

编译器到底对我的代码做了什么

大一的学年结束后我们可以用 C/C++, python, Java 等多种高级语言与计算机进行沟通让它为我们工作。大二学习了计算机组成原理以后我们知道计算机之所以能“听懂”这么多高级语言是因为有汇编语言的存在。但从高级语言到汇编语言乃至机器码的翻译过程究竟是怎样的，这其中计算机究竟经历了什么，或者它对我们的代码做了什么，这是我们这个学期将在编译原理这门课中学到的。

关键词：预处理，编译，汇编，链接器，C++, GCC, G++

一、引言

作为编译原理课实验部分的第一次准备工作，我以一个简单的斐波那契函数程序为例，在 Linux 环境 (Ubuntu18.04) 下，用 GCC 和 G++ 作为工具，通过不同的编译命令尝试了解我的编译器可以完成哪些工作或者具有哪些功能，以及一个 C++/C 程序在这整个编译过程中经历了什么。

程序源码如下：

```
#include<iostream>
using namespace std;

int main()//Fibonacci function
{
    int a,b,i,n,t;
    a=0;
    b=1;
    i=1;
    cin>>n;
    cout<<a<<endl;
    cout<<b<<endl;
    while(i<n)
    {
        t=b;
        b=a+b;
        cout<<b<<endl;
        a=t;
        i=i+1;
    }
    cout<<"new a is"<< a <<endl;
    cout<<"new b is"<< b <<endl;
    return 0;
}
```

在正式工作开始前先写两个自己遇到的 Linux 下编程的坑（其实是我太菜了）

(1) 关于 vi 编辑器的一个坑：

Ubuntu 中 18.04 中自带了 vi，但是当使用这个初始的 vi 时，会发现当你想用上下左右箭头去换行时，会直接输入字母 abcd，而且你的退格键也仅仅光标退格而并不删除字母。这简直会让第一次使用这个东西的人抓狂甚至忍不住想要砸键盘！（**我没说这是我室友的反应**）不过我们其实不用害怕，这是因为系统自带的 vi 其实并不是我们想象中的 vim，只要我们卸载原来的 vi 然后重装一下 vim 就行了。

sudo apt-get remove vim-common

sudo apt-get install vim

两行命令解决烦恼，但这才只是刚刚开始。

(2) 当我用把样例代码直接 copy 到 vi 中然后使用 g++ 编译时，出现了一个报错，内容是 **“error: stray ‘\302’ in program ^”** **(逐渐烦躁)**

这个错误产生的原因是我直接从 PPT 复制了代码，带入了中文空格，但是由于很难依靠肉眼找到这个中文空格到底在哪里，所以只能老老实实重新打一遍代码，还好代码不长。以前在 vs 上编程的时候如果出现中文字符都会直接给我标记出来，但是在 Linux 下的 vi 没有这个功能。看来在 Linux 下编程 **copy 代码需谨慎**，上次也是直接从 PDF 复制代码结果程序一直报错半天也没看出来到底咋回事，最后发现 PDF 中的“-”是中文的。

(3) 解决上一个问题以后程序还是没有顺利运行，这次编译时新的报错是这样的：

“error: ‘endl’ was not declared in this scope” **(心态爆炸)**

这次的原因是给的样例代码中没有加入“using namespace std”。Linux 下要求如果不加这个命令想要使用 cout 应该写成“std::cout”。在程序开头加上那句话后，终于可以成功编译了。

一个如此简单的程序如果用我们熟悉的 VS 去编写，从开始写到编译完成运行可能一共也用不了 3 分钟，但这次居然经历如此多的曲折，新手很容易在这个阶段开始怀疑人生，但其实这些都只是一些无关紧要的经验问题，Linux 环境下编程和 Windows 下可能在细节方面有很多不同，我们还需要努力适应。接下来终于可以进入正题。

一、预处理器

预处理阶段 g++ -E 命令

```
lsy@ubuntu:~/桌面/CP/Test2$ g++ -E test2.cpp -o test2.i
lsy@ubuntu:~/桌面/CP/Test2$ ls
test2.cpp  test2.i  test2.o
lsy@ubuntu:~/桌面/CP/Test2$
```

根据在网上了解到的，预处理阶段编译器处理源代码中以# 开始的预编译指令，例如展开所有宏定义、插入 include 指向的文件等，以获得经过预处理的源程序。生成 **test2.i** 文件后，打开，发现整个文件长度居然有 29178 行，然而源代码其实只有 20 行，查阅资料得知这是将源代码中的头文件进行了替代的结果，在查看文件过程中确实发现了很多文件路径，比如这样：

```
# 1 test2.cpp
# 1 "/usr/include/c++/7/iostream" 1 3
# 36 "/usr/include/c++/7/iostream" 3

# 37 "/usr/include/c++/7/iostream" 3

# 1 "/usr/include/x86_64-linux-gnu/c++/7/bits/c++config.h" 1 3
# 229 "/usr/include/x86_64-linux-gnu/c++/7/bits/c++config.h" 3
```

我们切换到/usr/include 目录，ls 一下，就能看到

```
lsy@ubuntu:/usr/include$ ls
aio.h          FlexLexer.h      mcheck.h        pwd.h           sys
aliases.h      fmtmsg.h          memory.h         python3.6m      syscall.h
alloca.h        fnmatch.h         misc             rdma            sysexits.h
argp.h          fpu_control.h     mntent.h        re_comp.h       syslog.h
argz.h          fstab.h           monetary.h      regex.h         tar.h
ar.h            fts.h             mqueue.h        regexp.h        termio.h
arpa            ftw.h             mtd             reglib          termios.h
asm             _G_config.h       net              resolv.h        tgmath.h
asm-generic     gconv.h           netash          rpc             thread_db.h
assert.h        getopt.h          netatalk         rpcsvc          time.h
bits            glob.h            netax25          sched.h         ttyent.h
byteswap.h      gnu               netdb.h          scsi            uchar.h
c++             gnumake.h         netetconet       search.h        ucontext.h
complex.h       gnu-versions.h   netinet          semaphore.h     ulimit.h
cpio.h          grp.h            netipx           setjmp.h        unistd.h
crypt.h         gshadow.h        netiucv          sgtty.h         ustat.h
ctype.h         iconv.h          netpacket        shadow.h        utime.h
dirent.h        ifaddrs.h        netrom           signal.h         utmp.h
dlfcn.h         inttypes.h        netrose          sound            utmpx.h
drm             langinfo.h        nfs              spawn.h         values.h
elf.h           lastlog.h         nl_types.h       stab.h          video
endian.h        libgen.h          nss.h            stdc-predef.h   wait.h
envz.h          libintl.h         obstack.h        stdint.h        wchar.h
err.h           libio.h           paths.h          stdio_ext.h     wctype.h
errno.h         limits.h          poll.h           stdio.h         wordexp.h
error.h         link.h            printf.h         stdlib.h        X11
execinfo.h      locale.h          proc_service.h   string.h        x86_64-linux-gnu
fcntl.h         malloc.h          protocols        strings.h        xen
features.h      malloc.h          pthread.h        stropts.h       xorg
fenv.h          math.h            pty.h           sudo_plugin.h
```

我们可以看到许多我们熟悉的头文件，如 **stdio.h, error.h** 等

由此我们可以确定这些路径正是头文件所在地址的路径，后面许多复杂的函数应该就是头文件中的具体内容。而在这个文件的最底部，就是我们的源码，只不过少了 **#include** 的部分

```
28155 using namespace std;
28156
28157 int main()
28158 {
28159     int a,b,i,n,t;
28160     a=0;
28161     b=1;
28162     i=1;
28163     cin>>n;
28164     cout<<a<<endl;
28165     cout<<b<<endl;
28166     while(i<n)
28167     {
28168         t=b;
28169         b=a+b;
28170         cout<<b<<endl;
28171         a=t;
28172         i=i+1;
28173     }
28174     cout<<"new a is"<<a<<endl;
28175     cout<<"new b is"<<b<<endl;
28176     return 0;
28177 }
28178
```

二、汇编器

编译过程 `g++ -S` 命令

```
lsy@ubuntu:~/桌面/CP/Test2$ g++ -S test2.cpp -o test2.s
lsy@ubuntu:~/桌面/CP/Test2$ ls
test2.cpp  test2.i  test2.o  test2.s
lsy@ubuntu:~/桌面/CP/Test2$
```

通过 `-S` 命令可以生成源代码的汇编文件，文件格式为 `.s` 文件，打开后可以查看目标程序的汇编代码。这个 24 行的源码最终生成的汇编代码有 169 行。

下图是生成的目标程序的汇编代码的一部分

```
17 main:
18 .LFB1493:
19 .cfi_startproc
20 pushq %rbp
21 .cfi_def_cfa_offset 16
22 .cfi_offset 6, -16
23 movq %rsp, %rbp
24 .cfi_def_cfa_register 6
25 subq $32, %rsp
26 movq %fs:40, %rax
27 movq %rax, -8(%rbp)
28 xorl %eax, %eax
29 movl $0, -24(%rbp)
30 movl $1, -20(%rbp)
31 movl $1, -16(%rbp)
32 leaq -28(%rbp), %rax
33 movq %rax, %rsi
34 leaq _ZSt3cin(%rip), %rdi
35 call _ZNSirsERi@PLT
36 movl -24(%rbp), %eax
37 movl %eax, %esi
38 leaq _ZSt4cout(%rip), %rdi
39 call _ZNSolsEi@PLT
40 movq %rax, %rdx
41 movq _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_0_ES6_@GOTPCREL(%rip), %rax
42 movq %rax, %rsi
43 movq %rdx, %rdi
44 call _ZNSolsEPFRSoS_E@PLT
45 movl -20(%rbp), %eax
46 movl %eax, %esi
```

但是观察了一下发现了一些不像汇编代码的东西，为了易于区分用 `vi` 打开，

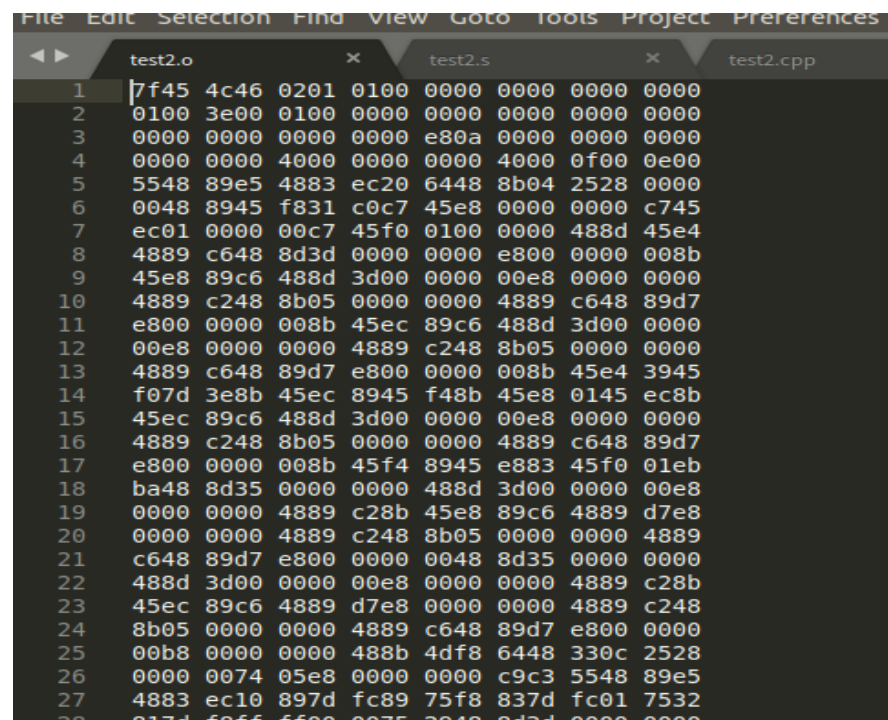
```
.local _ZStL8__ioinit
.comm _ZStL8__ioinit,1,1
.LC0:
.string "new a is"
.LC1:
.string "new b is"
.text
.globl main
.type main, @function
main:
.LFB1493:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $32, %rsp
movq %fs:40, %rax
movq %rax, -8(%rbp)
xorl %eax, %eax
movl $0, -24(%rbp)
movl $1, -20(%rbp)
```

可以发现蓝色字体的部分应该基本都是我们平常所说的汇编代码, 黄色部分的东西不太清楚是什么, *CFI* 是出现频率最高的关键字, 查阅资料后得知 *CFI* 全称是 *Call Frame Instructions*, 即调用框架指令。 *CFI* 提供的调用框架信息, 为实现堆栈回绕(stack unwinding)或异常处理(exception handling)提供了方便, 它在汇编指令中插入指令符(directive), 以生成 DWARF[3] 可用的堆栈回绕信息。 “*.cfi_def_cfa_offset 16*” 一句, 该指令表示: 此处距离 *CFA* 地址为 16 字节。 *CFA(Canonical Frame Address)* 是标准框架地址, 它被定义为在前一个调用框架中调用当前函数时的栈顶指针。

三、汇编器

g++ -c 命令

这一命令执行后会生成一个.o 文件, 是一个二进制文件, 这个过程实际上把汇编语言程序代码翻译成目标机器指令的过程。其最终生成的是可重定位的机器代码。在这个过程中其实还有许多中间代码的生成, 关于这些部分将在本文后面的部分提到。




四、链接器

.o 文件并不是可执行文件, 我们还需要通过链接器生成一个可执行文件。 gcc / g++ 链接器是 gas 提供的, 负责将程序的目标文件与所需的所有附加的目标文件连接起来, 最终生成可执行文件。附加的目标文件包括静态连接库和动态连接库。生成的 test2.o, 将其与 C 标准输入输出库进行连接, 最终生成程序 test2, 可以直接运行。

```
lsy@ubuntu:~/桌面/CP/Test2$ g++ test2.o -o test2
lsy@ubuntu:~/桌面/CP/Test2$ ls
test2  test2.cpp  test2.i  test2.o  test2.s
lsy@ubuntu:~/桌面/CP/Test2$ ./test2
5
0
1
1
2
3
5
new a is3
new b is5
lsy@ubuntu:~/桌面/CP/Test2$
```

至此我们已经基本浏览了一段源代码变身为可执行程序的全过程, 经历这些过程后我们可以发现一共生成了这些东西:

 test2	13.2 KB
 test2.cpp	362 字节
 test2.i	666.3 KB
 test2.o	3.8 KB
 test2.s	3.9 KB

但我们还没有完成全部任务, 编译器还有许多我们没看到的功能, 这些文件生成过程中还有许多我们没有看到的中间过程。

五、编译优化

-O0、-O1、-O2、-O3 参数

可能是为了弥补部分程序员 (比如我) 在写代码的时候由于对计算机工作原理的理解不够深入可能做出的“愚蠢”决定, 编译器其实自带了一些优化功能, 用 VS 编程的时候编译器悄悄地在进行这个工作, 但在 Linux 下我们可以人为的输入参数来决定编译器的优化级别。我用一个循环展开的例子来做介绍。源码如下:

```

#include<iostream>
#include<time.h>
using namespace std;
#define OP *;
#define len 100000;
void combine1(float* a, float dest);
void combine2(float* a, float dest);
int main()
{
    srand((unsigned)time(NULL));
    clock_t start1, end1, start2, end2;
    float a[100000];
    for (int i = 0; i < 100000; i++)
    {
        a[i] = rand() /float(100.00);
    }
    float dest1=1;
    float dest2=1;
    start1 = clock();
    combine1(a, dest1);
    end1 = clock();
    cout << (end1 - start1) / float(CLOCKS_PER_SEC) << endl;
    start2 = clock();
    combine2(a, dest2);
    end2 = clock();
    cout << (end2 - start2)/float(CLOCKS_PER_SEC) << endl;
    return 0;
}

```

```

void combine1(float* a, float dest)
{
    dest = 1;
    for (int i = 0; i<100000; i++)
    {
        float val;
        val = a[i];
        dest = dest * val;
    }
}
void combine2(float* a, float dest)
{
    int i;
    int l = len;
    int limit = len - 1;
    float*data = a;
    float acc = 1;
    for (i = 0; i<limit; i += 2)
    {
        acc = (acc * data[i])*data[i + 1];
    }
    for (; i<l; i++)
    {
        acc = acc*data[i];
    }
}

```

combine1 和 combine2 都是对一个长度为 100000 的 float 数组进行累乘运算的函数，这个数组的值是随机赋予的。而 combine2 其实是对 combine1 通过循环展开后进行了优化的版本。根据计算，combine2 的性能应该是 combine1 的 4-5 倍（此数据来源于《深入理解计算机系统》p367）。那么按理说我们运行的话，combine2 函数用的时间应该就是 combine1 的四分之一左右。那么事实果真如此吗？


```
lsy@ubuntu:~$ ./test
0.000277
0.000217
lsy@ubuntu:~$ ./test
0.000325
0.000234
lsy@ubuntu:~$ ./test
0.000254
0.00023
lsy@ubuntu:~$
```

我连续执行三次 test，可以看到每次 combine2 确实会比 combine1 快，但并不是理论上的快了 4-5 倍。当然并不是我们对函数的性能分析出错了，而是因为针对 combine1 那个“愚蠢”的函数，编译器实在看不下去在编译的时候已经帮我们做了一部分优化。但这个优化并不彻底，毕竟编译器不能越权，如果程序员执意用冒泡排序，那么编译器也没必要非要把他的冒泡优化为快排，毕竟它并没有义务为程序员的愚蠢买单。如果我们去 Windows 环境下编译的话，也能发现一样的结果。

那我们现在使用 -O 参数优化编译一下，再执行试试

```
lsy@ubuntu:~$ g++ -O1 test.cpp -o test01
lsy@ubuntu:~$ ./test01
1e-06
1e-06
lsy@ubuntu:~$ g++ -O2 test.cpp -o test02
lsy@ubuntu:~$ ./test02
1e-06
1e-06
```

可以看到，无论是进行 O1 优化还是 O2 优化，都已经让这两个函数的运行时间过小以至于难以区分差别。

编译器是由一群非常高明的程序员编写的，他们在构造编译器的优化功能时已经把一些基本的循环展开、减少非必要过程调用等常见的优化方式加入了其中。关于 GCC 我们可以通过下面这条命令查看具体的优化选项：

gcc -Q --help=optimizers -O1

如果要查看 -O2 级别的优化选项只需要把最后一个 O1 改为 O2 即可。

```
-faggressive-loop-optimizations [enabled]
-falign-functions [disabled]
-falign-jumps [disabled]
-falign-labels [disabled]
-falign-loops [disabled]
-fassociative-math [disabled]
-fasynchronous-unwind-tables [enabled]
-fauto-inc-dec [enabled]
-fbranch-count-reg [enabled]
-fbranch-probabilities [disabled]
-fbranch-target-load-optimize [disabled]
-fbranch-target-load-optimize2 [disabled]
-fbtr-bb-exclusive [disabled]
-fcaller-saves [disabled]
-fcode-hoisting [disabled]
-fcombine-stack-adjustments [enabled]
-fcompare-elim [enabled]
-fconserve-stack [disabled]
-fcprop-registers [enabled]
-fcrossjumping [disabled]
-fcse-follow-jumps [disabled]
```


查询结果中，【enabled】表示这个选项在此级别优化中开启，【disabled】表示未开启。

对比 O2 和 O1 的优化选项就会发现 O2 比 O1 enabled 的选项更多。

关于所有的优化级别可以做出如下总结：

O0 选项不进行任何优化，在这种情况下，编译器尽量的缩短编译消耗（时间，空间），此时，debug 会产出和程序预期的结果。当程序运行被断点打断，此时程序内的各种声明是独立的，我们可以任意的给变量赋值，或者在函数体内把程序计数器指到其他语句,以及从源程序中 精确地获取你期待的结果。

O1 优化会消耗少多的编译时间，它主要对代码的分支，常量以及表达式等进行优化。

O2 会尝试更多的寄存器级的优化以及指令级的优化，它会在编译期间占用更多的内存和编译时间。

O3 在 O2 的基础上进行更多的优化，例如使用伪寄存器网络，普通函数的内联，以及针对循环的更多优化。

Os 主要是对代码大小的优化，我们基本不用做更多的关心。通常各种优化都会打乱程序的结构，让调试工作变得无从着手。并且会打乱执行顺序，依赖内存操作顺序的程序需要做相关处理才能确保程序的正确性。

六、中间代码生成

在汇编器将源代码转换为机器码的过程中，其实有很多中间代码的生成，我们可以通过以下几个命令去尝试了解

gcc -fdump-tree-original-raw test.c //得到文本格式的 AST

gcc -O0 -fdump-tree-all-graph test.c //中间代码生成的多阶段输出

然后 ls 一下就会发现居然生成这么多东西……

```
stack.png
test2.cpp
test2.cpp.003t.original
test2.cpp.229r.expand
test2.cpp.229r.expand.dot
test2.cpp.230r.vregs
test2.cpp.230r.vregs.dot
test2.cpp.231r.into_cfglayout
test2.cpp.231r.into_cfglayout.dot
test2.cpp.232r.jump
test2.cpp.232r.jump.dot
test2.cpp.244r.reginfo
test2.cpp.244r.reginfo.dot
test2.cpp.264r.outof_cfglayout
test2.cpp.264r.outof_cfglayout.dot
test2.cpp.265r.split1
test2.cpp.265r.split1.dot
test2.cpp.267r.dfinit
test2.cpp.267r.dfinit.dot
test2.cpp.268r.mode_sw
test2.cpp.268r.mode_sw.dot
test2.cpp.269r.asmcons
test2.cpp.269r.asmcons.dot
test2.cpp.273r.ira.dot
test2.cpp.274r.reload
test2.cpp.274r.reload.dot
test2.cpp.278r.split2
test2.cpp.278r.split2.dot
test2.cpp.282r.pro_and_epilogue
test2.cpp.282r.pro_and_epilogue.dot
test2.cpp.285r.jump2
test2.cpp.285r.jump2.dot
test2.cpp.298r.stack
test2.cpp.298r.stack.dot
test2.cpp.299r.alignments
test2.cpp.299r.alignments.dot
test2.cpp.301r.mach
test2.cpp.302r.barriers
test2.cpp.306r.shorten
test2.cpp.307r.nothrow
test2.cpp.308r.dwarf2
test2.cpp.309r.final
test2.cpp.310r.dfinish
test2.i
test2.s
lsy@ubuntu:~/桌面/CP/Test2/MakeFile$
```

original 文件是第一条命令生成的文件，其他都是第二条命令生成的，打开.original 文件

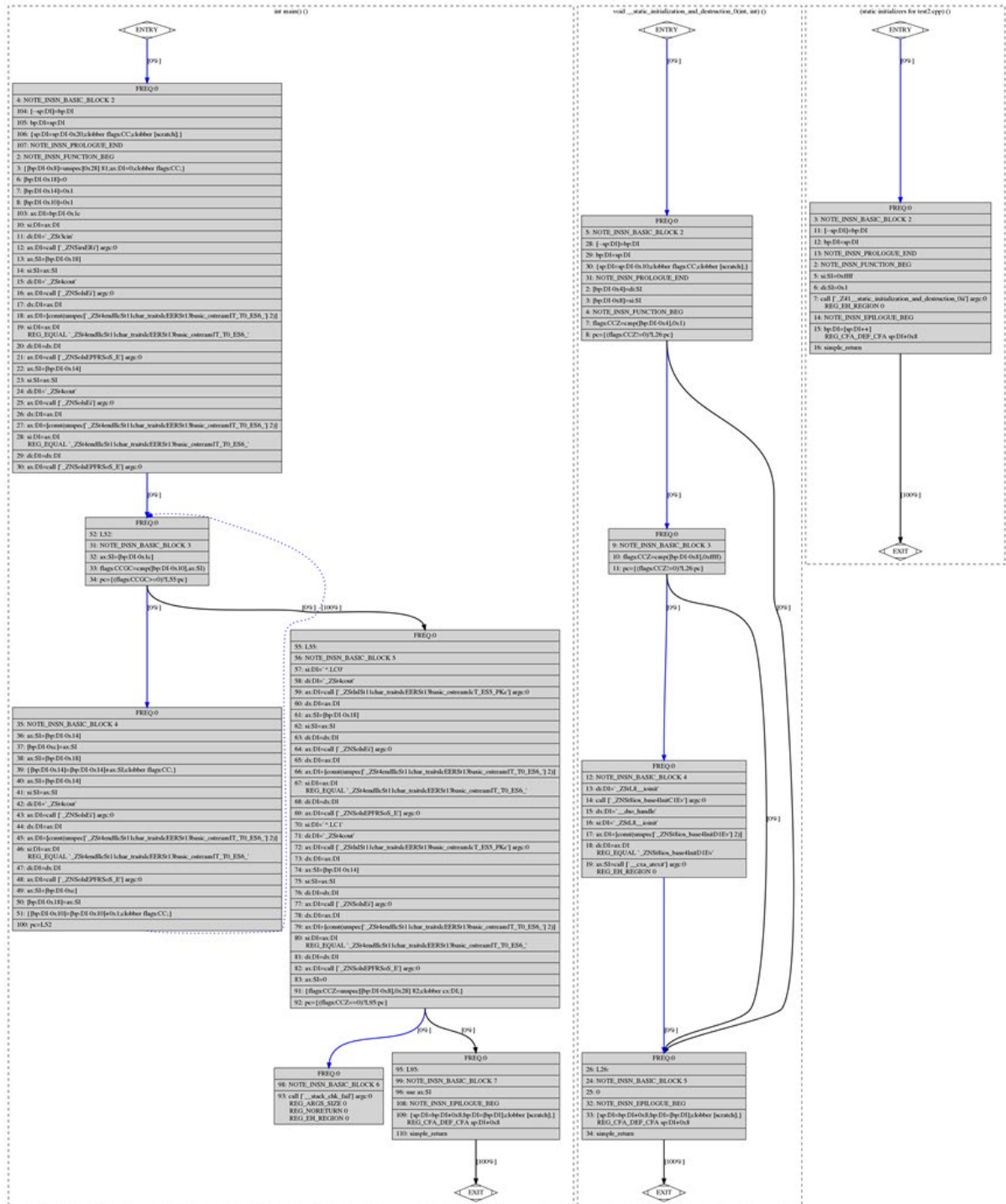
```
1
2 ;; Function std::exception::exception() (null)
3 ;; enabled by -tree-original
4
5 @1      must_not_throw_expr type: @2      line: 63      body: @3
6 @2      void_type          name: @4      algn: 8
7 @3      statement_list     0 : @5      1 : @6
8 @4      type_decl          name: @7      type: @2      srcp: <built-in>:0
9                               note: artificial
10 @5      cleanup_point_expr type: @2      op 0: @8
11 @6      bind_expr          type: @2      body: @9
12 @7      identifier_node    strg: void    lngt: 4
13 @8      expr_stmt          type: @2      expr: @10
14 @9      cleanup_point_expr type: @2      op 0: @11
15 @10     modify_expr        type: @2      op 0: @12      op 1: @13
16 @11     expr_stmt          type: @2      expr: @14
17 @12     indirect_ref       type: @15     op 0: @16
18 @13     constructor        lngt: 0
19 @14     convert_expr        type: @2      op 0: @17
20 @15     record_type        size: @18     algn: 64      bfld: @19
21 @16     nop_expr           type: @20     op 0: @21
22 @17     modify_expr        type: @22     op 0: @23      op 1: @24
23 @18     integer_cst        type: @25     int: 64
24 @19     record_type        name: @26     size: @18     algn: 64
25                               vfld: @27     tag : struct   flds: @27
26                               fncls: @28    binf: @29
27 @20     reference_type      size: @18     algn: 64      refd: @15
28 @21     parm_decl          name: @30     type: @31     scpe: @28
29                               srcp: exception.h:63 note: artificial
```

不出意外的看不懂

而.dot 文件可以在 Linux 中通过 *graphviz* 工具实现可视化查看，
我选取了一个文件名中带 stack 的 dot 文件，猜测和堆栈有关

dot -T png test2.cpp.298r.stack.dot -o stack.png

得到了下面这个树状图



可以看到整个图其实大概分为三个部分，在最上面给出了每部分的名称，

int main() ()

void __static_initialization_and_destruction_0(int, int) ()

(static initializers for test2.cpp) ()

可以看出最长的那部分是关于 main 函数的，而两外两部分从函数名推测是一些预编译阶段进行堆栈或者环境变量初始化的工作过程。

七、结语

简单来讲，编译器就是将“一种语言”翻译为“另一种语言”的程序。一个现代编译器的主要工作流程：源代码 (source code) → 预处理器 (preprocessor) → 编译器 (compiler) → 目标代码 (object code) → 链接器 (Linker) → 可执行程序 (executables)。我们现在能使用 C++，Java，python 这些高级语言进行编程，得益于编译器帮助我们翻译了我们的代码。

编译器的发展过程中，有限状态自动机、正则表达式、上下文无关文法等思想理论的提出或者相关技术的发明极大的推动了整个计算机产业的发展，今天的我们站在巨人的肩膀上，继续为互联网和计算机的发展贡献智（苦）慧（力）。

如果没有编译器让程序员靠汇编代码（这好像是我下周的作业？）或者机器码去编写程序，无疑是一件无比痛苦的事情。而如果没有编译器帮我们优化我们愚蠢的代码，也许机器也会觉得“痛苦”。由此我们得出一个道理：沟通，让世界更美好。