

斐波那契数列 in C

```
#include <stdio.h>

int main(void) {
    int x, y, z;

    while (1) {
        x = 0;
        y = 1;
        do {
            printf("%d\n", x);

            z = x + y;
            x = y;
            y = z;
        } while (x < 255);
    }
}
```

简单过一下每次do while的运算结果

x	y	z
0	1	1
1	1	2
1	2	3

利用三个整数进行整体向左平移的丝滑运算，看上去很简洁易懂。

compiler

作者应该用的是macos，正好学学macos的工具

```
% gcc -o fib fib.c
```

编译一个名为fib.c的C源文件，生成一个可执行文件fib

插播一段关于compiler的复习笔记

编译器（compiler）是一种将源代码（source code）转换为机器语言（machine language）或中间代码（intermediate code）的程序

把人类可读的代码，转化为计算机可以直接执行的低级指令

compiler workflow:

1. 词法分析（Lexical Analysis）：
把代码分解成tokens，比如关键字、操作符、变量等
 2. 语法分析（Syntax Analysis）：
检查语法结构正确性
 3. 语义分析（Semantic Analysis）：
确保源码有效，比如变量是否定义，类型是否匹配
 4. 优化（Optimization）：
改进代码的效率，减少执行时间或节省空间
 5. 代码生成（Code Generation）：
生成机器代码或中间代码，以便计算机执行
- **Object Code**: 目标代码，通常是中间形式的代码，可以进一步链接生成可执行文件
 - **Executable File**: 可执行文件，编译后的最终文件，计算机可以直接运行
 - **Linker**: 链接器，把多个目标文件链接为一个可执行文件
 - **Interpreter**: 解释器，与编译器不同的是，解释器逐行读取并执行代码，而不是一次性转换为机器码

实例

假设我们有一个项目叫**计算器**，它包含几个不同的小文件，每个文件完成不同的计算任务：

- add.c: 负责加法计算
- subtract.c: 负责减法计算
- main.c: 负责控制整个计算器的工作流程，比如输入数字、选择计算方式等

首先，我们写的这些.c文件都是我们人类可以理解的代码。接下来，编译器会分别把每个.c文件编译成独立的Object Code文件：

- add.c 编译后生成 add.o
- subtract.c 编译后生成 subtract.o
- main.c 编译后生成 main.o

这些 .o 文件就是编译后的 Object Code 文件。它们已经包含了每个小程序块的机器代码，**但这时还不能直接运行**，因为：

- main.o 可能需要调用 add.o 和 subtract.o 中的函数，但它还不知道这些函数在哪里
- 每个 .o 文件的代码没有被安排到最终的内存地址

这时 Linker 就会出场，把 add.o、subtract.o 和 main.o 文件组合起来，连接所有的符号（比如 main 想调用 add 和 subtract 函数），并重新定位它们的内存地址，最后生成一个完整的可执行文件，比如 calculator.exe。

linker也是由source code决定的吗？

非也，或者说**间接**。（linker是由.o文件决定的，而.o又是由source code决定的。）

回顾：source code会告诉compiler要做哪些事情、定义了哪些function、用了哪些external function，然后compiler根据这些生成各个.o文件。

在这些.o中，包含了很多重要的信息，比如：

- symbol table：列出在source code中定义的functions和variables（比如 add、subtract）
- undefined reference：记录source code中用到的、但在当前文件中没有define的functions和variables（比如 main.o可能引用了add，但它是在 add.o文件中定义的）。

当编译器生成这些.o文件后，Linker负责把它们组合成一个完整的exe。在这个过程中，Linker需要解决几个source code带来的问题，比如：

- 如果 main.c 中调用了 add 函数，但它不在 main.c 中定义，Linker 就会去找到 add.o 中对应的定义，并将它们连接起来。
- source code中的每个functions和variables都会在内存中占用位置，Linker 会根据所有代码块的大小来安排内存地址。

常见编译器

- **GCC (GNU Compiler Collection)**：常用开源编译器，支持C、C++
- **Clang**：基于LLVM的C语言家族编译器，有良好的错误信息和现代化设计
- **Microsoft Visual C++ (MSVC)**：微软开发的C和C++编译器，集成在Visual Studio中

继续

```
% otool -tv fib
```

查看一个fib内部的汇编代码

对于windows来说, 可以用 `dumpbin /DISASM fib.exe`

```
fib:
(__TEXT,__text section)
_main:
00000000100000f20 pushq %rbp
00000000100000f21 movq %rsp, %rbp
00000000100000f24 subq $0x20, %rsp
00000000100000f28 movl $0x0, -0x4(%rbp)
00000000100000f2f movl $0x0, -0x8(%rbp)
00000000100000f36 movl $0x1, -0xc(%rbp)
00000000100000f3d leaq 0x56(%rip), %rdi
00000000100000f44 movl -0x8(%rbp), %esi
00000000100000f47 movb $0x0, %al
00000000100000f49 callq 0x100000f78
00000000100000f4e movl -0x8(%rbp), %esi
00000000100000f51 addl -0xc(%rbp), %esi
00000000100000f54 movl %esi, -0x10(%rbp)
00000000100000f57 movl -0xc(%rbp), %esi
00000000100000f5a movl %esi, -0x8(%rbp)
00000000100000f5d movl -0x10(%rbp), %esi
00000000100000f60 movl %esi, -0xc(%rbp)
00000000100000f63 movl %eax, -0x14(%rbp)
00000000100000f66 cmpl $0xff, -0x8(%rbp)
00000000100000f6d jl 0x100000f3d
00000000100000f73 jmp 0x100000f2f
```

看懂两方的联动

```
00000000100000f20 pushq %rbp
00000000100000f21 movq %rsp, %rbp
00000000100000f24 subq $0x20, %rsp
00000000100000f28 movl $0x0, -0x4(%rbp)
```

这四句话和源码没太多的关联性，只是setup Stack Frame

插播一段关于stack frame的复习

把栈帧想象成一组小抽屉，用来存放和管理每个函数的随身物品

- 如果函数需要输入参数（比如上文compiler复习的例子中，add需要两个数），这些参数会保存在stack frame中
- 调用一个函数时，程序需要记住函数执行完后回到哪里继续执行，这个返回地址就存储在stack frame里
- 函数里定义的variable在stack frame里分配空间
- Saved Registers（寄存器）：在函数执行前，某些 CPU registers的值会保存到stack frames中，这样在函数返回时可以恢复之前的状态，防止关键数据被覆盖

workflow:

在程序执行的过程中，当一个函数被调用时，stack frame 会被压入 stack（push 到栈里）；当函数结束时，栈帧会从栈中弹出（pop 出来），程序继续到之前的代码位置执行。

继续

```
0000000100000f2f movl $0x0, -0x8(%rbp)
```

这一句是把 \$0x0 移动到 -0x8(%rbp) 去，前者是一个0的值，后者是一个address offset。

0x8 = 内存中的具体位置。

rbp = Register base pointer，x86的栈基指针stack base pointer。一句话复习stack base pointer：sp 向当前 stack（栈）顶的位置，也就是栈中最上面的元素或空闲位置。每当新的数据进栈时，SP 会向下移动，指向新的栈顶位置；当数据被移出栈时，SP 又会向上移动。

0x8 => x

插播一段关于address offset的复习

Address Offset（地址偏移量）是相对于某个基准地址（base address）的距离或偏移量。可以理解为在特定的内存区域中，从起始位置开始计算的位移

表示某个变量、指令或数据存储到什么位置

假设我们有一个内存区域的起始地址 (base address) 是 1000, 而某个变量位于这个区域的 1010 地址处, 那么这个变量的 address offset 就是 10 (即 $1010 - 1000$), (意味着这个变量距离起始地址有 10 个单位的偏移量)

使用场景:

- arrays, struct
 - array 第一个元素的偏移量是 0, 第二个元素的偏移量是元素大小的倍数
- stack frames: 局部变量、参数和返回地址都有相对于栈帧起始地址的偏移量。编译器用这些偏移量来快速找到它们的位置
- memory management & assembly

继续

```
0000000100000f36 movl $0x1, -0xc(%rbp)
```

这边就是同上, 把1的值推进0xc

0xc => y

```
0000000100000f3d leaq 0x56(%rip), %rdi
0000000100000f44 movl -0x8(%rbp), %esi
0000000100000f47 movb $0x0, %al
0000000100000f49 callq 0x100000f78
```

这四行都是与 printf 相关的。前三行用来setup所需要的东西, callq用来call一个内存位置 0x100000f78, 目前还没出现。

0x56(%rip) 大概是new line string的内存位置

-0x8(%rbp) 就是x了, 上面有记录, 所以第二行就是在输出x的值

\$0x0 是什么?

总之, 就是在对应

```
printf("%d\n", x);
```

```
0000000100000f4e movl -0x8(%rbp), %esi
0000000100000f51 addl -0xc(%rbp), %esi
0000000100000f54 movl %esi, -0x10(%rbp)
```

第一行，第二行都是俩老熟人了，x和y。第一行把内存位置 0x8 所指的x的值，移动到esi register的值中

x --> esi

第二行用addl，把内存位置 0xc 所指的y的值，加到esi的值中

add y --> esi

第三行，把esi的值，移动到 0x10 这个内存位置，结合源码看也就是z。可以记一下笔记：

0x10 => z

问题：esi这个寄存器是从哪冒出来的？

解答：ESI 是 x86 架构中的一个寄存器，全称是 Extended Source Index。它属于一组称为“索引寄存器”（index registers）的一部分。ESI 和 EDI（Extended Destination Index）常常一起使用，用于数据处理和字符串操作。

esi是一个约定俗成的暂存位置，可以按需用来存储任意数据

总之，就是在对应

```
z = x + y;
```

```
0000000100000f57 movl -0xc(%rbp), %esi
0000000100000f5a movl %esi, -0x8(%rbp)
```

无需多言，（内存位置0xc所对应的）y（的值）推入esi推入x

总之，就是在对应

```
x = y;
```

```
0000000100000f5d movl -0x10(%rbp), %esi
0000000100000f60 movl %esi, -0xc(%rbp)
```

同上，对应

```
y = z;
```

```
0000000100000f63 movl %eax, -0x14(%rbp)
```

这句话是在？

```
0000000100000f66 cmpl $0xff, -0x8(%rbp)
0000000100000f6d j1l 0x100000f3d
0000000100000f73 jmp 0x100000f2f
```

第一行，用cmpl将 0xff 内存位置的值（大概是255）和x作比较

第二行，jl = jump if less than, (0x100000) f3d代表的是：

```
0000000100000f3d leaq 0x56(%rip), %rdi
```

是printf的一部分。结合上面就是如果x小于255，即跳回printf这一步。如果不小于255，↓

第三行，jmp = nonconditional jump, 直接跳回f2f，代表的是：

```
0000000100000f2f movl $0x0, -0x8(%rbp)
```

也就是 $x = 0$ 这一步。

然后程序就会一直一直一直跑下去啦~最后输出这样的结果：


结果

```
0
1
1
2
3
```


5
8
13
21
34
55
89
144
233
0
1
1
2
3
5
8
13
21
34
55
89
144
233
0
1
1
2
3
5
8
13
21
34
55
89
144
233
...

子子孙孙无穷匮也2333333

20241103 011451 （我知道很巧确实就是）

01:15:09 

截慢一步