编译原理预习作业

课号：0994　学号：2013921　姓名：周延霖

一、C 程序优化

题目：

```
/* LOOP #1 */
for (i = 0; i < N; i++) {
    a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
}

/* LOOP #2 */
b = a;
for (i = 0; i < N; i++) {
    *b = *b * 2000;
    *b = *b / 10000;
    b++;
}
```

- 为了获得运行更快的目标程序，你选择哪种编程方式？

解答：

具体代码如下：

wxn1 〉 wxn1 〉 C main 〉 main()

```c
#include<stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{
    int max = 100000;
    printf("以下程序规模大小为%d\n", max);
    int a[max];
    clock_t start, finish;
    double Total_time;
    for(int i = 0; i < max;i++)
    {
        a[i] = 1;
    }
    start = clock();
    for(int i = 0; i < max;i++)
    {
        a[i] = a[i] * 2000;
        a[i] = a[i] / 10000;
    }
    finish = clock();
    Total_time = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("第一个程序共执行了%f seconds\n", Total_time);
    int* b;
    b = a;
    start = clock();
    for(int i = 0; i < max;i++)
    {
        *b = *b * 2000;
        *b = *b / 10000;
        b++;
    }
    finish = clock();
    Total_time = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("第二个程序共执行了%f seconds\n", Total_time);
}
```

根据以上代码，通过改 max 值来更改不同的空间范围，来看不同的 max 值对应的不同的规模大小来得出的不同时间：

1.首先是在 mac 系统下的 Xcode 软件下的 Clang 13.1.6 编译器运行不同程序规模所运行的不同时间，以下同一第一个程序为下标寻址，第二个程序为指针寻址：

```
以下程序规模大小为50
第一个程序共执行了0.000004 seconds
第二个程序共执行了0.000001 seconds
Program ended with exit code: 0
```

```
以下程序规模大小为100
第一个程序共执行了0.000004 seconds
第二个程序共执行了0.000000 seconds
Program ended with exit code: 0
```

```
以下程序规模大小为1000
第一个程序共执行了0.000015 seconds
第二个程序共执行了0.000006 seconds
Program ended with exit code: 0
```

```
以下程序规模大小为10000
第一个程序共执行了0.000038 seconds
第二个程序共执行了0.000033 seconds
Program ended with exit code: 0
```

```
以下程序规模大小为100000
第一个程序共执行了0.000329 seconds
第二个程序共执行了0.000384 seconds
Program ended with exit code: 0
```

```
以下程序规模大小为1000000
第一个程序共执行了0.003365 seconds
第二个程序共执行了0.003446 seconds
Program ended with exit code: 0
```

由于发现这样测试更改起来非常的麻烦，并且每次编辑也非常浪费时间，精度和细度的要求也不够，所以将源程序增加的 while 循环，使得其可以自动运行多次程序，while 循环体如下所示：

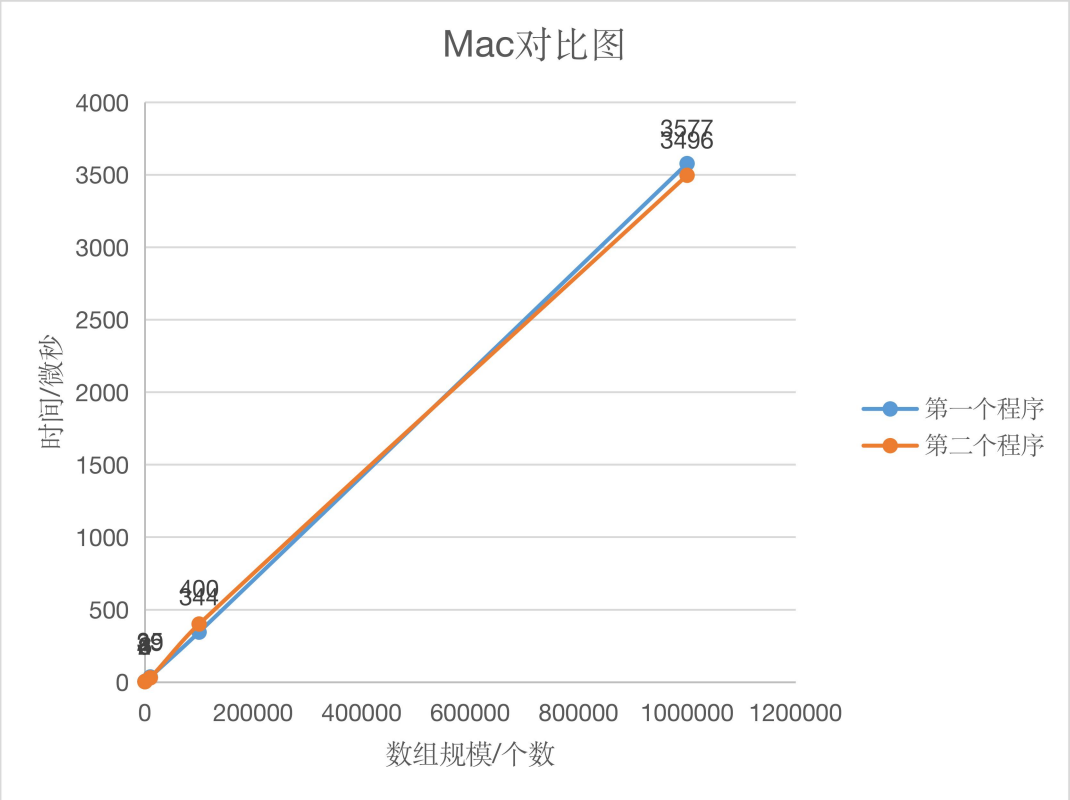
```
while(n <= max)
{
    printf("以下程序规模大小为%d\n", n);
    start = clock();
    for(int i = 0; i < n;i++)
    {
        a[i] = a[i] * 2000;
        a[i] = a[i] / 10000;
    }
    finish = clock();
    Total_time = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("第一个程序共执行了%f seconds\n", Total_time);
    int* b;
    b = a;
    start = clock();
    for(int i = 0; i < n;i++)
    {
        *b = *b * 2000;
        *b = *b / 10000;
        b++;
    }
    finish = clock();
    Total_time = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("第二个程序共执行了%f seconds\n", Total_time);
    if(n < 100) n += 10;
    else if(n < 1000) n += 100;
    else n *= 10;
}
```

运行时间如下图所示：

```
以下程序规模大小为10
第一个程序共执行了0.000003 seconds
第二个程序共执行了0.000000 seconds
以下程序规模大小为20
第一个程序共执行了0.000002 seconds
第二个程序共执行了0.000001 seconds
以下程序规模大小为30
第一个程序共执行了0.000002 seconds
第二个程序共执行了0.000000 seconds
以下程序规模大小为40
第一个程序共执行了0.000002 seconds
第二个程序共执行了0.000000 seconds
以下程序规模大小为50
第一个程序共执行了0.000000 seconds
第二个程序共执行了0.000000 seconds
以下程序规模大小为60
第一个程序共执行了0.000002 seconds
第二个程序共执行了0.000000 seconds
以下程序规模大小为70
第一个程序共执行了0.000001 seconds
第二个程序共执行了0.000002 seconds
以下程序规模大小为80
第一个程序共执行了0.000001 seconds
第二个程序共执行了0.000002 seconds
以下程序规模大小为90
第一个程序共执行了0.000001 seconds
第二个程序共执行了0.000001 seconds
以下程序规模大小为100
第一个程序共执行了0.000000 seconds
第二个程序共执行了0.000002 seconds
以下程序规模大小为200
第一个程序共执行了0.000002 seconds
第二个程序共执行了0.000002 seconds
```

```
以下程序规模大小为300
第一个程序共执行了0.000003 seconds
第二个程序共执行了0.000002 seconds
以下程序规模大小为400
第一个程序共执行了0.000002 seconds
第二个程序共执行了0.000002 seconds
以下程序规模大小为500
第一个程序共执行了0.000003 seconds
第二个程序共执行了0.000002 seconds
以下程序规模大小为600
第一个程序共执行了0.000002 seconds
第二个程序共执行了0.000002 seconds
以下程序规模大小为700
第一个程序共执行了0.000003 seconds
第二个程序共执行了0.000003 seconds
以下程序规模大小为800
第一个程序共执行了0.000002 seconds
第二个程序共执行了0.000003 seconds
以下程序规模大小为900
第一个程序共执行了0.000005 seconds
第二个程序共执行了0.000004 seconds
以下程序规模大小为1000
第一个程序共执行了0.000006 seconds
第二个程序共执行了0.000006 seconds
以下程序规模大小为10000
第一个程序共执行了0.000035 seconds
第二个程序共执行了0.000029 seconds
以下程序规模大小为100000
第一个程序共执行了0.000344 seconds
第二个程序共执行了0.000400 seconds
以下程序规模大小为1000000
第一个程序共执行了0.003577 seconds
第二个程序共执行了0.003496 seconds
```

多次测量以上数据，并取数据的平均值可以画出如下的折线图：

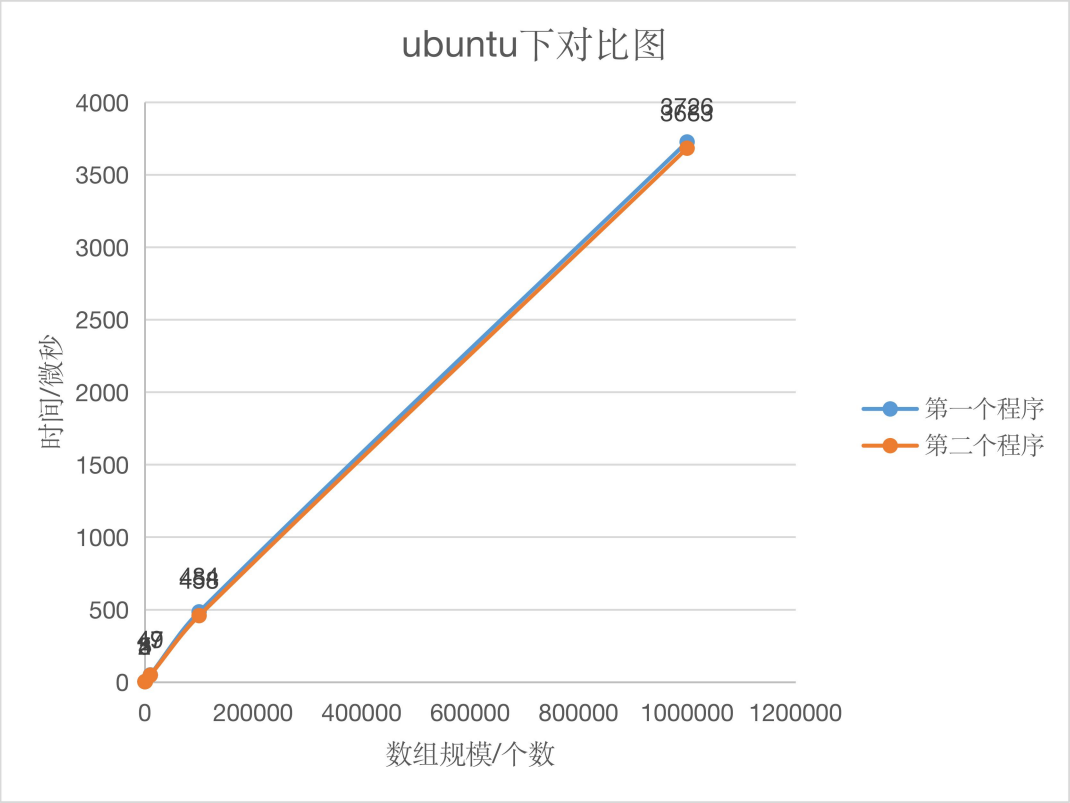


2.接下来由于要在不同的系统下进行相应的测试，所以接下来用 ubuntu 下用 gcc 10.1.0 程序来测试编写好的程序，编译方法如下：

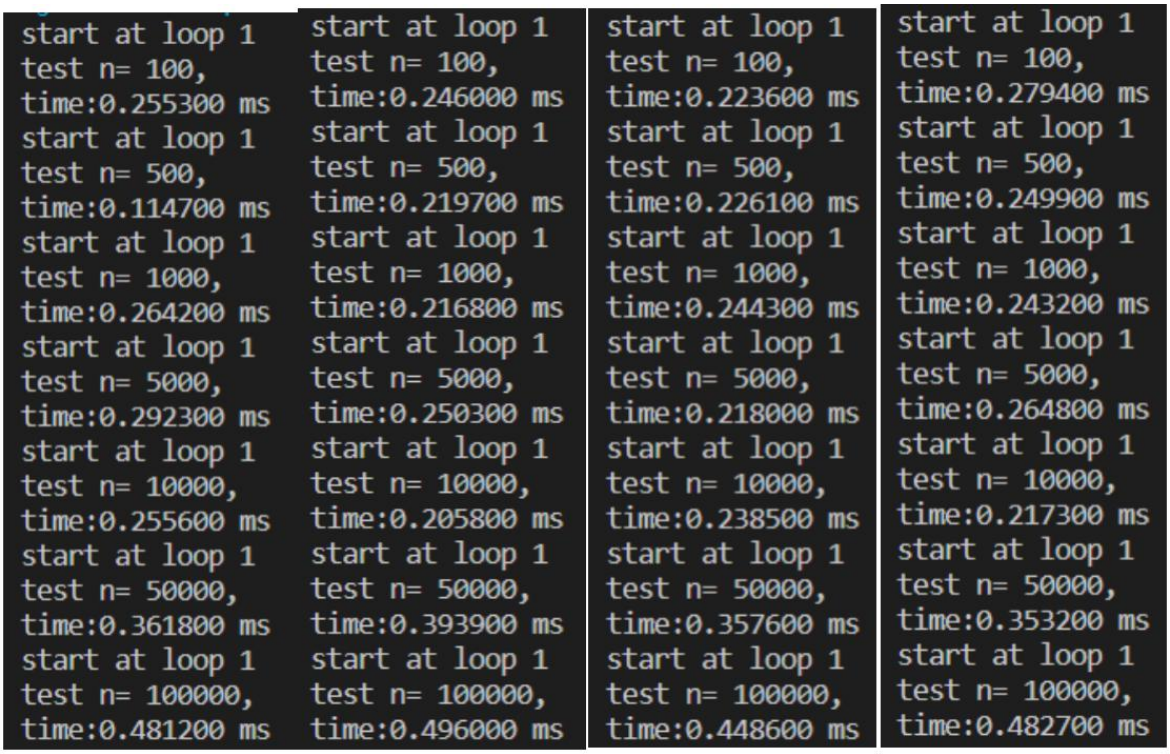
```
zhouyanlin@ubuntu:~/Desktop/splint$ gcc -o zyl zyl.c
zhouyanlin@ubuntu:~/Desktop/splint$ ./zyl
以下程序规模大小为10
```

所得到的输出如下：

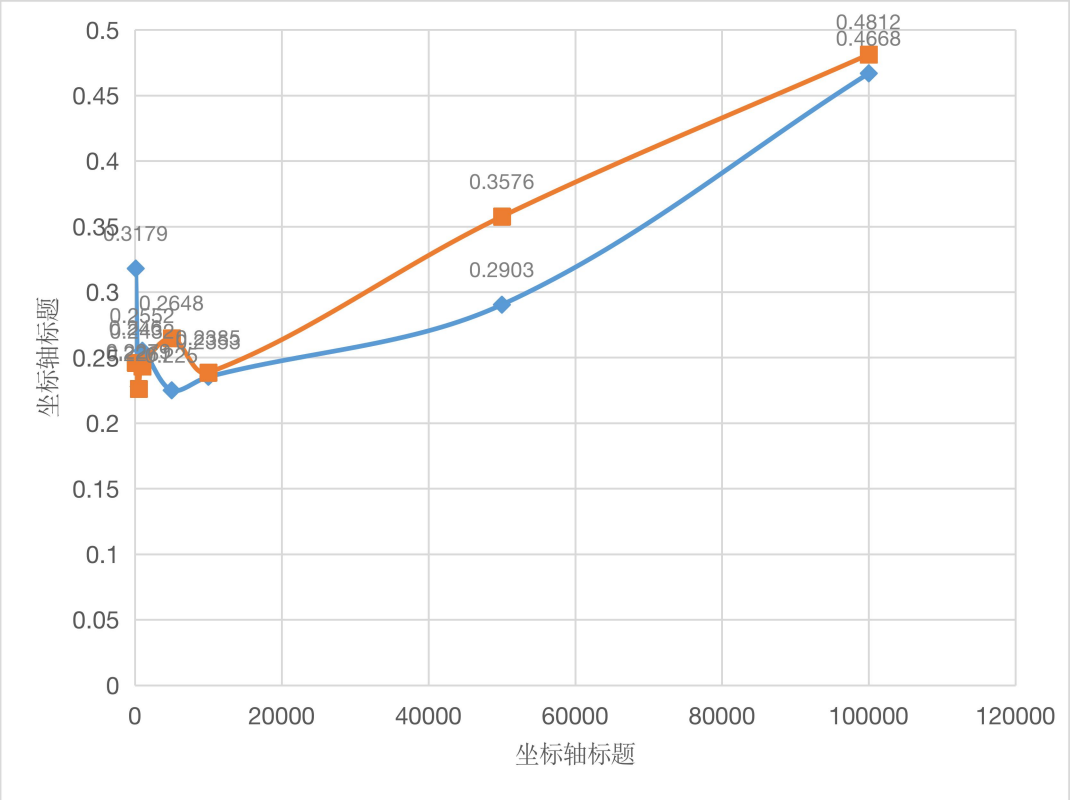
多次测量以上数据，并取数据的平均值可以画出如下的折线图：



3.由于想知道不同的操作系统的模式下运行的差别，借助同学的电脑 Windows 系统下来运行，visual studio 软件下的 gcc 8.1.0 下运行的结果如下：



	n=100	n=1000	n=10000	100000
数组下标	1	2	3	4
指针	2	3	4	5



4.总结与分析：

由以上在三个系统下对不同规模下的程序进行运行比较可以看出：

- 在ubuntu下的gcc相比本机mac系统下的Xcode运行慢一些，但是我认为可能是因为虚拟机本身运行时跟底层硬件交互途径就要比本机长，所以如果本机操作系统就是ubuntu的话，这两种编译器应该是差不多的；
- 在windows下的运行结果的大概趋势与ubuntu和Mac下基本上是相同的，但总体时间上并没有在Mac下的程序运行速度快；
- 对不同规模的运行结果可以看出，不论是在哪个系统下，对于小规模（100以下），用数组或者是替代指针的方法快慢不定；当上升一定规模后，可以看出指针比较快，但有时候可能有小概率事

件使得数组会较快一点，所以可以得出一个初步的结论是编译器可能会对指针有一定的优化。

二、分词，构造语法树

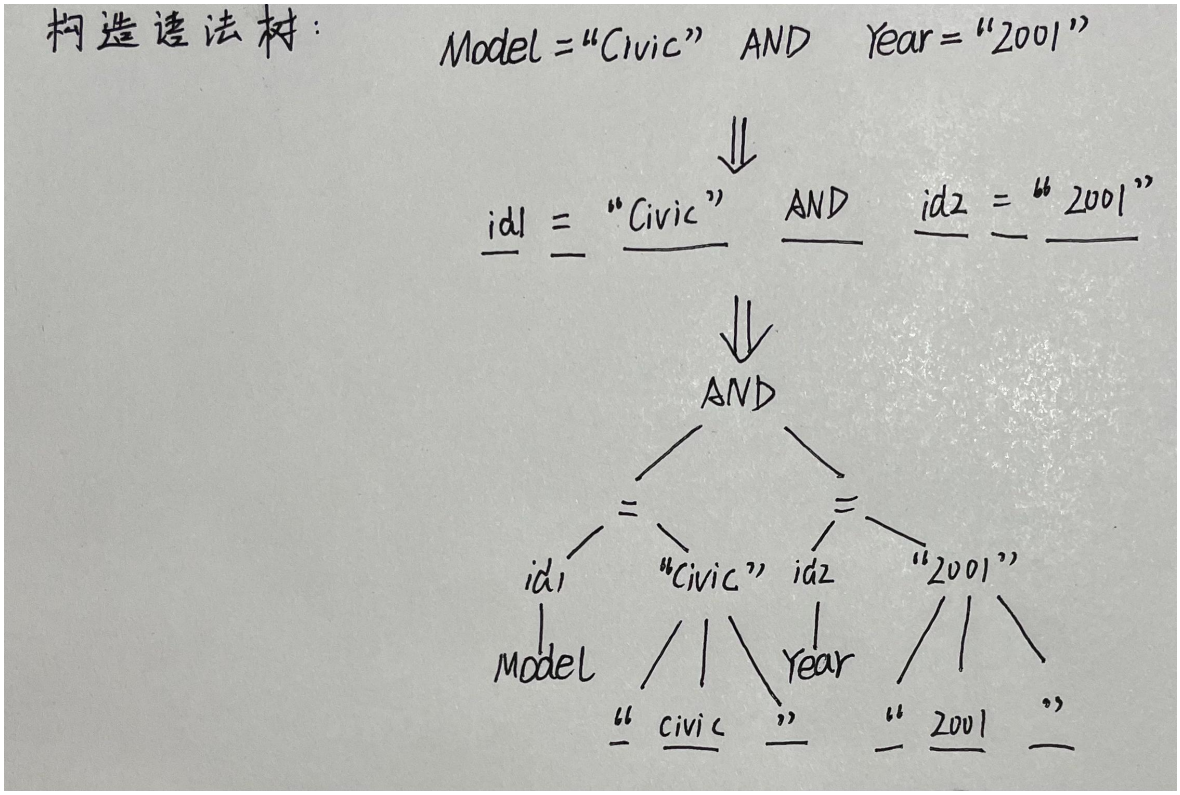
题目：

Model=“Civic” AND Year=“2001”

ID#	Model	Year	Color	Dealer	Price
6734	Civic	2001	White	OR	$17,000
4395	Civic	2001	Red	CA	$17,000

对上图的数据库查询表达式进行分词、构造语法树

解答：



三、静态检查

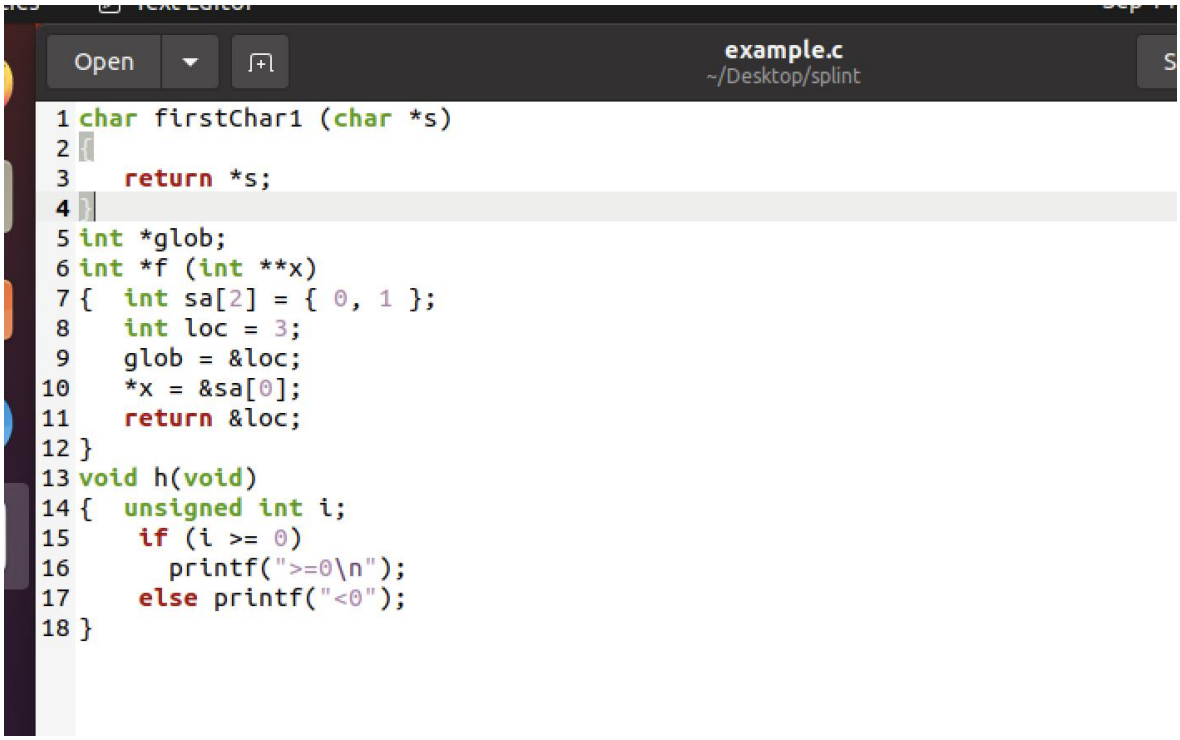
题目：

```
wxn1 〉 wxn1 〉 C main 〉 h()
1  char firstChar1 (char *s)
2  {
3     return *s;
4  }
5  int *glob;
6  int *f (int **x)
7  {  int sa[2] = { 0, 1 };
8     int loc = 3;
9     glob = &loc;
10    *x = &sa[0];
11    return &loc;          Address of stack memory associated with local variable 'loc' returned
12 }
13 void h(void)
14 {
15     unsigned int i;
16     if (i >= 0)          Variable 'i' is uninitialized when used here
17       printf(">=0\n");   Implicitly declaring library function 'printf' with type 'int (const char *, ...)'
18     else printf("<0");
19 }
```

检查上述代码的错误

解答：

首先在 ubuntu 下编写代码如下图所示：



接下来在虚拟机下下载 splint 静态检测工具，并输入 splint example.c 来检测代码，结果如下所示：

```
zhouyanlin@ubuntu:~/Desktop/splint$ splint example.c
Splint 3.1.2 --- 20 Feb 2018

example.c: (in function f)
example.c:11:11: Stack-allocated storage &loc reachable from return value: &loc
  A stack reference is pointed to by an external reference when the function
  returns. The stack-allocated storage is destroyed after the call, leaving a
  dangling reference. (Use -stackref to inhibit warning)
example.c:11:11: Immediate address &loc returned as implicitly only: &loc
  An immediate address (result of & operator) is transferred inconsistently.
  (Use -immediatetrans to inhibit warning)
example.c:11:16: Stack-allocated storage *x reachable from parameter x
   example.c:10:4: Storage *x becomes stack-allocated storage
example.c:11:16: Function returns with global glob referencing released storage
  A global variable does not satisfy its annotations when control is
  transferred. (Use -globstate to inhibit warning)
   example.c:11:11: Storage glob released
example.c: (in function h)
example.c:15:9: Comparison of unsigned value involving zero: i >= 0
  An unsigned value is used in a comparison with zero in a way that is either a
  bug or confusing. (Use -unsignedcompare to inhibit warning)
example.c:15:9: Variable i used before definition
  An rvalue is used that may not be initialized to a value on some execution
  path. (Use -usedef to inhibit warning)
example.c:5:6: Variable exported but not used outside example: glob
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (Use -exportlocal to inhibit warning)

Finished checking --- 7 code warnings
zhouyanlin@ubuntu:~/Desktop/splint$
```

其实在 Xcode 下输入此代码也可以看到会出现两个警告和一个错误如题目中的图片所示。

通过静态检查工具以及 Mac 下的 Xcode 的报错，结合自己的思考，我认为有以下几个错误：

- 首先，&loc 与一个外部指针相关，但这个区域在返回后就被摧毁了，会导致外部指针指向一个危险的区域；
- 当控制权从函数转移之后得到的全局变量将是不理想的；
- 将不确定大小的 i 与 0 进行比较；
- 引用了外部不存在的声明；

四、语法描述

题目：

- 在变量声明中，允许任意多个标识符形成的标识符列表，如下所示，（用自然语言）描述标识符列表的递归定义（见第6、7页）

  int a, b, …, x;

解答：

递归定义：

1. int 是标识符；
2. 字符是变量表达式；
3. 若 $expression_1$ 和 $expression_2$ 是表达式，
   则 $expression_1$ ， $expression_2$ 是表达式；（得去除空格）
4. “;” 是结束符。