Assignment 2

张中辉　邓钰川

2023 年 3 月 7 日

目录

1 作业内容概述

- 学习利用 CMake 编译多文件系统
- 完成有关 Types and Structs 相关题目。
- 了解 C++ 的内存管理机制与 C++ 的动态内存申请
- 了解 Debug 模式和 Release 模式的区别

具体来说，您需要完成以下任务，并提供相应的材料形成报告 (**不是单纯的截屏集合**)，并在 3 月 21 日之前发送邮箱至 1376447388@qq.com(张中辉) 或者 2021141460159@stu.scu.edu.cn(邓钰川)。

2 Makefile & Cmake

2.1 Makefile 的简略介绍

Makefile 的介绍已在群里给出，更多信息请参考官方文档[1]

2.2 Cmake 的使用

CMake 是一个开源、跨平台的工具系列，旨在构建、测试和打包软件。CMake 用于使用简单的平台和独立于编译器的配置文件控制软件编译过程，并生成可在您选择的编译器环境中使用的本机 makefile 和工作区[2]

CMake 需要 CMakeLists.txt 才能正常运行。CMakeLists.txt 由命令、注释和空格组成。命令包括命令名、括号和参数，参数用空格分隔。命令不区分大小写。CMake 官方文档如下[3]

以下是一些重要的编译参数

```
# CMake 最小版本要求为 2.8.3
cmake_minimum_required(VERSION 2.8.3)

# 指定工程名为 HELLOWORLD
project(HELLOWORLD)

# 定义 SRC 变量，其值为 sayhello.cpp hello.cpp
set(SRC sayhello.cpp hello.cpp)

# 将/usr/include/myincludefolder 和 ./include 添加到头文件搜索路径
include_directories(/usr/include/myincludefolder ./include)

# 将/usr/lib/mylibfolder 和 ./lib 添加到库文件搜索路径
link_directories(/usr/lib/mylibfolder ./lib)
```

[1] https://www.gnu.org/software/make/manual/make.html
[2] https://cmake.org/documentation/
[3] https://cmake.org/documentation/

```
15
16 # 通过变量 SRC 生成 libhello.so 共享库
17 add_library(hello SHARED ${SRC})
18
19 # 添加编译参数 -Wall -std=c++11 -O2
20 add_compile_options(-Wall -std=c++11 -O2)
21
22 # 编译 main.cpp 生成可执行文件 main
23 add_executable(main main.cpp)
24
25 # 将 hello 动态库文件链接到可执行文件 main
26 target_link_libraries(main hello)
27
28 # 添加 src 子目录，src 中需有一个 CMakeLists.txt
29 add_subdirectory(src)
30
31 # 定义 SRC 变量，其值为当前目录下所有的源代码文件
32 aux_source_directory(. SRC)
33
34 # 编译 SRC 变量所代表的源代码文件，生成 main 可执行文件
35 add_executable(main ${SRC})
36
37 # 在 CMAKE_CXX_FLAGS 编译选项后追加-std=c++11
38 set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
39
40 # 设定编译类型为 debug，调试时需要选择 debug
41 set(CMAKE_BUILD_TYPE Debug)
42 # 设定编译类型为 release，发布时需要选择 release
43 set(CMAKE_BUILD_TYPE Release)
```

对于已经存在 CMakelist 的项目文件，可以采取下列操作进行编译

```
1 mkdir build
2 cd build
3 cmake ..
4 make
```

2.3 Exercise

在 stuinfo.h 中声明一个名为 stuinfo 的结构和下面的四个函数原型。在 stufun.cpp 中实现这四个函数。编写一个包含 main () 的 main.cpp，并演示原型函数的所有特性。

为 cmake 编写一个 CMakeLists.txt，以自动创建 Makefile。运行 cmake 和 make，然后最后运行程序。

```
struct stuinfo
{
    char name[20];
    double score[3];
    double ave;
};
```

Function prototypes:

```
void inputstu(stuinfo stu[] , int n);
//  asks the user to enter each of the preceding items of
//information to set the corresponding members of the structure.
void showstu(stuinfo stu[] , int n); //displays the contents of the structure, one student one line.
void sortstu(stuinfo stu[] , int n); //sorts in descending order of  average of three scores.
bool findstu(stuinfo stu[] , int n, char ch[]); //finds if  given characters is the student' s name.
```

1. 编写 CMakeLists 时，不建议大家提交 IDE 自动生成的文件（比如利用 Vscode 插件或者 CLion 直接建立），最好自己上手编写尝试。
2. 代码运行结果需要截屏附在报告里。

3 Types

3.1 问答题

对于问答题，请回答以下问题

1. static 用法和作用
2. 什么是隐式转换, 如何消除隐式转换?

3.2 程序解释题

运行下列程序，请你解释下列程序的输出

```
#include <iostream>

using std::cout;
using std::endl;

int main() {
```

```
    int num1 = 1234567890;
    int num2 = 1234567890;
    int sum = num1 + num2;
    cout << "sum = " << sum << endl;

    float f1 = 1234567890.0f;
    float f2 = 1.0f;
    float fsum = f1 + f2;
    cout << "fsum = " << fsum << endl;
    cout << "(fsum == f1) is " << (fsum == f1) << endl;

    float f = 0.1f;
    float sum10x = f + f + f + f + f + f + f + f + f + f;
    float mul10x = f * 10;

    cout<<"sum10x = "<< sum10x << endl;
    cout<<"mul10x = "<< mul10x << endl;
    cout<<"(sum10x == 1) is "<< (sum10x == 1.0) << endl;
    cout<<"(mul10x == 1) is "<< (mul10x == 1.0) << endl;
    return 0;
}
```

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

int main()
{
    cout << fixed;
    float f1 = 1.0f;
    cout<<"f1 = "<<f1<<endl;

    float a = 0.1f;
    float f2 = a+a+a+a+a+a+a+a+a+a;
    cout<<"f2 = "<<f2<<endl;

    if(f1 == f2)
        cout << "f1 = f2" << endl;
    else
        cout << "f1 != f2" << endl;

```

```
    return 0;
}
```

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

int main()
{
    int a, b;
    double c, d;

    a = 19.99 + 21.99;
    b = (int)19.99 + (int)21.99;
    c = 23 / 8;
    d = 23 / 8.0;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    cout << "d = " << d << endl;
    cout << "0/0= " << 0/0 << endl;

    return 0;
}
```

3.3 编程题

请实现一个好的计算器。编程语言必须为 C++，下面是对于这个计算器的相关要求。

3.3.1 Requirements

1. When you run your program and input an express in a line as follows, it can output the correct results. The operator precedence (order of operations) should be correct.

```
    2+3
    5
    5+2*3
    11
```

2. Use parentheses to enforce the priorities.

```
(5+2)*3
21
```

3. Some math functions can be supported.

```
sqrt(3.0)
1.7
```

4. Variables can be defined as follows. (Optional)

```
x=3
y=6
x+2*y
15
```

5. It can support arbitrary precision. (Optional)

```
99999999999999999999999999.222222222 + 1.0
100000000000000000000000000.222222222
```

More features can be found in the calculator BC in Unix-like systems.[4]

4 Structs

4.1 结构体对齐问题

- 结构体内成员按照声明顺序存储, 第一个成员地址和整个结构体地址相同。
- 未特殊说明时, 按结构体中 size 最大的成员对齐 (若有 double 成员, 按 8 字节对齐。)

c++11 以后引入两个关键字 alignas 与 alignof。其中 alignof 可以计算出类型的对齐方式, alignas 可以指定结构体的对齐方式。但是 alignas 在某些情况下是不能使用的, 具体见下面的例子:

```
// alignas 生效的情况
struct Info {
    uint8_t a;
    uint16_t b;
    uint8_t c;
};
```

[4] https://www.gnu.org/software/bc/manual/html_mono/bc.html

```
std::cout << sizeof(Info) << std::endl;// 6 2 + 2 + 2
std::cout << alignof(Info) << std::endl;// 2

struct alignas(4) Info2 {
    uint8_t a;
    uint16_t b;
    uint8_t c;
};
std::cout << sizeof(Info2) << std::endl;// 8 4 + 4
std::cout << alignof(Info2) << std::endl;// 4
```

```
// alignas 失效的情况
struct Info {
    uint8_t a;
    uint32_t b;
    uint8_t c;
};
std::cout << sizeof(Info) << std::endl;// 12 4 + 4 + 4
std::cout << alignof(Info) << std::endl; // 4
struct alignas(2) Info2 {
    uint8_t a;
    uint32_t b;
    uint8_t c;
};
std::cout << sizeof(Info2) << std::endl;// 12 4 + 4 + 4
std::cout << alignof(Info2) << std::endl;// 4
```

请你根据以上程序片段尝试给出 alignas 生效和失效的情形解释。

4.2 Exercise

下面是一个简易的二维计算几何的相关结构声明：

```
struct Point { double x, 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 与偏差值
```

您将会使用二维平面上的任意三个点，如果这三个点可以构成三角形，请您分别计算三角形的重心，外心，内心和垂心。如果不能构成三角形，请您给出警告。您需要将这些函数编译到共享库“lib.so”中，使用该共享库并编写一个简单的程序来调用这四个个函数并显示结果。函数的定义可以参考下方。(建议使用 std::pair)

```
// 三角形的重心
std::pair<bool, Point> barycenter(Point A, Point B, Point C);

// 三角形的外心
std::pair<bool, Point> circumcenter(Point A, Point B, Point C);

// 三角形的内心
std::pair<bool, Point> incenter(Point A, Point B, Point C);

// 三角形的垂心
std::pair<bool, Point> orthocenter(Point A, Point B, Point C);
```

PS: 这里定义采用常量引用应该是更好的，但是因为大家还没有学习常量引用，这里只采取值引用的方法。如果采取常量引用，这里可以写成

```
// 三角形的重心
std::pair<bool, Point> barycenter(const Point &A, const Point &B, const Point &C);

// 三角形的外心
std::pair<bool, Point> circumcenter(const Point &A, const Point &B, const Point &C);

// 三角形的内心
std::pair<bool, Point> incenter(const Point &A, const Point &B, const Point &C);

// 三角形的垂心
std::pair<bool, Point> orthocenter(const Point &A, const Point &B, const Point &C);
```

5 C++ 动态内存申请

5.1 C++ 的内存分区

C++ 中的内存分区, 分别是堆、栈、自由存储区、全局/静态存储区、常量存储区和代码区。

- 栈: 在执行函数时, 函数内局部变量的存储单元都可以在栈上创建, 函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中, 效率很高, 但是分配的内存容量有限

- 堆: 就是那些由 new 分配的内存块, 他们的释放编译器不去管, 由我们的应用程序去控制, 一般一个 new 就要对应一个 delete。如果程序员没有释放掉, 那么在程序结束后, 操作系统会自动回收
- 自由存储区: 如果说堆是操作系统维护的一块内存, 那么自由存储区就是 C++ 中通过 new 和 delete 动态分配和释放对象的抽象概念。需要注意的是, 自由存储区和堆比较像, 但不等价。
- 全局/静态存储区: 全局变量和静态变量被分配到同一块内存中, 在以前的 C 语言中, 全局变量和静态变量又分为初始化的和未初始化的, 在 C++ 里面没有这个区分了, 它们共同占用同一块内存区, 在该区定义的变量若没有初始化, 则会被自动初始化, 例如 int 型变量自动初始为 0
- 常量存储区: 这是一块比较特殊的存储区, 这里面存放的是常量, 不允许修改
- 代码区: 存放函数体的二进制代码

下面用一段 C++ 代码举例展示数据在内存中存放的位置

```
class A {
  int num;
}

static int a;    //(1)
auto  b=0;        //(2)

int  main()
{
  char  s[]="abc";    //(3)
  char  *p="123456";  //(4)。

  p1=  (char  *)malloc(10);  // (5)
  A *a = new A();            // (6)
}
```

请你分别指出 (1)(2)(3)(4)(5)(6) 中的数据分别在内存的什么位置。

5.2 问答题

请你查阅相关资料，回答以下问题

1. new 和 malloc 的区别
2. delete p、delete[ ] p、allocator 都有什么作用?
3. malloc 申请的存储空间能用 delete 释放吗?
4. malloc 与 free 的实现原理?

5.3 Exercise

请实现一个项目编程实现如下功能：

- 用 C++ 动态内存分配 n 个整数空间
- 用随机数初始化这 n 个空间，随机数范围为 [0,n -1]
- 输出这 n 个空间的最大和最小数值
- 释放这 n 个空间

您的函数声明应当放在头文件 RandomGenerator.h 中，实现放在 RandomGenerator.cpp 中，并在 main.cpp 中编写一个测试程序输出相关结果。请在报告中附上您的程序运行截图以及构建项目的 Cmakelists/makefile 文件或者 g++ 指令。

6 Debug 和 Release

6.1 Debug 和 Release 的区别

1. 调试版本, 包含调试信息, 所以容量比 Release 大很多, 并且不进行任何优化 (优化会使调试复杂化, 因为源代码和生成的指令间关系会更复杂), 便于程序员调试。在 Windows 的 Debug 模式下生成两个文件, 除了.exe 或.dll 文件外, 还有一个.pdb 文件, 该文件记录了代码中断点等调试信息;
2. 发布版本, 不对源代码进行调试, 编译时对应用程序的速度进行优化, 使得程序在代码大小和运行速度上都是最优的。(调试信息可在单独的 PDB 文件中生成)。Release 模式下生成一个文件.exe 或.dll 文件。
3. 实际上,Debug 和 Release 并没有本质的界限, 他们只是一组编译选项的集合, 编译器只是按照预定的选项行动。事实上, 我们甚至可以修改这些选项, 从而得到优化过的调试版本或是带跟踪语句的发布版本。

下面我们通过 Linux，VS2022 的 Debug/Release 模式对动态内存申请后越界访问进行深度讨论。

6.2 如何判断动态申请越界（C 方式，注意源程序后缀为.c）

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char *p;
    p = (char *)malloc(10 * sizeof(char));
    if (p == NULL)
        return -1;
    strcpy(p, "123456789");
```

```
    p[10] = 'a';    //此句越界 (1)
    p[14] = 'A';    //此句越界
    p[15] = 'B';    //此句越界
    p[10] = '\xfd'; //此句越界 (2)
    printf("addr:%p\n", p);
    for (int i = -4; i < 16; i++) //注意，只有 0-9 是合理范围，其余都是越界读
        printf("%p:%02x\n", (p+i), p[i]);
    free(p);//(3)

    return 0;
}
```

在 VS2022 的 x86/Debug，Release 模式与 Linux 用 GDB 进行 Debug 下运行：

1. (1)(2)(3) 全部注释，观察运行结果
2. (1) 放开，(2)(3) 注释，观察运行结果
3. (1)(3) 放开，(2) 注释，观察运行结果
4. (1)(2)(3) 全部放开，观察运行结果

请提供截图并用文字说明 VS 的 Debug 模式与 Linux 下是如何判断动态申请内存访问越界的？再观察 VS2022 x86/Release 下的运行结果。

6.3 如何判断动态申请越界（C++ 方式，注意源程序后缀为.cpp）

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char *p;
    p = new(nothrow) char[10];
    if (p == NULL)
        return -1;
    strcpy(p, "123456789");
    p[10] = 'a';    //此句越界 (1)
    p[14] = 'A';    //此句越界
    p[15] = 'B';    //此句越界
    p[10] = '\xfd'; //此句越界 (2)
```

```
17      cout << "addr:" << hex << (void *)(p) << endl;
18      for (int i = -4; i < 16; i++) //注意，只有 0-9 是合理范围，其余都是越界读
19          cout << hex << (void *)(p + i) << ":" << int(p[i]) << endl;
20      delete[]p;//(3)
21
22      return 0;
23  }
```

在 VS2022 的 x86/Debug 模式与 Linux 用 GDB 进行 Debug 下运行：

1. (1)(2)(3) 全部注释，观察运行结果
2. (1) 放开，(2)(3) 注释，观察运行结果
3. (1)(3) 放开，(2) 注释，观察运行结果
4. (1)(2)(3) 全部放开，观察运行结果

请提供截图并用文字说明 VS 的 Debug 模式与 Linux 下是如何判断动态申请内存访问越界的？再观察 VS2022 x86/Release 下的运行结果

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）

在理解前两种情况的情况下，自行构造相似的程序，来观察数组越界后的内存表现，并验证与动态申请是否相似要求：

1. 字符数组用 char a[10]; 形式
2. 整形数组用 int a[10]; 形式
3. 测试程序在 VS2022 x86/Debug，VS2022 x86/Release，Linux 三种环境下运行
4. 每种讨论的结果可截图 + 文字说明，如果几种环境的结果一致，用一个环境的截图 + 文字说明即可

6.5 总结

仔细总结本小节作业（多种形式的测试程序/多个编译器环境/不同结论），谈谈你对内存越界访问的整体理解，包括但不限于操作系统/编译器如何防范越界、你应该养成怎样的使用习惯来尽量防范越界。