a.out`

背后是通过调用命令行工具完成的

- RTFM: `gcc --help`; `man gcc`
  - 控制行为的三个选项：`-E`, `-S`, `-c`

本次课程

- 预热：编译、链接、加载到底做了什么？
- RTFSC 时需要关注的 C 语言特性

进入 C 语言之前：预编译

#include <> 指令

以下代码有什么区别？

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

为什么在没有安装库时会发生错误？

```
#include <SDL2/SDL2.h>
```

你可能在书/阅读材料上了解过一些相关的知识

- 但更好的办法是阅读命令的日志
- `gcc --verbose a.c`

有趣的预编译

以下代码会输出什么？

- 为什么？

```c
#include <stdio.h>

int main() {
#if aa == bb
  printf("Yes\n");
#else
  printf("No\n");
#endif
}
```

宏定义与展开

宏展开：通过复制/粘贴改变代码的形态

- `#include` → 粘贴文件
- aa, bb → 粘贴符号

知乎问题：如何搞垮一个 OJ？

```
#define A "aaaaaaaaaa"
#define TEN(A) A A A A A A A A A A
#define B TEN(A)
#define C TEN(B)
#define D TEN(C)
#define E TEN(D)
#define F TEN(E)
#define G TEN(F)
int main() { puts(G); }
```

宏定义与展开

如何躲过 Online Judge 的关键字过滤？

```
#define SYSTEM sys ## tem
```

如何毁掉一个身边的同学？

```
#define true (__LINE__ % 16 != 0)
```

宏定义与展开

```
#define s ((((((((((((((( 0
#define _ * 2)
#define X * 2 + 1)
static unsigned short stopwatch[] = {
  s _ _ _ _ _ X X X X X _ _ _ X X _ ,
  s _ _ _ X X X X X X X X X _ X X X ,
  s _ _ X X X _ _ _ _ _ X X X _ X X ,
  s _ X X _ _ _ _ _ _ _ _ _ X X _ _ ,
  s X X _ _ _ _ _ _ _ _ _ _ X X _ ,
  s X X _ X X X X X _ _ _ _ _ X X _ ,
  s X X _ _ _ _ _ X _ _ _ _ _ X X _ ,
  s X X _ _ _ _ _ X _ _ _ _ _ X X _ ,
  s _ X X _ _ _ _ X _ _ _ _ X X _ _ ,
  s _ _ X X X _ _ _ _ _ X X X _ _ _ ,
  s _ _ _ X X X X X X X X X _ _ _ _ ,
  s _ _ _ _ _ X X X X X _ _ _ _ _ _ , };
```

X-Macros

宏展开：通过复制/粘贴改变代码的形态

- 反复粘贴，直到没有宏可以展开为止

例子：X-macro

```
#define NAMES(X) \
  X(Tom) X(Jerry) X(Tyke) X(Spike)

int main() {
  #define PRINT(x) puts("Hello, " #x "!");
  NAMES(PRINT)
}
```

有趣的预编译

发生在实际编译之前

- 也称为元编程 (meta-programming)
  - gcc 的预处理器同样可以处理汇编代码
  - C++ 中的模板元编程; Rust 的 macros; ...

Pros

- 提供灵活的用法 (X-macros)
- 接近自然语言的写法

Cons

- 破坏可读性 **IOCCC**、程序分析 (补全)、……

```
#define L (
int main L ) { puts L "Hello, World" ); }
```

编译与链接

(先行剧透本学期的主要内容)

编译

一个不带优化的简易 (理想) 编译器

- C 代码的连续一段总能找到对应的一段连续的机器指令
  - 这就是为什么大家会觉得 C 是高级的汇编语言！

```c
int foo(int n) {
  int sum = 0;
  for (int i = 1; i <= n; i++) {
    sum += i;
  }
  return sum;
}
```

链接

将多个二进制目标代码拼接在一起

- C 中称为编译单元 (compilation unit)
- 甚至可以链接 C++, **rust**, ... 代码

```
extern "C" {
  int foo() { return 0; }
}
int bar() { return 0; }
```

加载：进入 C 语言的世界

C 程序执行的两个视角

静态：C 代码的连续一段总能对应到一段连续的机器指令
动态：C 代码执行的状态总能对应到机器的状态

- 源代码视角
  - 函数、变量、指针……
- 机器指令视角
  - 寄存器、内存、地址……

两个视角的共同之处：内存

- 代码、变量 (源代码视角) = 地址 + 长度 (机器指令视角)
- (不太严谨地) 内存 = 代码 + 数据 + 堆栈
  - 因此理解 C 程序执行最重要的就是内存模型

从 main 函数开始执行

标准规定 C 程序从 main 开始执行

- (思考题：谁调用的 main？进程执行的第一条指令是什么？)

```c
int main(int argc, char *argv[]);
```

- argc (argument count): 参数个数
- argv (argument vector): 参数列表 (NULL结束)

上次课已经演示过

- `ls -al` (`argc = 2, argv = ["ls", "-al", NULL]`)

main, argc 和 argv

一切皆可取地址！

```
void printptr(void *p) {
  printf("p = %p;  *p = %016lx\n", p, *(long *)p);
}
int x;
int main(int argc, char *argv[]) {
  printptr(main);  // 代码
  printptr(&main);
  printptr(&x);    // 数据
  printptr(&argc); // 堆栈
  printptr(argv);
  printptr(&argv);
  printptr(argv[0]);
}
```

C Type System

类型：对一段内存的解读方式

- 非常“汇编”——没有 class, polymorphism, type traits, ...
- C 里所有的数据都可以理解成是地址 (指针) + 类型 (对地址的解读)

例子 (是不是感到学了假的 C 语言)

```c
int main(int argc, char *argv[]) {
  int (*f)(int, char *[]) = main;
  if (argc != 0) {
    char ***a = &argv, *first = argv[0], ch = argv[0][0];
    printf("arg = \"%s\";  ch = '%c'\n", first, ch);
    assert(***a == ch);
    f(argc - 1, argv + 1);
  }
}
```

End.

- C 语言简单 (在可控时间成本里可以精通)
- C 语言通用 (大量系统是用 C 语言编写的)
- C 语言实现对底层机器的精确控制 (**鸿蒙**)
- 推荐阅读：**The Art of Readable Code**