2 Makefile & Cmake

2.3 Exercise

首先给出项目文件夹的树形结构：

```
|-- CMakeLists.txt
|-- build
`-- lib
    |-- CMakeLists.txt
    |-- main.cpp
    |-- stuinfo.cpp
    `-- stuinfo.h
```

项目中的 CMakeLists.txt 内容：

```
PROJECT(student)
# 项目名称
add_subdirectory(lib bin)
# 加入子文件夹lib 并且把其生成的二进制信息与.so文件与可执行文件存于 /build/bin
```

lib文件夹中的 CMakeLists.txt 内容：

```
add_library(stuinfo SHARED stuinfo.cpp) #创建库
add_executable(main main.cpp) #生成可执行文件
target_link_libraries(main stuinfo) #把库链接到main
```

stuinfo.h 内容：

```
#include<bits/stdc++.h>
using namespace std;
struct stuinfo{
    char name[20];
    double score[3];
    double ave;
};
void inputstu(stuinfo stu[],int n);
void showstu(stuinfo stu[],int n);
void sortstu(stuinfo stu[],int n);
void findstu(stuinfo stu[],int n,char ch[]);
```

stuinfo.cpp 内容：

```
#include "stuinfo.h"
#include <bits/stdc++.h>
using namespace std;
inline bool cmp(stuinfo x,stuinfo y){ return x.ave>y.ave; }
void inputstu(stuinfo stu[],int n){
```

```
    cout<<"Please enter student information"<<endl;
    for(int i=0;i<n;i++){
        scanf("%s",stu[i].name);
        cin>>stu[i].score[0]>>stu[i].score[1]>>stu[i].score[2];
        stu[i].ave=(stu[i].score[0]+stu[i].score[1]+stu[i].score[2])/3.0;
    }
}
void showstu(stuinfo stu[],int n){
    cout<<"The student information is displayed as follows"<<endl;
    for(int i=0;i<n;i++){
        printf("%s ",stu[i].name);
        cout<<stu[i].score[0]<<' '<<stu[i].score[1]<<' '<<stu[i].score[2]<<'
'<<stu[i].ave<<endl;
    }
}
void sortstu(stuinfo stu[],int n){
    cout<<"The ranking of students by average score is as follows"<<endl;
    sort(stu,stu+n,cmp);
    for(int i=0;i<n;i++) printf("%s %lf\n",stu[i].name,stu[i].ave);
}
void findstu(stuinfo stu[],int n,char ch[]){
    int l=strlen(ch);
    for(int i=0;i<n;i++){
        bool tag=1;
        for(int j=0;j<l;j++){
            if(ch[j]!=stu[i].name[j]){ tag=0; break; }
        }
        if(tag){
            cout<<"Given characters is the student's name."<<endl;
            return;
        }
    }
    cout<<"Given characters is not the student's name."<<endl;
}
```

main.cpp 内容：

```
#include "stuinfo.h"
#include <bits/stdc++.h>
using namespace std;
stuinfo stu[100];
int main(){
    cout<<"Please enter the number of students"<<endl;
    int n; cin>>n;
    inputstu(stu,n);
    showstu(stu,n);
    sortstu(stu,n);
    char ch[20];
    cout<<"Please enter the name of the student you want to find"<<endl;
    scanf("%s",ch);
    findstu(stu,n,ch);
```

```
}
```

生成Makefile文件以及运行结果：

```
root@730aa3418e90:/ws/cmake01/build# cmake ..
CMake Warning (dev) in CMakeLists.txt:
  No cmake_minimum_required command is present.  A line of code such as

    cmake_minimum_required(VERSION 3.18)

  should be added at the top of the file.  The version specified may be lower
  if you wish to support older CMake versions for this project.  For more
  information run "cmake --help-policy CMP0000".
This warning is for project developers.  Use -Wno-dev to suppress it.

-- Configuring done
-- Generating done
-- Build files have been written to: /ws/cmake01/build
root@730aa3418e90:/ws/cmake01/build# make
[ 50%] Built target stuinfo
[100%] Built target main
```

```
root@730aa3418e90:/ws/cmake01/build/bin# ./main
Please enter the number of students
5
Please enter student information
jack 10 10 10
lisa 20 0 50
liya 100 90 5
bob 38 67 100.0
sara 77.9 23.0 0
The student information is displayed as follows
jack 10 10 10 10
lisa 20 0 50 23.3333
liya 100 90 5 65
bob 38 67 100 68.3333
sara 77.9 23 0 33.6333
The ranking of students by average score is as follows
bob 68.333333
liya 65.000000
sara 33.633333
lisa 23.333333
jack 10.000000
Please enter the name of the student you want to find
YangXinyu
Given characters is not the student's name.
```

3 Types

3.1问答题

- **问**：static 用法和作用

  答：用于声明静态变量、静态函数和静态类成员变量。**静态变量**：约等于全局变量（但它只能在声明它的源文件里面被调用），它被初始化之后直到程序结束时才被销毁，如何在函数里声明的，那么在多次调用同一个函数时，静态变量的值会保留在内存中，不会被重新初始化；**静态函数**：和静态变量一样，就是只能在申明它的源文件里面调用；**静态类成员变量**：被该类的所有对象所共享的，不属于任何对象。类的静态成员变量和静态成员函数有一种特殊的使用方式，可以在不同的源文件中共享使用。

- **问**：什么是隐式转换, 如何消除隐式转换?

  答：隐式转换指的是在表达式中，编译器会自动将其中一种数据类型转换成另一种数据类型，当发生赋值语句、函数传参、类的初始化等许多时候都会出现。如何消除：使用显式类型转换来明确指定数据类型，或使用类型安全枚举，函数重载，关键字explicit等方法。

3.2程序解释题

3.2.1 代码1

第一次求和发生了溢出

第二部分 1234567890.0 超过了 $float$ 的精度范围

第三部分浮点数的加法运算中会有误差积累，乘法操作的误差小很多。

3.2.2 代码2

$0.1f$ 在 $float$ 里会表示成 0.1000000014901 （二进制无法表示 0.1 ），做加法误差会很大。

3.2.3 代码3

a 是 41.98 取整等于 41 ； b 是 19 + 21 = 40 ； c 等于两个整数相除，在 c++ 里整数相除等于一个整数；d 等于两个浮点数相除，结果也是浮点数。0/0 会 RE ，运行错误。

4 Structs

4.1结构体对齐问题

`alignas`和`alignof`是C++11中引入的两个关键字，用于控制和查询对象的对齐方式。

`alignof`则用于查询一个类型的对齐方式，它返回一个`size_t`类型的值，表示指定类型的对齐方式。语法为`alignof(T)`，其中`T`是一个类型。

`alignas`用于指定一个对象的对齐方式，它可以被用作类型或者变量的修饰符。语法为`alignas(T)`，其中`T`是一个类型或一个常数表达式，它指定了对齐方式。当 `alignas` 的指示小于其自然的对齐大小（即最大的元素的大小）时，指示会被忽略。也就是两份代码中生效失效的情况。

4.2 Exercise

首先给出项目文件夹的树形结构：

```
|-- CMakeLists.txt
|-- build
`-- lib
    |-- CMakeLists.txt
    |-- geo.cpp
    |-- geo.h
    `-- main.cpp
```

项目中的 CMakeLists.txt 内容：

```
project(geometry)
add_subdirectory(lib bin)
```

lib文件夹中的 CMakeLists.txt 内容：

```
add_library(geo SHARED geo.cpp)
add_executable(main main.cpp)
target_link_libraries(main geo)
target_compile_options(main PUBLIC -Wall -O2) # 加入编译选项
```

geo.h 内容：

```
#include <bits/stdc++.h>
using namespace std;
struct Point {
    double x, y;
    Point operator+(const Point& other) const { return {x+other.x,y+other.y}; }
    Point operator-(const Point& other) const { return {x-other.x,y-other.y}; }
    double operator*(const Point& other) const { return x*other.y-y*other.x; }
    double len(){ return sqrt(x*x+y*y); }
}; // 点
using Vec = Point; // 向量
struct Line { Point P; Vec v; }; // 直线（点向式）
struct Seg { Point A, B; }; // 线段（存两个端点）
struct Circle { Point O; double r; }; // 圆（存圆心和半径）
const Point O = {0, 0}; // 原点
const Line Ox = {O, {1, 0}}, Oy = {O, {0, 1}}; // 坐标轴
const double PI = acos(-1),EPS = 1e-9; //PI
// 三角形的重心
pair<bool, Point> barycenter(const Point &A, const Point &B, const Point &C);
// 三角形的外心
pair<bool, Point> circumcenter(const Point &A, const Point &B, const Point &C);
// 三角形的内心
pair<bool, Point> incenter(const Point &A, const Point &B, const Point &C);
// 三角形的垂心
pair<bool, Point> orthocenter(const Point &A, const Point &B, const Point &C);
```

geo.cpp 内容：

```
#include "geo.h"
#include <bits/stdc++.h>
using namespace std;
inline bool chk(Point A,Point B,Point C){
    Vec a=B-A,b=C-A;
    if(abs(a*b)<=EPS) return 0;
    return 1;
}
double distance(Point A, Point B) {
    return sqrt((A.x-B.x)*(A.x-B.x)+(A.y-B.y)*(A.y-B.y));
}
// 三角形的重心
pair<bool, Point> barycenter(const Point &A, const Point &B, const Point &C){
    if(!chk(A,B,C)) return {0,{0,0}};
    Point BC=C+B,AC=C+A,AB=B+A;
    Point ans=BC+AC+AB;
```

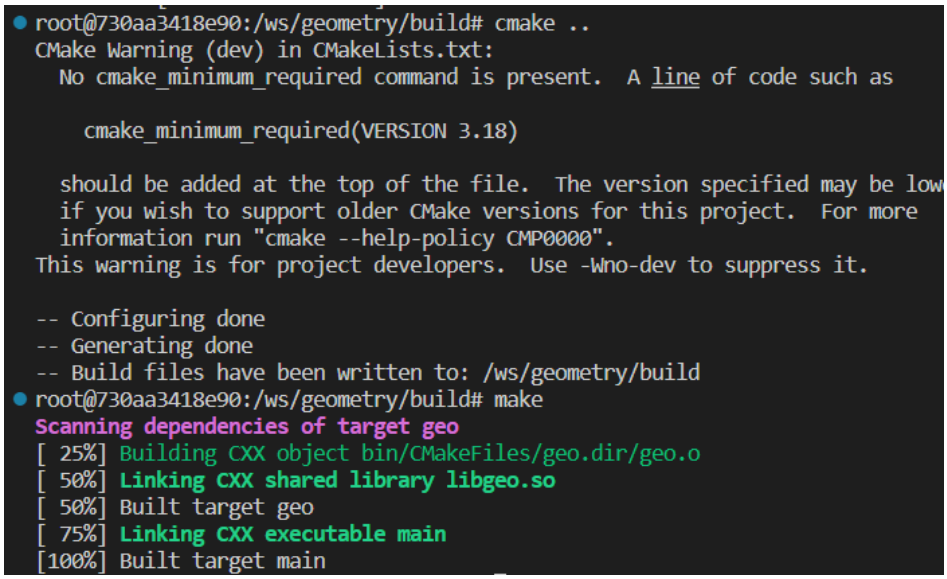
```
    ans.x/=6.0,ans.y/=6.0;
    return {1,ans};
}
// 三角形的外心
pair<bool, Point> circumcenter(const Point &a, const Point &b, const Point &c){
    if(!chk(a,b,c)) return {0,{0,0}};
    double x1=a.x,y1=a.y,x2=b.x,y2=b.y,x3=c.x,y3=c.y;
    double a1=2*(x2-x1),b1=2*(y2-y1),c1=x2*x2+y2*y2-x1*x1-y1*y1;
    double a2=2*(x3-x2),b2=2*(y3-y2),c2=x3*x3+y3*y3-x2*x2-y2*y2;
    double x=(c1*b2-c2*b1)/(a1*b2-a2*b1);
    double y=(a1*c2-a2*c1)/(a1*b2-a2*b1);
    return {1,{x,y}};
}
// 三角形的内心
pair<bool, Point> incenter(const Point &A, const Point &B, const Point &C){
    if(!chk(A,B,C)) return {0,{0,0}};
    double a=distance(B,C),b=distance(A,C),c=distance(A,B);
    double x=(a*A.x+b*B.x+c*C.x)/(a+b+c);
    double y=(a*A.y+b*B.y+c*C.y)/(a+b+c);
    Point ans={x,y};
    return {1,ans};
}
// 三角形的垂心
pair<bool, Point> orthocenter(const Point &A,const Point &B,const Point &C) {
    if(!chk(A,B,C)) return {0,{0,0}};
    double a=distance(B,C),b=distance(A,C),c=distance(A,B);
    double s=(a+b+c)/2;
    double area=sqrt(s*(s-a)*(s-b)*(s-c));
    double x=(a*a*(b*b+c*c-a*a)*A.x+b*b*(c*c+a*a-b*b)*B.x+c*c*(a*a+b*b-
c*c)*C.x)/(16*area*area);
    double y=(a*a*(b*b+c*c-a*a)*A.y+b*b*(c*c+a*a-b*b)*B.y+c*c*(a*a+b*b-
c*c)*C.y)/(16*area*area);
    return {1,{x,y}};
}
```

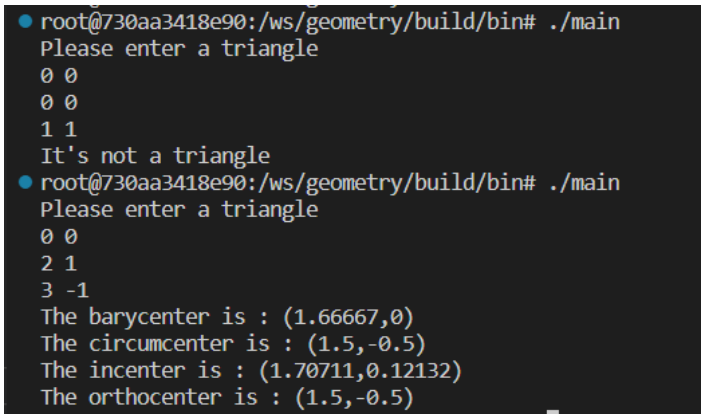
main.cpp 内容：

```
#include "geo.h"
#include <bits/stdc++.h>
using namespace std;
int main(){
    cout<<"Please enter a triangle"<<endl;
    Point A,B,C;
    cin>>A.x>>A.y>>B.x>>B.y>>C.x>>C.y;
    pair<bool,Point> ans=barycenter(A,B,C);
    if(!ans.first){ cout<<"It's not a triangle"<<endl; return 0; }
    cout<<"The barycenter is : ("<<ans.second.x<<","<<ans.second.y<<")"<<endl;
    ans=circumcenter(A,B,C);
    cout<<"The circumcenter is : ("<<ans.second.x<<","<<ans.second.y<<")"<<endl;
    ans=incenter(A,B,C);
    cout<<"The incenter is : ("<<ans.second.x<<","<<ans.second.y<<")"<<endl;
```

```
    ans=orthocenter(A,B,C);
    cout<<"The orthocenter is : ("<<ans.second.x<<","<<ans.second.y<<")"<<endl;
}
```

生成Makefile文件以及运行结果：





5 C++ 动态内存申请

5.1 C++ 的内存分区

（1）全局/静态存储区

（2）全局/静态存储区

（3）栈

（4）储存于栈，指向常量储存区

（5）堆，动态分配空间

（6）堆，动态分配空间

5.2 问答题

（1）new 和 malloc 的区别：我简单列出了个表格，展示 C++ 中 new 和 malloc 的主要区别：

特性	new	malloc
类型	运算符	函数
构造	自动调用构造函数	不调用构造函数
大小	自动计算类型大小	手动指定字节数
异常	抛出std::bad_alloc异常	返回NULL
指针	返回类型化指针	返回void指针，需要强制类型转换
重载	可以重载	不能重载
类重载	可以用于实现自定义的内存分配策略	不能用于实现自定义的内存分配策略

（2）delete p、delete[ ] p、allocator 都有什么作用：delete p 用于释放一个指针 p 所指向的动态分配的内存。当我们使用 new 运算符动态分配内存时，系统会在堆上为对象分配一块内存，并返回指向该内存块的指针。当对象不再需要时，我们需要使用 delete 运算符释放这些动态分配的内存，以便它们可以被操作系统回收。

delete p 只能用于释放通过 new 运算符动态分配的内存，不能用于释放栈上分配的内存。对于数组的动态分配内存，应该使用 **delete[]** 运算符来释放内存，而不是使用 delete 运算符。例如，对于指针 p 指向的动态分配的数组，应该使用 delete[] p 来释放内存。使用 **allocator** 可以分配内存，而不必使用 new 运算符。可以使用 allocator 的 allocate 函数来分配内存，并使用 deallocate 函数来释放内存。

（3）malloc 申请的存储空间能用 delete 释放吗：malloc 函数用于动态分配内存，而不会调用对象的构造函数和析构函数。相对应地，delete 运算符会调用对象的析构函数，并释放对象所占用的内存。因此，不能将 malloc 分配的内存直接用 delete 运算符释放。

如果需要使用 delete 运算符释放内存，应该使用 new 运算符来分配内存。如果必须使用 malloc 分配内存，则应该使用 free 函数来释放内存。给一个用 malloc 和 free 的例子：

```
#include <cstdlib>
#include <iostream>

int main() {
    int* p = (int*)malloc(sizeof(int)); // 使用 malloc 分配内存
    *p = 10; // 对内存进行赋值

    std::cout << *p << std::endl; // 输出内存中的值

    free(p); // 使用 free 函数释放内存
    p = nullptr; // 将指针设置为 nullptr，以避免悬空指针

    return 0;
}
```

PS：使用 malloc 分配内存时，应该使用类型转换将 void* 指针转换为需要的类型指针。同时，为避免悬空指针，应该在释放内存之后将指针设置为 nullptr。

（4）malloc 与 free 的实现原理：malloc 函数用于在程序运行期间动态地申请一块指定大小的内存空间，以供程序使用。它的实现原理是在系统的堆区（heap）中分配一块连续的内存空间，并返回该内存空间的起始地址，以供程序使用。

在内部实现时，malloc 函数会记录已经分配的内存块和空闲的内存块，以便在下一次 malloc 调用时可以快速分配已经释放的空间。同时，为了避免内存碎片，malloc 函数会按照一定的规则分配内存，以尽可能地减少未分配的内存块。

free 的原理： free 函数用于释放由 malloc 函数分配的内存空间。在内部实现时，free 函数会将需要释放的内存空间标记为已经释放，以便下一次 malloc 函数调用时可以重用该空间。

5.3 Exercise

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.11)
project(random_generator_project)
add_library(${PROJECT_NAME} SHARED src/RandomGenerator.cpp)
target_include_directories(${PROJECT_NAME} PUBLIC
${CMAKE_CURRENT_SOURCE_DIR}/include)
add_executable(main src/main.cpp)
target_link_libraries(main ${PROJECT_NAME})
```

```
//RandomGenerator.hpp
namespace random_generator{
int* Generate(int );
void print(int* ,int );
void Del(int* );
}
```

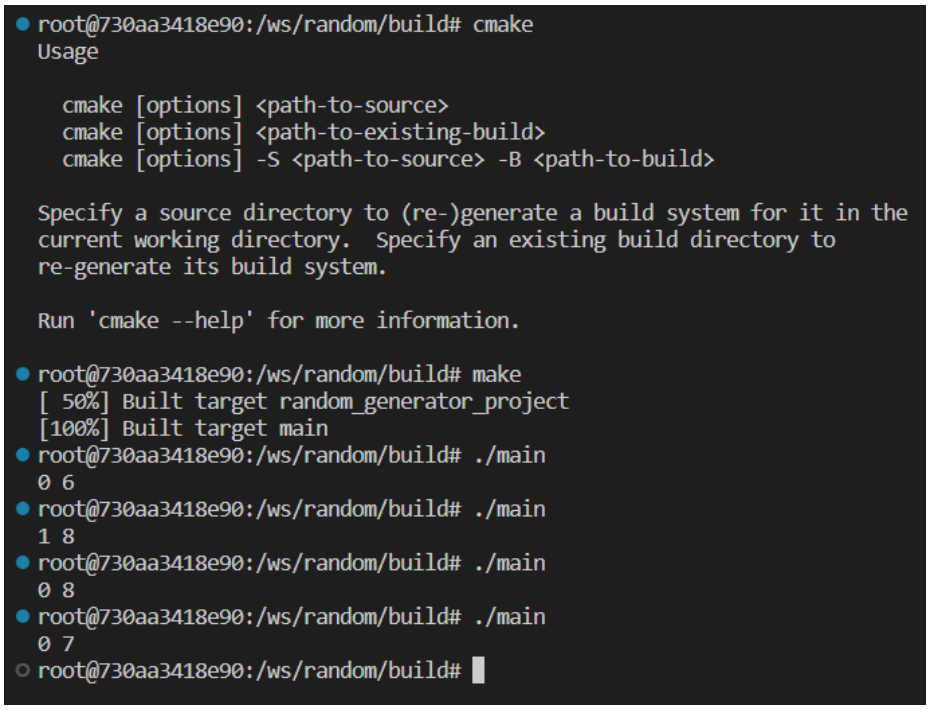
```
//RandomGenerator.cpp
#include <RandomGenerator.hpp>
#include <bits/stdc++.h>
using namespace std;
namespace random_generator{

mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
int* Generate(int n){
    int* A = new int[n];
    for(int i=0; i<n; i++) A[i] = rng() % n;
    return A;
}
void print(int* A, int n){
    if(A == nullptr) return;
    int maxn = A[0], minn = A[0];
    for(int i=1; i<n; i++)
        maxn = max(maxn, A[i]), minn = min(minn, A[i]);
    cout<< minn << ' ' << maxn << endl;
```

```
}
void Del(int* A){
    delete []A;
}

}
```

生成Makefile文件以及运行结果：



6 Debug & Release

6.2和6.3的截屏我放在了 *6.2&6.3的运行结果截屏合集.png* 里

6.2 如何判断动态申请越界（C 方式，注意源程序后缀为.c）

由于 molloc 是用链表来维护的堆，所以申请的空间的前后位置不能改，这也是在 VSdebug 唯一一回报错的原因。通过实践发现，这段空间 VSdebug 下是 0xfffffffd ， linux 下是 0 ， 其实用 malloc 开空间越界的话不管哪个平台都不能完全给你报错，在linux下我提供一个方法判断动态申请越界：

可以使用以下命令进行编译：

```
g++ -fsanitize=address -g your_program.cpp -o your_program
```

直接运行编译后的程序即可。如果存在内存越界的问题会输出错误信息。

6.3 如何判断动态申请越界（C 方式，注意源程序后缀为.c）

通过观察发现和 6.2 没什么区别，只是 c++ 没有报错了（bushi）。根据之前对动态申请空间的了解，使用 `new` 创建的对象，访问越界会导致程序直接崩溃，即程序会抛出一个异常。而使用 `malloc` 创建的内存块，访问越界则不会导致程序崩溃，程序可能会继续执行下去，但是结果是不可预测的，这可能会导致程序出现各种难以调试的问题。这是因为在 C++ 中， `new` 运算符会使用 `operator new` 函数来分配内存，并

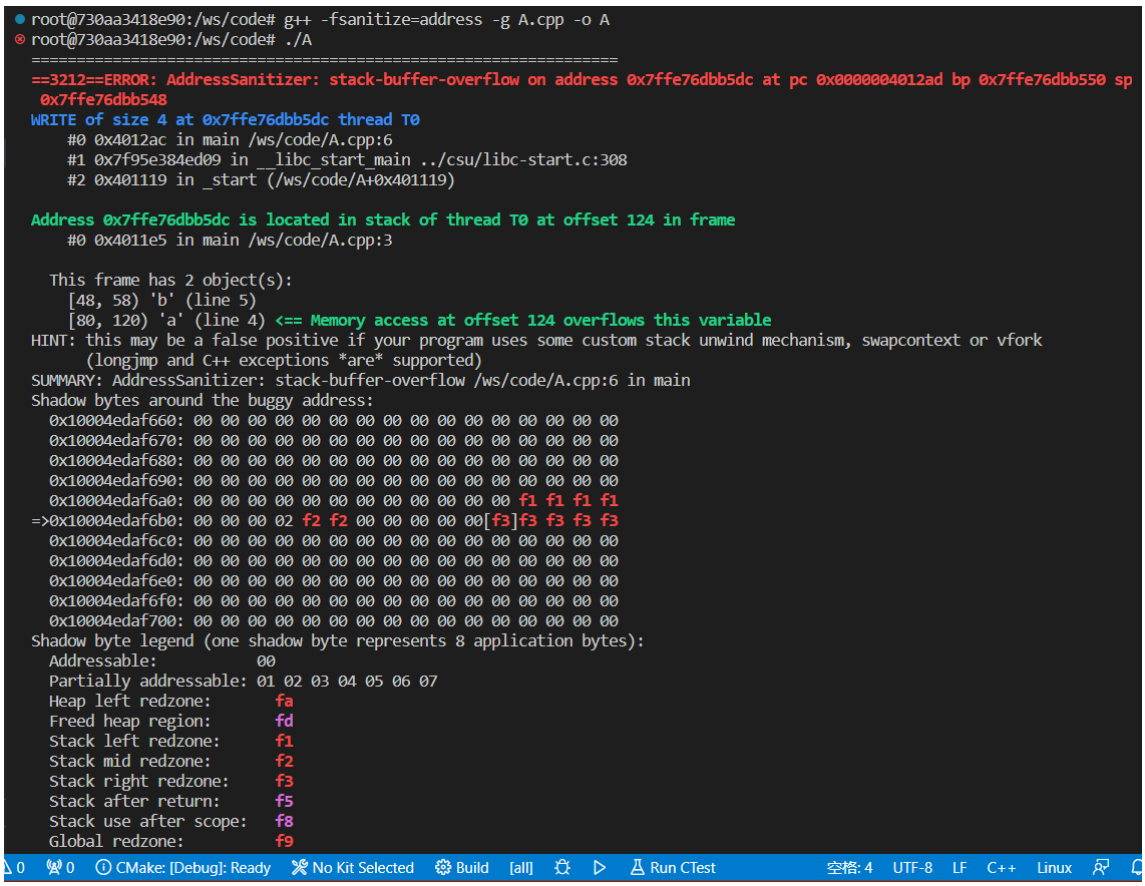
在分配内存时调用`std::bad_alloc`异常，当内存分配失败时，程序会抛出这个异常。而使用`malloc`则没有这个机制，即使分配内存失败也不会抛出异常，而是返回`NULL`指针。在访问越界时，使用`new`创建的对象，越界访问会触发操作系统的内存保护机制，从而导致程序崩溃。而使用`malloc`分配的内存，越界访问不会触发内存保护机制，因此程序不会崩溃，但是可能会出现不可预测的结果。

6.4 如何判断普通数组的越界访问（C++ 方式，注意源程序后缀为.cpp）

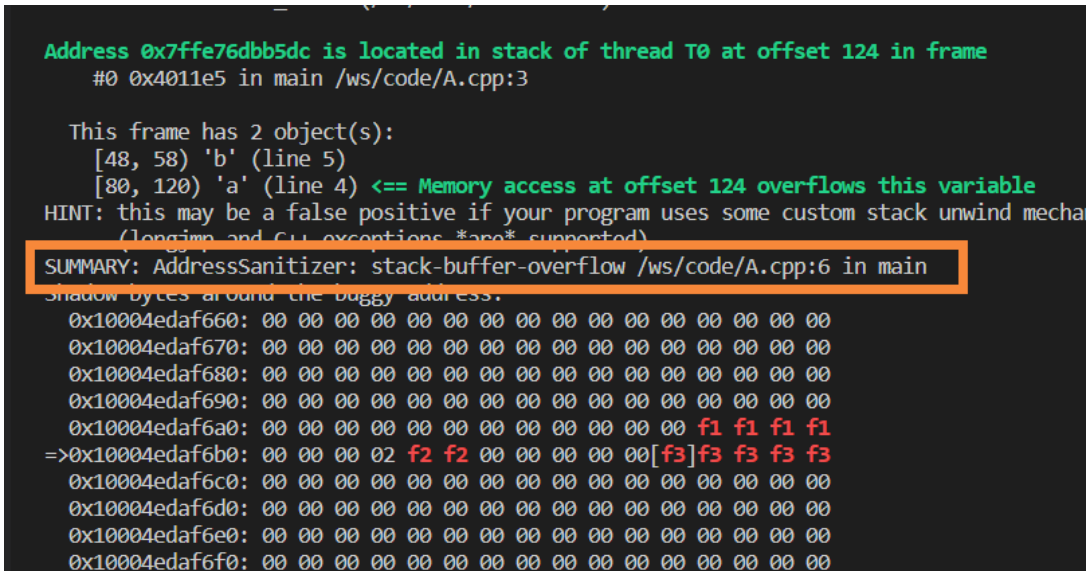
使用 6.2 中我提到的方法判断静态数组越界：

```cpp
//A.cpp
#include <bits/stdc++.h>
using namespace std;
int main(){
    int a[10];
    char b[10];
    a[11]=1,b[12]='K';
}
```

使用`g++ -fsanitize=address -g A.cpp -o A`编译，并运行：



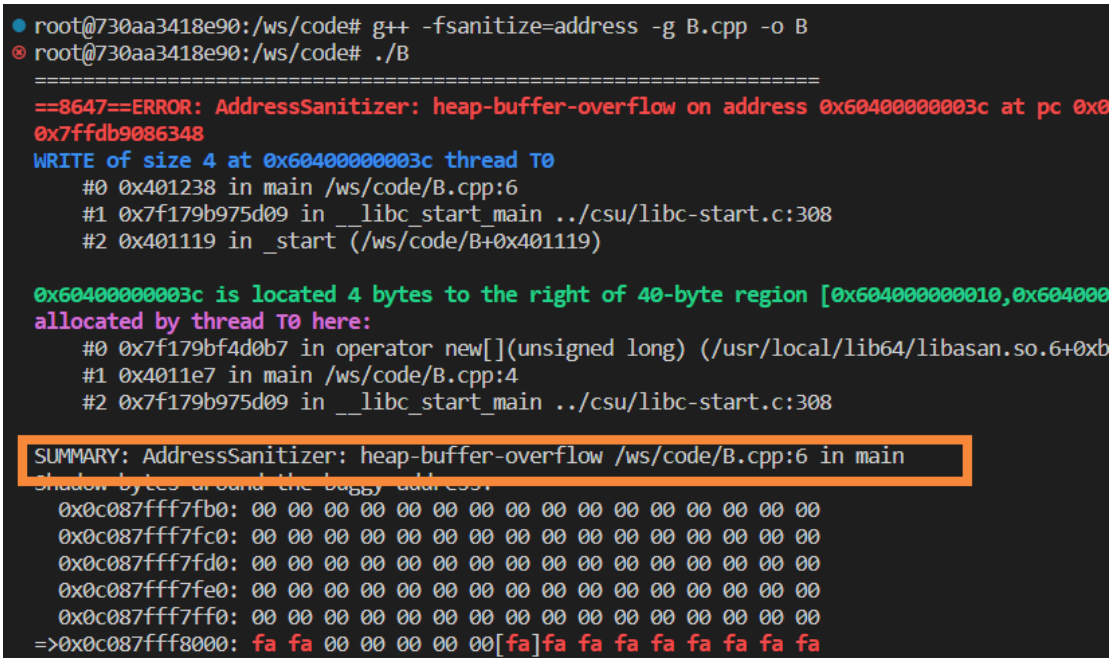
省流：cpp 文件的第 6 行数组越界



类似地，判断动态申请的数组的越界：

```cpp
//B.cpp
#include <bits/stdc++.h>
using namespace std;
int main(){
    int* a=new int[10];
    char* b=new char[10];
    a[11]=1,b[12]='K';
}
```

使用 `g++ -fsanitize=address -g B.cpp -o B` 编译，并运行，可以发现结果和静态数组一样。



在其他环境下结果相同，不赘述。

6.5 总结

又两个小点值得注意，发现很多情况下越界时都不会给出 CE 或 RE 的信息，越界发生的很隐蔽，此时应该考虑用 6.4 的方法编译运行；不同的平台的动态空间的初值是不一样的，释放空间不要无脑赋值，这里我认为 new 和 delete 比 malloc 和 free 更先进更安全。

作为程序员我们想要debug访问越界的锅，我们想知道的并不是内存里的哪个位置被错误访问了，而是我们的代码的哪一行访问到了越界的地方。所以动态&静态内存的分配原理并不重要，写的程序不越界，越界后会 debug 才是程序员要重视的能力。