Assignment(徐子翔-2022141470095-assignment1)

1 docker 环境配置

1.1 docker的安装和运行

在完成基本的安装和配置后，运行hello-world以得到成功安装的反馈

```
C:\Users\徐子翔>docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

再运行docker images以及docker ps 得到容器的hash值

```
C:\Users\徐子翔> docker images
REPOSITORY                TAG       IMAGE ID       CREATED         SIZE
scucpphw_test_img         latest    268ef1e1425c   43 hours ago    1.3GB
docker101tutorial         latest    368b799d82ba   44 hours ago    47MB
xzx34/docker101tutorial   latest    368b799d82ba   44 hours ago    47MB
alpine/git                latest    22d84a66cda4   3 months ago    43.6MB
hello-world               latest    feb5d9fea6a5   17 months ago   13.3kB
```

```
C:\Users\徐子翔> docker ps
CONTAINER ID   IMAGE                      COMMAND       CREATED        STATUS         PORTS     NAMES
8fd9ee166d63   scucpphw_test_img:latest   "/bin/bash"   43 hours ago   Up 7 minutes   22/tcp    youthful_maxwell
```

1.2 实现容器的对应、进入docker以及Linux文件操作

```
C:\Users\徐子翔> docker exec -it 8fd9ee166d63 /bin/bash
root@8fd9ee166d63:/# pwd
/
root@8fd9ee166d63:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var  ws
root@8fd9ee166d63:/# cd ws
root@8fd9ee166d63:/ws# ls
code
root@8fd9ee166d63:/ws# cd code
root@8fd9ee166d63:/ws/code# ls
11_operators       1_helloworld    3_streams       4_containers          6_classes_2         8_template_functions
13_move_semantics  2_typesStructs  3pt5_initsRefs  5_iterators_pointers  7_template_classes  9_lambdas
root@8fd9ee166d63:/ws/code#
```

重点复习的LInux下操作：

cd xx　　进入

cd ..　　退回上一级目录

ls　　显示当前目录的内容

pwd　　显示当前路径

2 利用g++编译和运行文件

2.1 编译test.cpp文件

解构编译指令的过程，实现分步编译

```
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# g++ test.cpp
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# ls
a.out  inefficency.cpp  other_test  test.cpp
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# g++ -E test.cpp -o test.i
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# ls
a.out  inefficency.cpp  other_test  test.cpp  test.i
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# g++ -S test.i -o test.s
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# ls
a.out  inefficency.cpp  other_test  test.cpp  test.i  test.s
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# g++ -c test.s -o test.o
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# lls
bash: lls: command not found
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# ls
a.out  inefficency.cpp  other_test  test.cpp  test.i  test.o  test.s
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3#  g++ test.o -o test
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# ls
a.out  inefficency.cpp  other_test  test  test.cpp  test.i  test.o  test.s
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3#
```

运行两种可执行文件，得到运行结果

```
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# ./a.out
vector v after call to generate_n() with lambda: 1 1 2 3 5 8 13 21 34
x: 1 y: 1
vector v after 1st call to fillVector(): 1 2 3 4 5 6 7 8 9
vector v after 2nd call to fillVector(): 10 11 12 13 14 15 16 17 18
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# ./test
vector v after call to generate_n() with lambda: 1 1 2 3 5 8 13 21 34
x: 1 y: 1
vector v after 1st call to fillVector(): 1 2 3 4 5 6 7 8 9
vector v after 2nd call to fillVector(): 10 11 12 13 14 15 16 17 18
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3#
```

2.2 使用time指令研究编译优化情况

根据是否使用O2优化进行编译，得到两个可执行文件

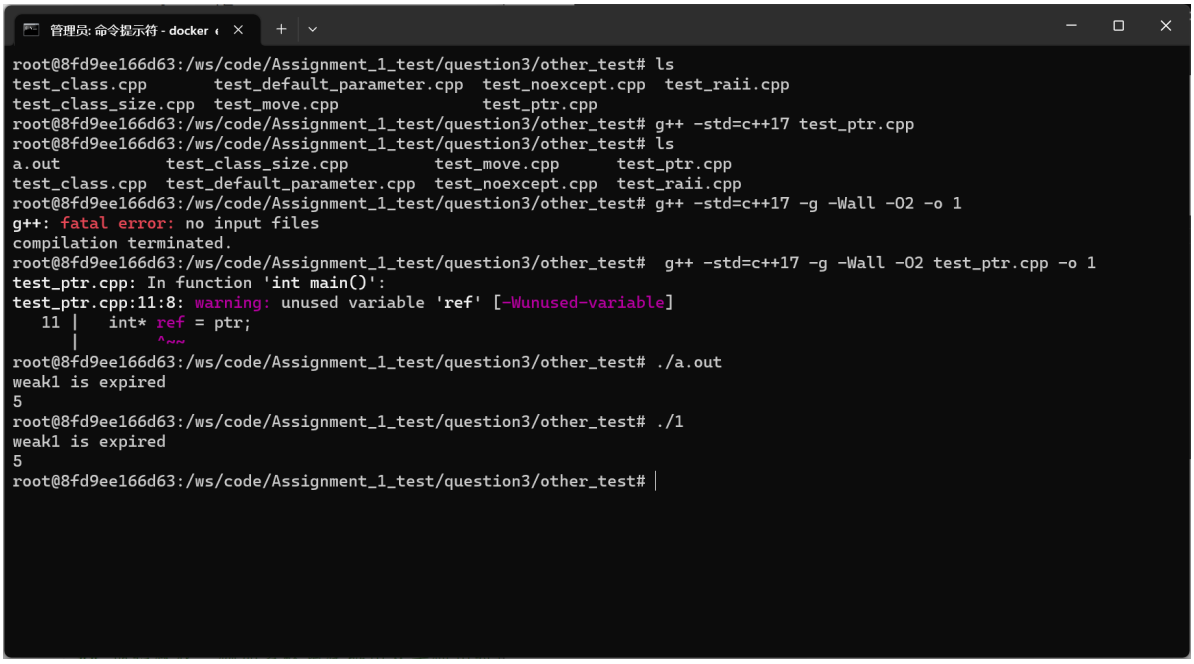
使用time函数研究两个可执行文件的运行情况

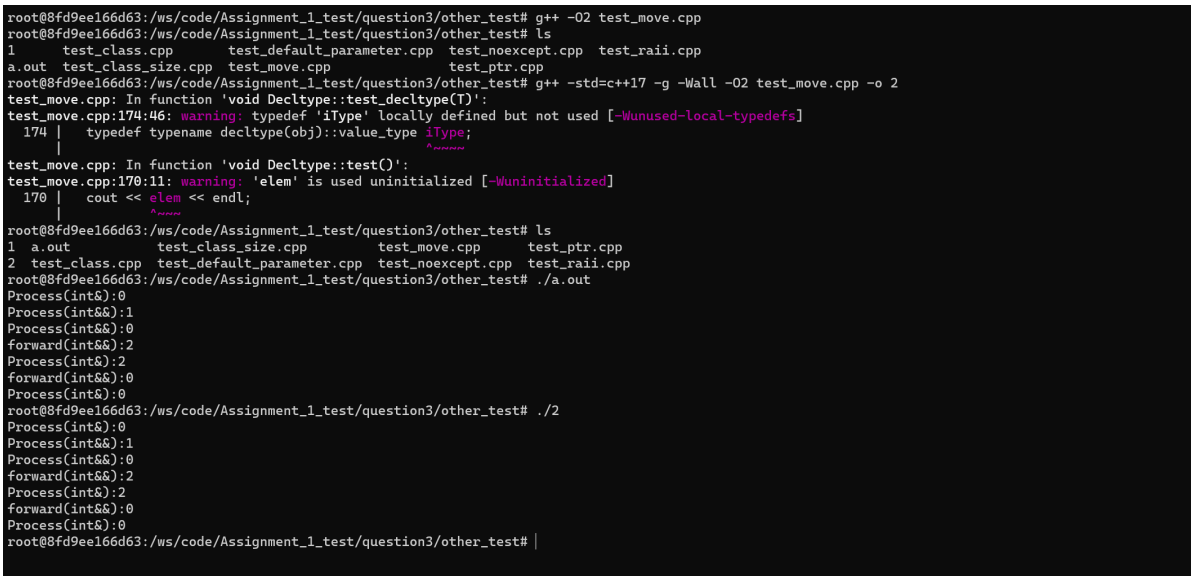
```
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# g++ inefficency.cpp -o without_o.out
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# g++ inefficency.cpp -O2 -o with_o.out
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# time ./with_o.out
result = 100904034

real    0m0.004s
user    0m0.002s
sys     0m0.000s
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# ./without_o.out
result = 100904034
root@8fd9ee166d63:/ws/code/Assignment_1_test/question3# time ./without_o.out
result = 100904034

real    0m2.419s
user    0m2.410s
sys     0m0.000s
```

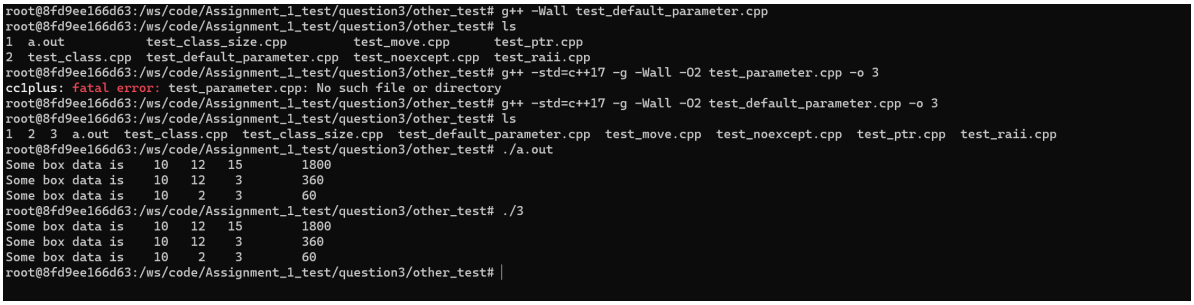
可以发现，O2优化开关极大地提升了代码的效率

事实上，对于大量调用函数以及使用了大量循环嵌套的算法，O2都能起到良好的效果

2.3 使用不同的g++参数进行编译







3 利用GDB调试文件

3.1 追踪函数sumOfSquare()的的栈帧与层级关系

```
(gdb) l
1       #include <iostream>
2       using namespace std;
3
4       int sumOfSquare(int a, int b) {
5               return a * a + b * b;
6       }
7
8       double sumOfSquare(double a, double b) {
9               return a * a + b * b;
10      }
(gdb)
11
12      int main() {
13              int m, n;
14              cout << "Enter two integer: ";
15              cin >> m >> n;
16              cout << "Their sum of square: " << sumOfSquare(m, n) << endl;
17
18              double x, y;
19              cout << "Enter two real number: ";
20              cin >> x >> y;
(gdb) b 4
Breakpoint 1 at 0x4011c0: file test_function_overload.cpp, line 5.
(gdb) b 8
Breakpoint 2 at 0x4011e0: file test_function_overload.cpp, line 9.
(gdb) run
Starting program: /ws/code/Assignment_1_test/question4/1
warning: Error disabling address space randomization: Operation not permitted
warning: File "/usr/local/lib64/libstdc++.so.6.0.29-gdb.py" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /usr/local/lib64/libstdc++.so.6.0.29-gdb.py
line to your configuration file "/root/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/root/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
Enter two integer: 3 4
```

```
(gdb) b 4
Breakpoint 1 at 0x4011c0: file test_function_overload.cpp, line 5.
(gdb) b 8
Breakpoint 2 at 0x4011e0: file test_function_overload.cpp, line 9.
(gdb) run
Starting program: /ws/code/Assignment_1_test/question4/1
warning: Error disabling address space randomization: Operation not permitted
warning: File "/usr/local/lib64/libstdc++.so.6.0.29-gdb.py" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /usr/local/lib64/libstdc++.so.6.0.29-gdb.py
line to your configuration file "/root/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/root/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
Enter two integer: 3 4

Breakpoint 1, sumOfSquare (a=3, b=4) at test_function_overload.cpp:5
5               return a * a + b * b;
(gdb) step
6       }
(gdb) step
Their sum of square: 25
main () at test_function_overload.cpp:19
19              cout << "Enter two real number: ";
(gdb) step
20              cin >> x >> y;
(gdb) step
Enter two real number: 1.1 0.2
21              cout << "Their sum of square: " << sumOfSquare(x, y) << endl;
(gdb) step

Breakpoint 2, sumOfSquare (a=1.1000000000000001, b=0.20000000000000001) at test_function_overload.cpp:9
9               return a * a + b * b;
(gdb) step
10      }
(gdb) step
Their sum of square: 1.25
main () at test_function_overload.cpp:23
23              return 0;
(gdb) step
24      }
```

3.2 追踪y的变化值和j的含义

这是一个相对繁琐的任务，在此之前我们先复习一下基本的gdb命令

break	b	设置断点,程序运行到断点的位置会停下来
info	i	描述程序的状态
run	r	开始运行程序
display	disp	跟踪查看某个变量,每次停下来都显示它的值
step	s	执行下一条语句,如果该语句为函数调用,则进入函数执行其中的第一条语句
next	n	执行下一条语句,如果该语句为函数调用,不会进入函数内部执行(即不会一步步地调试函数内部语句)
print	p	打印内部变量值

continue	c	继续程序的运行,直到遇到下一个断点
set var name=v		设置变量的值
start	st	开始执行程序,在main函数的第一条语句前面停下来
file		装入需要调试的程序
kill	k	终止正在调试的程序
watch		监视变量值的变化
backtrace	bt	产看函数调用信息(堆栈)
frame	f	查看栈帧
quit	q	退出GDB环境

首先阅读代码，在每个循环开始处设置断点

```
(gdb) l
1       // 基于范围的for循环来循环访问数组和矢量
2       // range-based-for.cpp
3       #include <iostream>
4       #include <vector>
5       using namespace std;
6
7       int main()
8       {
9           int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10
(gdb)
11          for( int y : x ) { // Access by value using a copy declared as a specific type.
12                             // Not preferred.
13              cout << y << " ";
14          }
15          cout << endl;
16
17          for( auto y : x ) { // Copy of 'x', almost always undesirable
18              cout << y << " ";
19          }
20          cout << endl;
(gdb)
21
22          for( auto &y : x ) { // Type inference by reference.
23              cout << y << " ";
24          }
25          cout << endl;
26
27          for( const auto &y : x ) { // Type inference by const reference.
28              cout << y << " ";
29          }
30          cout << endl;
(gdb) b 11
Breakpoint 1 at 0x401286: file test_range_based.cpp, line 11.
(gdb) b 17
Breakpoint 2 at 0x4012f0: file test_range_based.cpp, line 17.
(gdb) b 22
Breakpoint 3 at 0x40135a: file test_range_based.cpp, line 22.
(gdb) b 27
Breakpoint 4 at 0x4013c6: file test_range_based.cpp, line 27.
(gdb) l
31          cout << "end of integer array test" << endl;
32          cout << endl;
33
34          vector<double> v;
35          for (int i = 0; i < 10; ++i) {
36              v.push_back(i + 0.14159);
37          }
```

在第一个循环中，y是int类型，遍历了数组x中的所有元素

在第二个循环中，auto y实质上也是int类型，与第一个循环一致

在第三个循环中，y是一个取地址，包括了数组元素的地址信息和数值信息

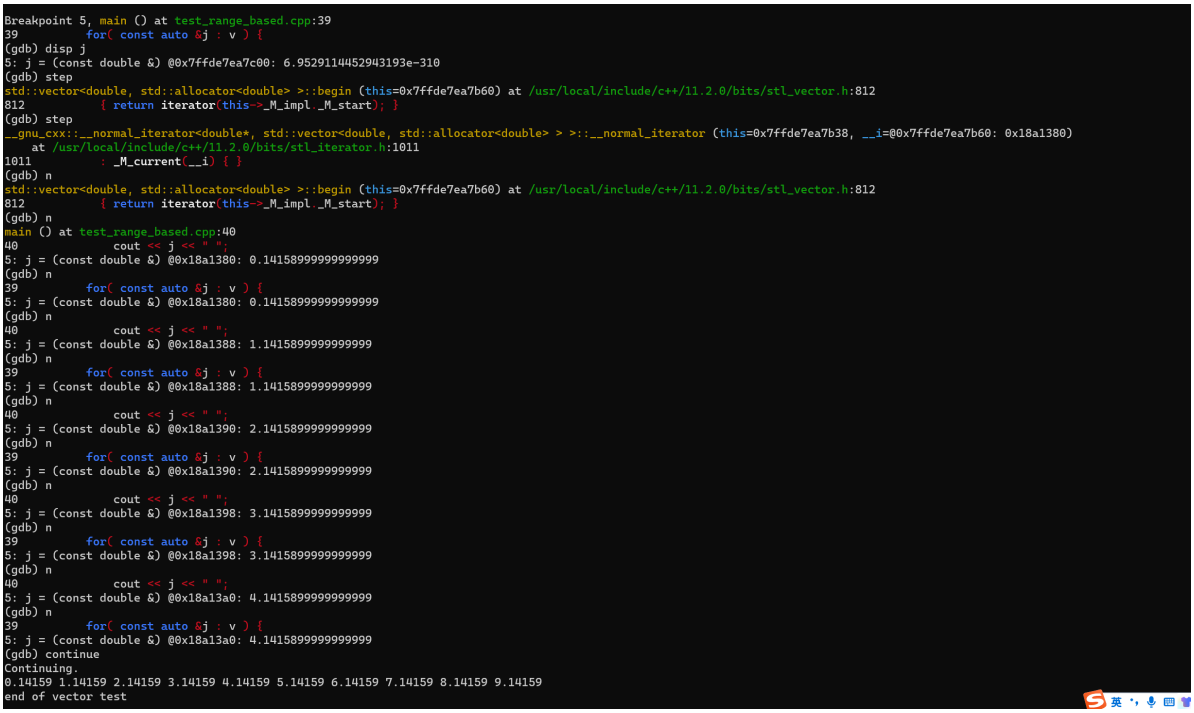
在第四个循环中，y是一个const int的取地址，y是不可修改的

最后，关于j

vector里存着若干实数 i+0.14159,而j实质上是一个地址，当我们输出j时，我们会得到j对应的地址的取值

```
Breakpoint 1, main () at test_range_based.cpp:11
11          for( int y : x ) { // Access by value using a copy declared as a specific type.
(gdb) p y
$1 = 4214944
(gdb) step
13              cout << y << " ";
(gdb) p y
$2 = 1
(gdb) step
11          for( int y : x ) { // Access by value using a copy declared as a specific type.
(gdb) p y
$3 = 1
(gdb) step
13              cout << y << " ";
(gdb) p y
$4 = 2
(gdb) n
11          for( int y : x ) { // Access by value using a copy declared as a specific type.
(gdb) disp y
1: y = 2
(gdb) continue
Continuing.
1 2 3 4 5 6 7 8 9 10

Breakpoint 2, main () at test_range_based.cpp:17
17          for( auto y : x ) { // Copy of 'x', almost always undesirable
(gdb) step
18              cout << y << " ";
(gdb) disp y
2: y = 1
(gdb) step
17          for( auto y : x ) { // Copy of 'x', almost always undesirable
2: y = 1
(gdb) step
18              cout << y << " ";
2: y = 2
(gdb) step
17          for( auto y : x ) { // Copy of 'x', almost always undesirable
2: y = 2
(gdb) step
18              cout << y << " ";
2: y = 3
(gdb) continue
Continuing.
1 2 3 4 5 6 7 8 9 10
```

```
Breakpoint 3, main () at test_range_based.cpp:22
22          for( auto &y : x ) { // Type inference by reference.
(gdb) disp y
3: y = (int &) @0x7f52e0eb5902: 264275272
(gdb) step
23              cout << y << " ";
3: y = (int &) @0x7ffde7ea7b80: 1
(gdb) step
22          for( auto &y : x ) { // Type inference by reference.
3: y = (int &) @0x7ffde7ea7b80: 1
(gdb) step
23              cout << y << " ";
3: y = (int &) @0x7ffde7ea7b84: 2
(gdb) step
22          for( auto &y : x ) { // Type inference by reference.
3: y = (int &) @0x7ffde7ea7b84: 2
(gdb) step
23              cout << y << " ";
3: y = (int &) @0x7ffde7ea7b88: 3
(gdb) continue
Continuing.
1 2 3 4 5 6 7 8 9 10

Breakpoint 4, main () at test_range_based.cpp:27
27          for( const auto &y : x ) { // Type inference by const reference.
(gdb) disp y
4: y = (const int &) <error reading variable>
(gdb) step
28              cout << y << " ";
4: y = (const int &) @0x7ffde7ea7b80: 1
(gdb) step
27          for( const auto &y : x ) { // Type inference by const reference.
4: y = (const int &) @0x7ffde7ea7b80: 1
(gdb) step
28              cout << y << " ";
4: y = (const int &) @0x7ffde7ea7b84: 2
(gdb) step
27          for( const auto &y : x ) { // Type inference by const reference.
4: y = (const int &) @0x7ffde7ea7b84: 2
(gdb) step
28              cout << y << " ";
4: y = (const int &) @0x7ffde7ea7b88: 3
(gdb) continue
Continuing.
1 2 3 4 5 6 7 8 9 10
end of integer array test
```



3.3 最有意思的一题

我们首先编译并运行这个代码

得到的这个输出只能说是匪夷所思

```
root@8fd9ee166d63:/ws/code/Assignment_1_test/question4# g++ -g -Wall test_const.cpp -o 3
root@8fd9ee166d63:/ws/code/Assignment_1_test/question4# ./3
hello
0x402004
3
7
10
6
```

首先是第一个部分，也是最合理的部分

输出p得到hello

根据注释，我们得到宏是全局的这一结论

预处理的宏在编译时已经消失，#define 的产物不会存在地址

然后我们愉快地开始调试

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from 3...
(gdb) l
10
11          static const int NUM = 3;
12          enum con {
13              NUM1 = 3
14          };
15      };
16
17      #define MAX(a,b) ((a) > (b) ? (a) : (b))
18      template<typename T>
19      inline int Max(const T& a, const T& b){
(gdb)
20          return (a>b ? a:b);
21      }
22      const int C::NUM;
23      int main() {
24          cout << p << endl;
25          C c;
26          cout << &c.NUM << endl;
27
28          cout << C::NUM1 << endl;
29
(gdb) b 25
Breakpoint 1 at 0x4011ba: file test_const.cpp, line 26.
(gdb)
Note: breakpoint 1 also set at pc 0x4011ba.
Breakpoint 2 at 0x4011ba: file test_const.cpp, line 26.
(gdb) l
30          int a=5, b=0;
31          cout<<MAX(++a, b)<<endl;
32          cout<<MAX(++a, b+10)<<endl;
33          a=5,b=0;
34          cout<<Max(++a,b)<<endl;
35      }
(gdb) b 30
Breakpoint 3 at 0x4011f2: file test_const.cpp, line 30.
(gdb) run
```

```
Breakpoint 3 at 0x4011f2: file test_const.cpp, line 30.
(gdb) run
Starting program: /ws/code/Assignment_1_test/question4/3
warning: Error disabling address space randomization: Operation not permitted
warning: File "/usr/local/lib64/libstdc++.so.6.0.29-gdb.py" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /usr/local/lib64/libstdc++.so.6.0.29-gdb.py
line to your configuration file "/root/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/root/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
hello

Breakpoint 1, main () at test_const.cpp:26
26          cout << &c.NUM << endl;
(gdb) disp c
1: c = {static NUM = 3}
(gdb) step
0x402004
28          cout << C::NUM1 << endl;
1: c = {static NUM = 3}
(gdb) step
3

Breakpoint 3, main () at test_const.cpp:30
30          int a=5, b=0;
1: c = {static NUM = 3}
(gdb) step
31          cout<<MAX(++a, b)<<endl;
1: c = {static NUM = 3}
(gdb) step
7
32          cout<<MAX(++a, b+10)<<endl;
1: c = {static NUM = 3}
(gdb) step
10
33          a=5,b=0;
1: c = {static NUM = 3}
(gdb) step
34          cout<<Max(++a,b)<<endl;
1: c = {static NUM = 3}
(gdb) step
Max<int> (a=@0x7ffd7d43b668: 6, b=@0x7ffd7d43b664: 0) at test_const.cpp:20
20          return (a>b ? a:b);
(gdb) step
21      }
(gdb) quit
```

c.NUM 是一个 static const int 它存在地址

C::NUM1是一个enum声明，可以得到3的输出

&C::NUM1中，C::NUM1是一个enum，它没有地址

而在题目的后半部分，我们也能知道其大概流程：++a这个命令，作为一个define中整体，在三目运算符中被反复调用，最终得到了奇怪的解

于是我们更新了cpp代码

```cpp
template<typename T>
inline int Max(const T& a, const T& b){
    return (a>b ? a:b);
}
const int C::NUM;
int main() {
    cout << p << endl; //输出hello
    C c;
    cout << &c.NUM << endl;//输出const 的地址
    cout << C::NUM1 << endl;//输出正常值3

    int a=5, b=0;
    cout<<Max(++a, b)<<endl;
    cout<<Max(++a, b+10)<<endl;
    a=5,b=0;
    cout<<Max(++a,b)<<endl;
}
```

再次编译运行

```
root@8fd9ee166d63:/ws/code/Assignment_1_test/question4# g++ test_const.cpp -o 3
root@8fd9ee166d63:/ws/code/Assignment_1_test/question4# ,/3
bash: ,/3: No such file or directory
root@8fd9ee166d63:/ws/code/Assignment_1_test/question4# ./3
hello
0x402004
3
6
10
6
```

得到了预期的结果，其中第二行为const的地址

结论上，const 有地址，enum与#define没有地址

受限于理论水平，在解决这一题的时候，我只能通过注释和调试情况来进行大胆的猜测和不严谨的描述，期待以后助教老师的深入讲解QWQ