Assignment 1

张中辉　邓钰川

2023 年 2 月 28 日

目录

1 作业内容概述

- 根据实验环境配置指南与提供的 Dockerfile 配置对应的 docker 环境
- 学习 linux 下简单的 g++ 命令行编译，GDB 调试。
- 根据提供的测试用例完成编译、运行，得到运行结果, 并回答有关问题
- 自行构造测试用例，得出运行结果，对运行情况和结果进行分析讨论
- 我们建议使用 Vscode/Clion/Vim 作为您的代码编辑器, 并链接到您的 docker(Optional)

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，并发送邮箱至 1376447388@qq.com(张中辉) 或者 2021141460159@stu.scu.edu.cn(邓钰川)。

2 配置对应的 docker 环境

根据实验环境配置指南与 Dockerfile 配置对应的 docker 环境，我们需要以下材料以证明您完成了对应的工作:

1. 终端执行 docker run hello-world,docker images,docker ps 的运行截图
2. 使用 docker exec -it <container id> /bin/bash 进入 docker 后 cd /ws/code/的截图

3 利用 g++ 编译单文件

3.1 g++ 编译过程

以我们提供的 test.cpp 文件举例

```
# 编译 test.cpp 文件，在 Linux 下，默认产生名为 a.out 的二进制可执行文件
g++ test.cpp
```

实际上，上面的一步编译指令包含了以下几个过程

第一步：预处理 Pre-processing，生成.i 文件

```
# -E 选项指示编译器仅对输入文件进行预编译
g++ -E test.cpp -o test.i
```

第二步：编译-Compiling，生成.s 文件

```
# -S 编译选项告诉 g++ 在为 c++ 代码产生了汇编语言文件后停止编译
# g++ 产生的汇编语言文件的缺省扩展名是 .s
g++ -S test.i -o test.s
```

第三步：汇编-Assembing，生成.o 文件

```
# -c 选项告诉 g++ 仅把源代码编译为机器语言的目标代码
# 缺省时 g++ 建立的目标代码文件有一个 .o 的扩展名
g++ -c test.s -o test.o
```

第四步：链接-Lingking，生成 bin 二进制文件

```
# -o 编译选项来为将产生的可执行文件指定文件名，如果不使用-o 参数，在 Linux 下默认输出名为 a.out 的可执行文件
g++ test.o -o test
```

3.2 g++ 重要编译参数

设定编译标准

```
# 使用 c++17 标准编译 test.cpp
g++ -std=c++17 test.cpp
```

打印警告信息

```
# 打印出 g++ 提供的警告信息
g++ -Wall test.cpp
```

编译带调试信息的可执行文件

```
# 产生带调试信息的可执行文件 test
g++ -g test.cpp -o test
```

我们可以输入指令：readelf -S 可执行程序，来查看该程序是否生成调试信息

```
readelf -S test | grep -i debug
```

优化源代码

```
## 所谓优化，例如省略掉代码中从未使用过的变量、直接将常量表达式用结果值代替等等，这些操作会缩减目标文件所包含的
# -O 选项告诉 g++ 对源代码进行基本优化。这些优化在大多数情况下都会使程序执行的更快。 -O2 选项告诉 g++ 产生尽可
# -O 同时减小代码的长度和执行时间，其效果等价于-O1
# -O0 表示不做优化
# -O1 为默认优化
```

```
# -O2 除了完成-O1 的优化之外，还进行一些额外的调整工作，如指令调整等。
# -O3 则包括循环展开和其他一些与处理特性相关的优化工作。
# 选项将使编译的速度比使用 -O 时慢，但通常产生的代码执行速度会更快。
# 使用 -O2 优化源代码，并输出可执行文件
g++ -O2 test.cpp
```

根据我们提供的 inefficency.cpp 文件，我们通过更改编译参数测试编译优化的效果先使用直接编译的方式生成对应可执行文件，接下来我们再使用优化后的编译方式，生成对应可执行文件，如下命令

```
# 代码无优化
g++ inefficency.cpp -o without_o.out
# 使用-O2 级别的代码优化
g++ inefficency.cpp -O2 -o with_o.out
```

后使用 time 命令打印对应的执行时间信息

```
time ./with_o.out
time ./without_o.out
```

根据以上举例，我们需要以下材料以证明您完成了对应的工作:

1. 单步与分步编译 test.cpp 文件，并利用 ls 指令打印出相应的过程文件并截图，执行单步编译得到的 a.out 可执行文件, 与分步编译得到的 test 可执行文件，给出相应的运行结果
2. 给出利用 time 命令打印 inefficency.cpp 优化编译与非优化编译的执行信息
3. 选择 3 到 4 个其他的 test 文件，尝试利用不同的 g++ 参数进行编译，给出对应生成文件与最终的执行截图

4 利用 GDB 调试文件

我们在生成 debug 的可执行程序后，直接输入指令 gdb 可执行程序名就可以进入调试状态，对该程序进行调试，输入 q（或者 quit、ctrl d）进行退出 gdb。

4.1 代码的查看

我们在这种无图形化界面的 Linux 下调试时，如何看到我们的代码呢？很简单，只需要输入 l（或者 list），就可以查看我们的代码

输入 l（list）：显示我们的代码（ 默认从中间显示 ），我们在 l 后面加个数字，便可以从指定位置显示。

当然，我们后续不用再输入指令，直接按回车键，依然会继续衔接上面的，往后打印 10 行。

4.2 断点操作

首先，假如我们没有设置断点，我们输入 r（run），此时程序则会从开始，一直运行到结束。假如我们设置了断点，程序则会运行到断点处进行停止。

r：运行程序，无断点的话，直接运行到结束，有断点运行到最近的断点处停止（r 不能在断点间移动，即运行到最近断点处后，再次输入 r，则会重新再次运行

b（break）行号 ：在某一行设置断点

b（break）函数名 ：在该函数处设置断点

输入 info b 指令可以查看所有的断点详细信息

4.3 常见的 gdb 命令

```
$(gdb)help(h) # 查看命令帮助，具体命令查询在 gdb 中输入 help + 命令
$(gdb)run(r) # 重新开始运行文件（run-text：加载文本文件，run-bin：加载二进制文
件）
$(gdb)start # 单步执行，运行程序，停在第一行执行语句
$(gdb)list(l) # 查看原代码（list-n，从第 n 行开始查看代码。list+ 函数名：查看具体函
数）
$(gdb)set # 设置变量的值
$(gdb)next(n) # 单步调试（逐过程，函数直接执行）
$(gdb)step(s) # 单步调试（逐语句：跳入自定义函数内部执行）
$(gdb)backtrace(bt) # 查看函数的调用的栈帧和层级关系
$(gdb)frame(f) # 切换函数的栈帧
$(gdb)info(i) # 查看函数内部局部变量的数值
$(gdb)finish # 结束当前函数，返回到函数调用点
$(gdb)continue(c) # 继续运行
$(gdb)print(p) # 打印值及地址
$(gdb)quit(q) # 退出 gdb
$(gdb)break+num(b) # 在第 num 行设置断点
$(gdb)info breakpoints # 查看当前设置的所有断点
$(gdb)delete breakpoints num(d) # 删除第 num 个断点
$(gdb)display # 追踪查看具体变量值
$(gdb)undisplay # 取消追踪观察变量
$(gdb)watch # 被设置观察点的变量发生修改时，打印显示
$(gdb)i watch # 显示观察点
$(gdb)enable breakpoints # 启用断点
$(gdb)disable breakpoints # 禁用断点
$(gdb)x # 查看内存 x/20xw 显示 20 个单元，16 进制，4 字节每单元
```

根据以上信息与以下三段需要您进行调试的代码，我们需要以下材料以证明您完成了对应的工作:

1. 对于代码片段一，分别追踪前后两次调用函数 sumOfSquare() 时函数调用的栈帧与层级关系，并给出过程的相关截图
2. 对于代码片段二，我们希望您能够追踪每次循环的变量 y 的具体变量值，并思考最后打印的 j 数值的意义
3. 对于代码片段三，我们希望您能根据调试与代码中的提示修改代码让其正常运行，并告诉我们 const,enum，define 三者中哪一个有地址，并将其地址打印出来。

```cpp
//test_function_overload.cpp
#include <iostream>
using namespace std;

int sumOfSquare(int a, int b) {
    return a * a + b * b;
}

double sumOfSquare(double a, double b) {
    return a * a + b * b;
}

int main() {
    int m, n;
    cout << "Enter two integer: ";
    cin >> m >> n;
    cout << "Their sum of square: " << sumOfSquare(m, n) << endl;

    double x, y;
    cout << "Enter two real number: ";
    cin >> x >> y;
    cout << "Their sum of square: " << sumOfSquare(x, y) << endl;

    return 0;
}
```

```cpp
//test_range_based.cpp
//基于范围的 for 循环来循环访问数组和矢量
#include <iostream>
#include <vector>
using namespace std;

int main()
```

```
{
    // Basic 10-element integer array.
    int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // Range-based for loop to iterate through the array.
    for( int y : x ) { // Access by value using a copy declared as a specific type.
                       // Not preferred.
        cout << y << " ";
    }
    cout << endl;

    // The auto keyword causes type inference to be used. Preferred.
    for( auto y : x ) { // Copy of 'x', almost always undesirable
        cout << y << " ";
    }
    cout << endl;

    for( auto &y : x ) { // Type inference by reference.
        // Observes and/or modifies in-place. Preferred when modify is needed.
        cout << y << " ";
    }
    cout << endl;

    for( const auto &y : x ) { // Type inference by const reference.
        // Observes in-place. Preferred when no modify is needed.
        cout << y << " ";
    }
    cout << endl;
    cout << "end of integer array test" << endl;
    cout << endl;

    // Create a vector object that contains 10 elements.
    vector<double> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(i + 0.14159);
    }

    // Range-based for loop to iterate through the vector, observing in-place.
    for( const auto &j : v ) {
        cout << j << " ";
    }
```

```
    cout << endl;
    cout << "end of vector test" << endl;
}
```

```
//test_const
#include <iostream>
using namespace std;
struct A {
#define p "hello"
};

class C {
public:
    // static const 可以直接在类内部初始化
    // no-const static 只能在外面初始化
    static const int NUM = 3; //声明
    enum con {
        NUM1 = 3
    };
};

#define MAX(a,b) ((a) > (b) ? (a) : (b))
template<typename T>
inline int Max(const T& a, const T& b){
    return (a>b ? a:b);
}
const int C::NUM;   // 定义
int main() {
    cout << p << endl; // macro is global
    C c;
    cout << &c.NUM << endl;     // 未定义的引用, 需要定义
    cout << C::NUM1 << endl;
    cout << &C:NUM1 << endl;  //error enum no address

    int a=5, b=0;
    cout<<MAX(++a, b)<<endl;              // a 被增加两次
    cout<<MAX(++a, b+10)<<endl;           // a 被累加一次
    a=5,b=0;
    cout<<Max(++a,b)<<endl;
}
```