Lab2 编译器认知实验

姓名/学号：宋尚儒/1120180717

实验目的

本实验的目的是了解工业界常用的编译器 GCC 和 LLVM，熟悉编译器的安装和使用过程，观察编译器工作过程中生成的中间文件的格式和内容， 了解编译器的优化效果， 为编译器的学习和构造奠定基础。

实验内容

本实验主要的内容为在 Linux 平台上安装和运行工业界常用的编译器 GCC 和LLVM，如果系统中没有安装，则需要首先安装编译器， 安装完成后编写简单的测试程序，使用编译器编译，并观察中间输出结果。

实验过程

实验环境

虚拟机：

- 操作系统：VMware Linux ubuntu 4.15.0-136-generic
- 处理器数量：2
- 每个处理器内核数量：2
- 物理内存：4GB

宿主机：

- 操作系统：Windows 10 家庭版 20H2
- 处理器核数：8
- 处理器主频：2.30GHz
- 处理器缓存
  - L1：256KB
  - L2：1.0MB
  - L3：8.0MB
- 物理内存：16GB

实验过程与分析

编写测试程序

共编写两个测试程序，分别是`main.c`、`Qsort.c`，代码结构参考如下

`main.c`

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 5000005

char rpath[] = "../data/in.txt";
```

```
int arr[MAXN];

extern void Qsort(int a[], int low, int high);

int get_arr(FILE *fp, int arr[])
{
    int id=0;
    fseek(fp, 0, 0);
    while(fscanf(fp,"%d",&arr[id])!=EOF)
        id++;
    return id;
}

int main()
{
    FILE *fp;
    fp = fopen(rpath, "r");
    int len=get_arr(fp,arr);
    fclose(fp);
    Qsort(arr,0,len-1);
    for(int i=len/2;i<len/2+10;i++)
        printf("%d\n",arr[i]);
}
```

`Qsort.c`

```
int Partition(int a[], int low, int high)
{
    int key = a[low];
    while(low<high)
    {
        while(low<high && a[high] >= key) --high;
        a[low] = a[high];
        while(low<high && a[low] <= key) ++low;
        a[high] = a[low];
    }
    a[low] = key;
    return low;
}
void Qsort(int a[], int low, int high)
{
    if(low < high)
    {
        int loc = Partition(a, low, high);
        Qsort(a, low, loc-1);
        Qsort(a, loc+1, high);
    }
}
```

使用的数据文件`in.txt`由如下C语言程序生成

```cpp
#include <bits/stdc++.h>
using namespace std;

int cnt=5e6;
int limit=1e8;

int main()
{
        freopen("in.txt","w",stdout);
        for(int i=0;i<cnt;i++)
         cout<<(rand()*rand())%limit<<endl;
        return 0;
}
```

GCC

- 查看编译器版本

```
# ssr @ ubuntu in ~/compile-lab/lab2 [0:37:00]
$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

  可知版本为gcc 5.4.0

- 使用编译器编译单个文件

```
# ssr @ ubuntu in ~/compile-lab/lab2 [0:38:46]
$ gcc -c ./src/Qsort.c  -o ./bin/Qsort.o
```

- 使用编译器编译链接多个文件

```
# ssr @ ubuntu in ~/compile-lab/lab2 [0:39:56]
$ gcc ./src/main.c  ./src/Qsort.c  -o ./bin/main
```

  生成`main`文件为可执行文件，执行结果如下

```
# ssr @ ubuntu in ~/compile-lab/lab2/bin [0:45:30]
$ ./main
43873120
43873122
43873137
43873166
43873184
43873245
43873258
43873301
43873336
43873336
```

- 查看预处理结果

```
# ssr @ ubuntu in ~/compile-lab/lab2 [0:46:51]
$ gcc -E  ./src/Qsort.c  -o ./src/Qsort.i

# ssr @ ubuntu in ~/compile-lab/lab2 [0:46:53]
$ gcc -E  ./src/main.c   -o ./src/main.i
```

  生成`Qsort.i`和`main.i`文件

  其中`Qsort.i`文件参考如下，因为没有引入库，所以结构相对简单，仅包含自定义的函数和通用的头

main.c × Qsort.i × main.i ×

```
# 1 "./src/Qsort.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "./src/Qsort.c"
int Partition(int a[], int low, int high)
{
    int key = a[low];
    while(low<high)
    {
        while(low<high && a[high] >= key) --high;
        a[low] = a[high];
        while(low<high && a[low] <= key) ++low;
        a[high] = a[low];
    }
    a[low] = key;
    return low;
}
void Qsort(int a[], int low, int high)
{
    if(low < high)
    {
        int loc = Partition(a, low, high);
        Qsort(a, low, loc-1);
        Qsort(a, loc+1, high);
    }
}
```

`main.i`复杂了很多，行数达到了1941，原因在于引入了两个库文件的函数定义，占用了较多字节，两库函数对应部分的结构可以参考如下，因为较长就不做全部展示

```
# 1 "./src/main.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 367 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 410 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 411 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 368 "/usr/include/features.h" 2 3 4
# 391 "/usr/include/features.h" 3 4
```

```
# 2 "./src/main.c" 2
# 1 "/usr/include/stdlib.h" 1 3 4
# 32 "/usr/include/stdlib.h" 3 4
# 1 "/usr/lib/gcc/x86_64-linux-gnu/5/include/stddef.h" 1 3 4
# 328 "/usr/lib/gcc/x86_64-linux-gnu/5/include/stddef.h" 3 4
typedef int wchar_t;
# 33 "/usr/include/stdlib.h" 2 3 4
```

末尾结构可参考如下，可见主函数基本没有变换，而自定义函数中因为使用了宏EOF被做出修改

```
# 5 "./src/main.c"
char rpath[] = "../data/in.txt";
int arr[5000005];

extern void Qsort(int a[], int low, int high);

int get_arr(FILE *fp, int arr[])
{
    int id=0;
    fseek(fp, 0, 0);
    while(fscanf(fp,"%d",&arr[id])!=
# 14 "./src/main.c" 3 4
                                   (-1)
# 14 "./src/main.c"
                                      )
        id++;
    return id;
}

int main()
{
    FILE *fp;
    fp = fopen(rpath, "r");
    int len=get_arr(fp,arr);
    fclose(fp);
    Qsort(arr,0,len-1);
    for(int i=len/2;i<len/2+10;i++)
        printf("%d\n",arr[i]);
}
```

- 查看语法分析树

如下图所示会产生大量中间文件，以`Qsort.c`对应的两个重要文件`Qsort.c.001t.tu`和`Qsort.c.004t.gimple`为例进行分析

```
# ssr @ ubuntu in ~/compile-lab/lab2/src [4:57:52] C:1
$ gcc -fdump-tree-all main.c Qsort.c

# ssr @ ubuntu in ~/compile-lab/lab2/src [4:58:03]
$ ls | grep Qsort
Qsort.c
Qsort.c.001t.tu
Qsort.c.002t.class
Qsort.c.003t.original
Qsort.c.004t.gimple
Qsort.c.006t.omplower
Qsort.c.007t.lower
Qsort.c.010t.eh
Qsort.c.011t.cfg
Qsort.c.012t.ompexp
Qsort.c.017t.fixup_cfg1
Qsort.c.018t.ssa
Qsort.c.025t.fixup_cfg3
Qsort.c.026t.inline_param1
Qsort.c.027t.einline
Qsort.c.042t.profile_estimate
Qsort.c.045t.release_ssa
Qsort.c.046t.inline_param2
Qsort.c.068t.fixup_cfg4
Qsort.c.183t.veclower
Qsort.c.184t.cplxlower0
Qsort.c.191t.optimized
Qsort.c.271t.statistics
```

以下两图为在符号表文件`Qsort.c.001t.tu`中找到的源码中两函数原型对应的标识节点的结构

```
@2095   identifier_node  strg: Partition              lngt: 9
@2096   function_type    size: @12      algn: 8       retn: @3
                         prms: @2104
@2097   function_decl    name: @2105    type: @2106   scpe: @155
                         srcp: Qsort.c:14             args: @2107
                         link: extern   body: @2108
@2098   parm_decl        name: @2109    type: @531    scpe: @2091
                         srcp: Qsort.c:1              chain: @2110
                         argt: @531     size: @22     algn: 64
                         used: 1
@2105   identifier_node  strg: Qsort    lngt: 5
@2106   function_type    size: @12      algn: 8       retn: @129
                         prms: @2119
@2107   parm_decl        name: @2109    type: @531    scpe: @2097
                         srcp: Qsort.c:14             chain: @2120
                         argt: @531     size: @22     algn: 64
                         used: 1
@2108   bind expr        type: @129     body: @2121
```

以下两图为在控制流图文件`Qsort.c.004t.gimple`的内容，可以看到其与C源码在较多部分比较相似，不同之处有如下

- 较多的形式为`D.x`的局部变量替代了C源码中的部分语法
- 使用`goto`和`if/else`替换了原来的循环和选择结构

```
Partition (int * a, int low, int high)
{
  long unsigned int D.1852;
  long unsigned int D.1853;
  int * D.1854;
  long unsigned int D.1856;
  long unsigned int D.1857;
  int * D.1858;
  int D.1859;
  long unsigned int D.1860;
  long unsigned int D.1861;
  int * D.1862;
  long unsigned int D.1863;
  long unsigned int D.1864;
  int * D.1865;
  int D.1866;
  long unsigned int D.1868;
  long unsigned int D.1869;
  int * D.1870;
```

```
int D.1871;
long unsigned int D.1872;
long unsigned int D.1873;
int * D.1874;
long unsigned int D.1875;
long unsigned int D.1876;
int * D.1877;
int D.1878;
long unsigned int D.1879;
long unsigned int D.1880;
int * D.1881;
int D.1882;
int key;

D.1852 = (long unsigned int) low;
D.1853 = D.1852 * 4;
D.1854 = a + D.1853;
key = *D.1854;
goto <D.1843>;
<D.1842>:
goto <D.1837>;
<D.1836>:
high = high + -1;
<D.1837>:
if (low < high) goto <D.1855>; else goto <D.1838>;
<D.1855>:
D.1856 = (long unsigned int) high;
D.1857 = D.1856 * 4;
D.1858 = a + D.1857;
D.1859 = *D.1858;
if (D.1859 >= key) goto <D.1836>; else goto <D.1838>;
<D.1838>:
D.1860 = (long unsigned int) low;
D.1861 = D.1860 * 4;
D.1862 = a + D.1861;
D.1863 = (long unsigned int) high;
D.1864 = D.1863 * 4;
D.1865 = a + D.1864;
D.1866 = *D.1865;
*D.1862 = D.1866;
goto <D.1840>;
<D.1839>:
low = low + 1;
<D.1840>:
if (low < high) goto <D.1867>; else goto <D.1841>;
<D.1867>:
D.1868 = (long unsigned int) low;
D.1869 = D.1868 * 4;
D.1870 = a + D.1869;
D.1871 = *D.1870;
if (D.1871 <= key) goto <D.1839>; else goto <D.1841>;
<D.1841>:
D.1872 = (long unsigned int) high;
D.1873 = D.1872 * 4;
D.1874 = a + D.1873;
D.1875 = (long unsigned int) low;
D.1876 = D.1875 * 4;
D.1877 = a + D.1876;
```

```
  D.1878 = *D.1877;
  *D.1874 = D.1878;
  <D.1843>:
  if (low < high) goto <D.1842>; else goto <D.1844>;
  <D.1844>:
  D.1879 = (long unsigned int) low;
  D.1880 = D.1879 * 4;
  D.1881 = a + D.1880;
  *D.1881 = key;
  D.1882 = low;
  return D.1882;
}


Qsort (int * a, int low, int high)
{
  int D.1886;
  int D.1887;

  if (low < high) goto <D.1884>; else goto <D.1885>;
  <D.1884>:
  {
    int loc;

    loc = Partition (a, low, high);
    D.1886 = loc + -1;
    Qsort (a, low, D.1886);
    D.1887 = loc + 1;
    Qsort (a, D.1887, high);
  }
  <D.1885>:
}
```

- 查看中间代码生成结果

  如下图所示会产生大量中间文件，以`Qsort.c`对应的文件`Qsort.c.192r.expand`和`Qsort.c.270r.dfinish`为例

```
# ssr @ ubuntu in ~/compile-lab/lab2/src [5:33:37]
$ gcc -fdump-rtl-all main.c Qsort.c

# ssr @ ubuntu in ~/compile-lab/lab2/src [5:33:55]
$ ls | grep Qsort
Qsort.c
Qsort.c.192r.expand
Qsort.c.193r.vregs
Qsort.c.194r.into_cfglayout
Qsort.c.195r.jump
Qsort.c.207r.reginfo
Qsort.c.225r.outof_cfglayout
Qsort.c.226r.split1
Qsort.c.228r.dfinit
Qsort.c.229r.mode_sw
Qsort.c.230r.asmcons
Qsort.c.234r.ira
Qsort.c.235r.reload
Qsort.c.238r.split2
Qsort.c.242r.pro_and_epilogue
Qsort.c.245r.jump2
Qsort.c.258r.stack
Qsort.c.259r.alignments
Qsort.c.261r.mach
Qsort.c.262r.barriers
Qsort.c.266r.shorten
Qsort.c.267r.nothrow
Qsort.c.268r.dwarf2
Qsort.c.269r.final
Qsort.c.270r.dfinish
```

  以下两图为在文件`Qsort.c.192r.expand`中找到的源码中两函数原型对应的结构的头部，该文件是从GIMPLE转向RTL的第一步

Partition

```
;; Function Partition (Partition, funcdef_no=0, decl_uid=1833, cgraph_uid=0, symbol_order=0)

Partition 0: size 4 align 4
        key_12

;; Generating RTL for gimple basic block 2

;; Generating RTL for gimple basic block 3

;; Generating RTL for gimple basic block 4

;; Generating RTL for gimple basic block 5
```

Qsort

```
;; Function Qsort (Qsort, funcdef_no=1, decl_uid=1848, cgraph_uid=1, symbol_order=1)

Partition 0: size 4 align 4
        loc_7

;; Generating RTL for gimple basic block 2

;; Generating RTL for gimple basic block 3

;; Generating RTL for gimple basic block 4


try_optimize_cfg iteration 1
```

以下两图为在文件`Qsort.c.270r.dfinish`中找到的源码中两函数原型对应的结构的头部，该文件是多遍优化的结果之一

Partition

```
;; Function Partition (Partition, funcdef_no=0, decl_uid=1833, cgraph_uid=0, symbol_order=0)

(note 1 0 6 NOTE_INSN_DELETED)
(note 6 1 123 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn/f 123 6 124 2 (set (mem:DI (pre_dec:DI (reg/f:DI 7 sp)) [0  S8 A8])
        (reg/f:DI 6 bp)) Qsort.c:2 65 {*pushdi2_rex64}
     (nil))
(insn/f 124 123 125 2 (set (reg/f:DI 6 bp)
        (reg/f:DI 7 sp)) Qsort.c:2 89 {*movdi_internal}
     (nil))
(insn 125 124 126 2 (set (mem/v:BLK (scratch:DI) [0  A8])
```

Qsort

```
;; Function Qsort (Qsort, funcdef_no=1, decl_uid=1848, cgraph_uid=1, symbol_order=1)

(note 1 0 6 NOTE_INSN_DELETED)
(note 6 1 49 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn/f 49 6 50 2 (set (mem:DI (pre_dec:DI (reg/f:DI 7 sp)) [0  S8 A8])
        (reg/f:DI 6 bp)) Qsort.c:15 65 {*pushdi2_rex64}
     (nil))
(insn/f 50 49 51 2 (set (reg/f:DI 6 bp)
        (reg/f:DI 7 sp)) Qsort.c:15 89 {*movdi_internal}
     (nil))
(insn/f 51 50 52 2 (parallel [
```

- 查看生成的目标代码

  以文件`Qsort.c`为例，输入`gcc -S ./src/Qsort.c -o Qsort.s`，生成文件`Qsort.s`,内容较长，仅展示部分

  下图展示`Qsort.c`中第一个函数Partition对应的汇编代码的头部结构和部分汇编语言

```
        .file   "Qsort.c"
        .text
        .globl  Partition
        .type   Partition, @function
Partition:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    %rdi, -24(%rbp)
        movl    %esi, -28(%rbp)
        movl    %edx, -32(%rbp)
        movl    -28(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -24(%rbp), %rax
        addq    %rdx, %rax
        movl    (%rax), %eax
        movl    %eax, -4(%rbp)
        jmp     .L2
```

下图展示`Qsort.c`中第二个函数Qsort对应的汇编代码的头部结构和部分汇编语言，可以看出其调用函数Partition

```
        .globl  Qsort
        .type   Qsort, @function
Qsort:
.LFB1:
        .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    %rdi, -24(%rbp)
        movl    %esi, -28(%rbp)
        movl    %edx, -32(%rbp)
        movl    -28(%rbp), %eax
        cmpl    -32(%rbp), %eax
        jge     .L13
        movl    -32(%rbp), %edx
        movl    -28(%rbp), %ecx
        movq    -24(%rbp), %rax
        movl    %ecx, %esi
        movq    %rax, %rdi
        call    Partition
```

LLVM

- 查看编译器的版本

```
# ssr @ ubuntu in ~/compile-lab/lab2 [6:07:44] C:1
$ clang --version
clang version 3.8.0-2ubuntu4 (tags/RELEASE_380/final)
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

- 使用编译器编译单个文件

```
# ssr @ ubuntu in ~/compile-lab/lab2 [20:43:40]
$ clang -c ./src/main.c -o ./bin/main.o
```

- 使用编译器编译链接多个文件

```
# ssr @ ubuntu in ~/compile-lab/lab2 [20:44:15]
$ clang ./src/main.c ./src/Qsort.c  -o ./bin/main
```

生成可执行文件main，执行结果如下

```
# ssr @ ubuntu in ~/compile-lab/lab2/bin [20:46:03]
$ ./main
43873120
43873122
43873137
43873166
43873184
43873245
43873258
43873301
43873336
43873336
```

- 查看编译流程和阶段

```
# ssr @ ubuntu in ~/compile-lab/lab2 [20:48:34]
$ clang -ccc-print-phases ./src/main.c  -c
0: input, "./src/main.c", c
1: preprocessor, {0}, cpp-output
2: compiler, {1}, ir
3: backend, {2}, assembler
4: assembler, {3}, object
```

  - 0：输入文件，找到源文件
  - 1：预处理
  - 2：编译器前端生成IR中间代码
  - 3：编译器后端生成汇编代码
  - 4：将汇编文件变为对象文件
- 查看词法分析结果

  以文件`Qsort.c`为例，输入`clang ./src/Qsort.c  -Xclang -dump-tokens`，输出较长，仅展示部分结果

  下图展示编译器识别出`Qsort.c`中第一个函数Partition的函数头，其格式为`int Partition(int a[], int low, int high)`，正好与以下字段对应

```
int 'int'       [StartOfLine]  Loc=<./src/Qsort.c:1:1>
identifier 'Partition'   [LeadingSpace] Loc=<./src/Qsort.c:1:5>
l_paren '('             Loc=<./src/Qsort.c:1:14>
int 'int'               Loc=<./src/Qsort.c:1:15>
identifier 'a'   [LeadingSpace] Loc=<./src/Qsort.c:1:19>
l_square '['            Loc=<./src/Qsort.c:1:20>
r_square ']'            Loc=<./src/Qsort.c:1:21>
comma ','               Loc=<./src/Qsort.c:1:22>
int 'int'       [LeadingSpace] Loc=<./src/Qsort.c:1:24>
identifier 'low'         [LeadingSpace] Loc=<./src/Qsort.c:1:28>
comma ','               Loc=<./src/Qsort.c:1:31>
int 'int'       [LeadingSpace] Loc=<./src/Qsort.c:1:33>
identifier 'high'        [LeadingSpace] Loc=<./src/Qsort.c:1:37>
r_paren ')'             Loc=<./src/Qsort.c:1:41>
l_brace '{'     [StartOfLine]  Loc=<./src/Qsort.c:2:1>
int 'int'       [StartOfLine] [LeadingSpace]   Loc=<./src/Qsort.c:3:5>
identifier 'key'         [LeadingSpace] Loc=<./src/Qsort.c:3:9>
equal '='       [LeadingSpace] Loc=<./src/Qsort.c:3:13>
identifier 'a'   [LeadingSpace] Loc=<./src/Qsort.c:3:15>
l_square '['            Loc=<./src/Qsort.c:3:16>
identifier 'low'                Loc=<./src/Qsort.c:3:17>
r_square ']'            Loc=<./src/Qsort.c:3:20>
semi ';'                Loc=<./src/Qsort.c:3:21>
```

  下图展示编译器识别出`Qsort.c`中第二个函数Qsort的函数头，其格式为`void Qsort(int a[], int low, int high)`，正好与以下字段对应

```
void 'void'      [StartOfLine]  Loc=<./src/Qsort.c:14:1>
identifier 'Qsort'       [LeadingSpace] Loc=<./src/Qsort.c:14:6>
l_paren '('             Loc=<./src/Qsort.c:14:11>
int 'int'               Loc=<./src/Qsort.c:14:12>
identifier 'a'   [LeadingSpace] Loc=<./src/Qsort.c:14:16>
l_square '['            Loc=<./src/Qsort.c:14:17>
r_square ']'            Loc=<./src/Qsort.c:14:18>
comma ','               Loc=<./src/Qsort.c:14:19>
int 'int'       [LeadingSpace] Loc=<./src/Qsort.c:14:21>
identifier 'low'         [LeadingSpace] Loc=<./src/Qsort.c:14:25>
comma ','               Loc=<./src/Qsort.c:14:28>
int 'int'       [LeadingSpace] Loc=<./src/Qsort.c:14:30>
identifier 'high'        [LeadingSpace] Loc=<./src/Qsort.c:14:34>
r_paren ')'             Loc=<./src/Qsort.c:14:38>
```

- 查看词法分析结果2

以文件`Qsort.c`为例，输入`clang ./src/Qsort.c  -Xclang -dump-raw-tokens`，输出较长，仅展示部分结果

下图展示编译器识别出`Qsort.c`中第一个函数Partition的函数头，其形式类似于之前的结果，但需要注意的是其中如`int`这种类型声明字段没有被直接识别，以及空格没有进行处理

```
raw_identifier 'int'     [StartOfLine]  Loc=<./src/Qsort.c:1:1>
unknown ' '             Loc=<./src/Qsort.c:1:4>
raw_identifier 'Partition'              Loc=<./src/Qsort.c:1:5>
l_paren '('             Loc=<./src/Qsort.c:1:14>
raw_identifier 'int'            Loc=<./src/Qsort.c:1:15>
unknown ' '             Loc=<./src/Qsort.c:1:18>
raw_identifier 'a'              Loc=<./src/Qsort.c:1:19>
l_square '['            Loc=<./src/Qsort.c:1:20>
r_square ']'            Loc=<./src/Qsort.c:1:21>
comma ','               Loc=<./src/Qsort.c:1:22>
unknown ' '             Loc=<./src/Qsort.c:1:23>
raw_identifier 'int'            Loc=<./src/Qsort.c:1:24>
unknown ' '             Loc=<./src/Qsort.c:1:27>
raw_identifier 'low'            Loc=<./src/Qsort.c:1:28>
comma ','               Loc=<./src/Qsort.c:1:31>
unknown ' '             Loc=<./src/Qsort.c:1:32>
raw_identifier 'int'            Loc=<./src/Qsort.c:1:33>
unknown ' '             Loc=<./src/Qsort.c:1:36>
raw_identifier 'high'           Loc=<./src/Qsort.c:1:37>
r_paren ')'             Loc=<./src/Qsort.c:1:41>
```

Qsort函数头类似，不再重复说明

- 查看语法分析结果

  以文件`Qsort.c`为例，输入`clang ./src/Qsort.c -Xclang -ast-dump`，输出较长，仅展示部分结果

  下图为生成结果的树的头部结构

```
TranslationUnitDecl 0x2b46b60 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x2b47058 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x2b46db0 '__int128'
|-TypedefDecl 0x2b470b8 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x2b46dd0 'unsigned __int128'
|-TypedefDecl 0x2b47148 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x2b47110 'char *'
|   `-BuiltinType 0x2b46bf0 'char'
|-TypedefDecl 0x2b473f8 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_l
ist_tag [1]'
| `-ConstantArrayType 0x2b473a0 'struct __va_list_tag [1]' 1
|   `-RecordType 0x2b47220 'struct __va_list_tag'
|     `-Record 0x2b47198 '__va_list_tag'
```

  下图展示经语法分析后产生的`Qsort.c`中第一个函数Partition对应的结构，可以看到其参数形式与源码相对应，函数中定义的局部变量key也在其中有对应形式

```
FunctionDecl 0x2b47710 <./src/Qsort.c:1:1, line:13:1> line:1:5 used Partition 'int (int *, int,
nt)'
|-ParmVarDecl 0x2b47500 <col:15, col:21> col:19 used a 'int *':'int *'
|-ParmVarDecl 0x2b47570 <col:24, col:28> col:28 used low 'int'
|-ParmVarDecl 0x2b475e0 <col:33, col:37> col:37 used high 'int'
`-CompoundStmt 0x2b93b10 <line:2:1, line:13:1>
  |-DeclStmt 0x2b930b0 <line:3:5, col:21>
  | `-VarDecl 0x2b47820 <col:5, col:20> col:9 used key 'int' cinit
  |   `-ImplicitCastExpr 0x2b93098 <col:15, col:20> 'int' <LValueToRValue>
  |     `-ArraySubscriptExpr 0x2b93070 <col:15, col:20> 'int' lvalue
  |       |-ImplicitCastExpr 0x2b93040 <col:15> 'int *':'int *' <LValueToRValue>
```

  下图展示经语法分析后产生的`Qsort.c`中第二个函数Qsort对应的结构，可以看到其参数形式与源码相对应

```
-FunctionDecl 0x2b93d80 <line:14:1, line:22:1> line:14:6 referenced Qsort 'void (int *, int, int
)'
  |-ParmVarDecl 0x2b93b68 <col:12, col:18> col:16 used a 'int *':'int *'
  |-ParmVarDecl 0x2b93bd8 <col:21, col:25> col:25 used low 'int'
  |-ParmVarDecl 0x2b93c48 <col:30, col:34> col:34 used high 'int'
  `-CompoundStmt 0x2b94508 <line:15:1, line:22:1>
    `-IfStmt 0x2b944d8 <line:16:5, line:21:5>
      |-<<<NULL>>>
      |-BinaryOperator 0x2b93ec0 <line:16:8, col:14> 'int' '<'
      | |-ImplicitCastExpr 0x2b93e90 <col:8> 'int' <LValueToRValue>
```

- 查看语法分析结果2

  以文件`Qsort.c`为例，输入`clang ./src/Qsort.c -Xclang -ast-view`

  会出现如下图所示报错

```
Stmt::viewAST is only available in debug builds on systems with Graphviz or gv
```

需要使用debug版本的clang才可以正常运行，记录一下自己的编译过程，注意编译需要较大的存储空间和较长的编译时间

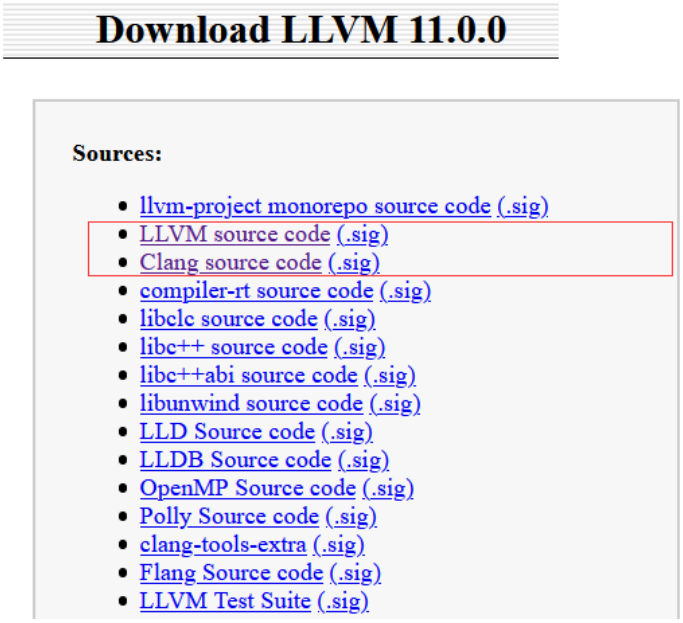
- 安装cmake，注意版本必须是3.13.4及以上

```
wget https://cmake.org/files/v3.13/cmake-3.13.4-Linux-x86_64.tar.gz
tar -xzvf cmake-3.13.4-Linux-x86_64.tar.gz

sudo mv cmake-3.13.0-Linux-x86_64 /opt/cmake-3.13.0

sudo ln -sf /opt/cmake-3.13.0/bin/*  /usr/bin/
```

- 从LLVM Download下载clang和llvm源码



  解压缩，分别将目录重命名为clang和llvm，将clang目录移动到llvm/tools/目录下
- 在llvm同级父目录下创建build目录，用于存放构建的中间产物和最终的可执行文件
- 由于编译过程需要较大的内存，需要扩大swap分区（此处取约20G）以构建较大的虚拟内存结构，否则会编译会报错并停止

```
sudo mkdir swapfile
cd /swapfile
sudo dd if=/dev/zero of=swap bs=1024 count=20000000
sudo mkswap -f  swap
sudo swapon swap
```

- 进入build目录，执行

  `cmake ../llvm -DLLVM_TARGETS_TO_BUILD=X86 -DCMAKE_BUILD_TYPE=Debug`
- 执行`make`即可开始编译，对于x核CPU，可以执行`make -jx`，该步骤需要较长时间
- 执行`sudo make install`进行安装

编译完成后重复执行命令，结果如下图所示，生成并打开了两个dot文件

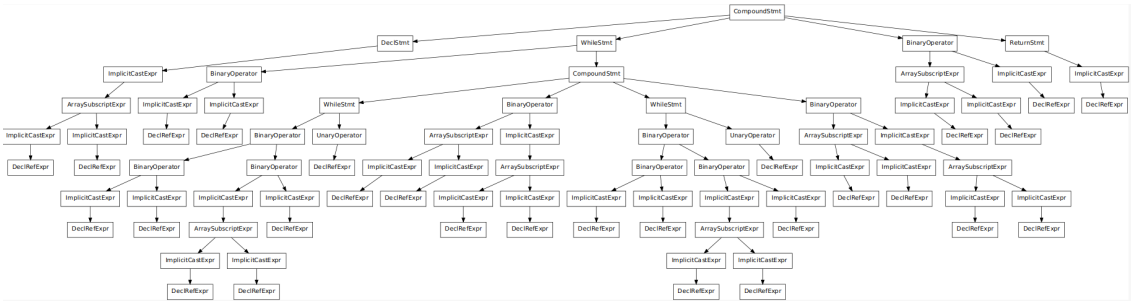
```
int Partition(int a[], int low, int high) {
    int key = a[low];
    while (low < high)
        {
            while (low < high && a[high] >= key)
                --high;
            a[low] = a[high];
            while (low < high && a[low] <= key)
                ++low;
            a[high] = a[low];
        }
    a[low] = key;
    return low;
}

Writing '/tmp/AST-76fae3.dot'...  done.
Trying 'xdg-open' program... Remember to erase graph file: /tmp/AST-76fae3.dot

void Qsort(int a[], int low, int high) {
    if (low < high) {
        int loc = Partition(a, low, high);
        Qsort(a, low, loc - 1);
        Qsort(a, loc + 1, high);
    }
}

Writing '/tmp/AST-b2d884.dot'...  done.
Trying 'xdg-open' program... Remember to erase graph file: /tmp/AST-b2d884.dot
```

下图为函数Partition对应的语法树结构图



- 查看编译优化的结果

  以文件`Qsort.c`为例，输入`clang ./src/Qsort.c -S -mllvm -print-after-all`

  输出非常长，这是因为经过了多次编译优化，只展示两个函数对应机器码的头部结构

```
Frame Objects:
  fi#-1: size=8, align=16, fixed, at location [SP-8]
  fi#0: size=8, align=8, at location [SP-16]
  fi#1: size=4, align=4, at location [SP-20]
  fi#2: size=4, align=4, at location [SP-24]
  fi#3: size=4, align=4, at location [SP-28]
  fi#4: size=1, align=1, at location [SP-29]
  fi#5: size=1, align=1, at location [SP-30]
Function Live Ins: %RDI, %ESI, %EDX

BB#0: derived from LLVM BB %0
    Live Ins: %RDI %ESI %EDX %RBP
        PUSH64r %RBP<kill>, %RSP<imp-def>, %RSP<imp-use>; flags: FrameSetup
        CFI_INSTRUCTION <call frame instruction>
```

```
Frame Objects:
  fi#-1: size=8, align=16, fixed, at location [SP-8]
  fi#0: size=8, align=8, at location [SP-16]
  fi#1: size=4, align=4, at location [SP-20]
  fi#2: size=4, align=4, at location [SP-24]
  fi#3: size=4, align=4, at location [SP-28]
Function Live Ins: %RDI, %ESI, %EDX

BB#0: derived from LLVM BB %0
    Live Ins: %RDI %ESI %EDX %RBP
        PUSH64r %RBP<kill>, %RSP<imp-def>, %RSP<imp-use>; flags: FrameSetup
```

- 查看生成的目标代码结果

  以文件`Qsort.c`为例，输入`clang ./src/Qsort.c -S`，生成文件`Qsort.s`,内容较长，仅展示部分

  下图展示`Qsort.c`中第一个函数Partition对应的汇编代码的头部结构和部分汇编语言

```
        .text
        .file   "./src/Qsort.c"
        .globl  Partition
        .align  16, 0x90
        .type   Partition,@function
Partition:                              # @Partition
        .cfi_startproc
# BB#0:
        pushq   %rbp
.Ltmp0:
        .cfi_def_cfa_offset 16
.Ltmp1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
.Ltmp2:
        .cfi_def_cfa_register %rbp
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movslq  -12(%rbp), %rdi
        movq    -8(%rbp), %rax
        movl    (%rax,%rdi,4), %edx
        movl    %edx, -20(%rbp)
```

下图展示`Qsort.c`中第二个函数Qsort对应的汇编代码的头部结构和部分汇编语言

```
        .globl  Qsort
        .align  16, 0x90
        .type   Qsort,@function
Qsort:                                  # @Qsort
        .cfi_startproc
# BB#0:
        pushq   %rbp
.Ltmp3:
        .cfi_def_cfa_offset 16
.Ltmp4:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
.Ltmp5:
```

可以看出其与gcc编译生成的汇编代码在部分特征上的相似于不同

各优化等级下编译程序运行效率测试

采用linux自带的`date`命令记录程序运行的起止时间，并编写脚本计算各语言程序运行10次后的平均运行时间，以下为脚本，为方便显示时间结果，使用`> /dev/null 2>&1`略去每次运行程序的输出

```
#!/bin/bash

function get_average_time()
{
    start=$(date +%s%N)
    for i in {1..5}
    do
        eval $1 > /dev/null 2>&1
    done
    end=$(date +%s%N)
    echo 运行5次,每次平均用时:  $((($end - $start) /5000000 ))毫秒
}

echo GCC O0
gcc -O0 ./src/main.c ./src/Qsort.c -o ./bin/main
cd ./bin
get_average_time './main'
cd ../

echo GCC O1
gcc -O1 ./src/main.c ./src/Qsort.c -o ./bin/main
cd ./bin
get_average_time './main'
cd ../

echo GCC O2
```

```
gcc -O2 ./src/main.c ./src/Qsort.c -o ./bin/main
cd ./bin
get_average_time './main'
cd ../

echo GCC O3
gcc -O3 ./src/main.c ./src/Qsort.c -o ./bin/main
cd ./bin
get_average_time './main'
cd ../

echo LLVM O0
clang -O0 ./src/main.c ./src/Qsort.c -o ./bin/main
cd ./bin
get_average_time './main'
cd ../

echo LLVM O1
clang -O1 ./src/main.c ./src/Qsort.c -o ./bin/main
cd ./bin
get_average_time './main'
cd ../

echo LLVM O2
clang -O2 ./src/main.c ./src/Qsort.c -o ./bin/main
cd ./bin
get_average_time './main'
cd ../

echo LLVM O3
clang -O3 ./src/main.c ./src/Qsort.c -o ./bin/main
cd ./bin
get_average_time './main'
cd ../
```

该脚本在本次实验中命名为`show_time.sh`，运行时的文件结构参考如下

```
$ tree
.
├── bin
│   ├── main
│   ├── main.o
│   └── Qsort.o
├── data
│   ├── DataGenerate.cpp
│   └── in.txt
├── show_time.sh
└── src
    ├── main.c
    └── Qsort.c
```

运行结果如下所示

```
# ssr @ ubuntu in ~/compile-lab/lab2 [23:55:55]
$ ./show_time.sh
GCC O0
运行10次,每次平均用时:  1839毫秒
GCC O1
运行10次,每次平均用时:  1371毫秒
GCC O2
运行10次,每次平均用时:  1304毫秒
GCC O3
运行10次,每次平均用时:  1413毫秒
LLVM O0
运行10次,每次平均用时:  1938毫秒
LLVM O1
运行10次,每次平均用时:  1300毫秒
LLVM O2
运行10次,每次平均用时:  1220毫秒
LLVM O3
运行10次,每次平均用时:  1296毫秒
```

程序运行效率分析

在之前的实验过程中，GCC编译器在各优化等级下编译程序的运行时间如下所示

优化等级	平均运行时间（ms）
O0	1839
O1	1371
O2	1304
O3	1413

LLVM编译器在各优化等级下编译程序的运行时间如下所示

优化等级	平均运行时间（ms）
O0	1938
O1	1300
O2	1220
O3	1296

分析可推测对于测试程序来说有如下结论

- 对于GCC和LLVM编译器，开启优化后编译程序的运行效率均有明显的提升，但O1、O2、O3各级优化之间效率差别不是非常明显
- 对于GCC和LLVM编译器，O2优化较O1优化，程序效率均略有提升
- 对于GCC和LLVM编译器，O3优化较O2优化，程序效率均略有下降，与期望不符，可能是程序编写问题
- GCC相较于LLVM编译器，在优化前程序效率较高，但在同等级的优化后程序效率较低

实验心得体会

这次实验总体感觉就是不是很顺利，前置知识太少，在只是简单了解了gcc和llvm编译器结构的前提下整个实验分析过程都像是盲人摸象，实验说明也没给出足够详细的指导，花费了大量时间摸索实验的知识和环境配置，走了不少弯路，但收获却感觉不是很多，如果能在获取足够理论知识后再进行这一实验想必能印证所学，实在是有些遗憾。