第一次作业

A. 选择

1. 下列关于声明与定义正确的有

   A. `float a;` 语句只声明了变量 `a`；

   B. 同一个函数只能够被声明一次，定义一次；

   C. 一个只被声明，没有被定义的函数，若该函数没有被调用，程序也能够被正常编译成可执行文件；

   D. g++编译中 `-c` 表示只编译不链接。

2. 下面说法正确的有

   A. 运行 `make` 命令没有指定任务时，默认会执行所有任务；

   B. makefile中每行命令的行首应有tab或者四个空格；

   C. 头文件引入必须写在源文件的开头；

   D. 为了在多个源文件中使用同一个函数，我们可以将该函数在头文件中声明来避免错误。

3. 设有如下宏定义，请问 `int z = 2 * (N + FUNC(5 + 1 >> 1))` 的值为

```
#define N 3
#define FUNC(n) (N+1)*n
```

   A. 24

   B. 26

   C. 46

   D. 48

4. 执行 `g++ -o main main.cpp` 时，传递给 `g++` 程序的命令参数 `argv[2]` 为

   A. g++

   B. main.cpp

   C. -o

   D. main

5. 以下类定义不能通过编译的是

A.

```
class A {
public:
    void func(int a,int b) {};
};
```

B.

```
class B {
public:
    void func(int a,int b=1){}
};
```

C.

```
class C {
public:
    void func(int a=3,int b){}
};
```

D.

```
class D {
public:
    void func(int a=3,int b=1){}
};
```

6. 下面说法正确的有

A. 编译器认为某个函数不值得内联，就会忽略内联修饰符；

B. 函数名相同，形参的个数或其中一个参数的类型不同，可以实现函数重载；

C. 返回值不同可以用来区分重载函数；

D. 内联函数的代码会在运行时被插入在每个调用该函数的地方。

7. 以下哪些描述属于下面代码无法通过编译的原因

```
class P {
    int data = 1;
    void func(P a);
    int func(int a) {data += a; return data;}
    int func(int i, int j=2) { data += i + j; return data;}
};
void P::func(P a) {  a.data += data; }

int main(){
    P a;
    a.func(1);
    return 0;
}
```

A. 没有设置成员变量、成员函数的访问权限；

B. 函数调用 `a.func(1)` 存在二义性；

C. 类P的 `func(P a)` 中修改了参数 `a` 的私有成员变量；

D. `main` 函数中调用了 `a` 的私有成员函数。

8. 下列说法正确的有

A. 假设A为一个类，a为该类的公有成员变量，则在该类的非静态成员函数中，可以通过 `(*this).a` 来访问成员变量 `a`；

B. 假设A为一个类， `auto x = new A[5];` 和 `auto* x = new A[5];` 中变量 `x` 的类型相同，都是 `A*`；

C. 代码 `auto a = 5, b = 'r'` 可以通过编译，并推断出 `a` 为 `int` 型， `b` 为 `char` 型；

D. 内联函数一般用于加速程序。

```
CD
D
A
D
C
AB
BD
ABD
```

B. Makefile条件编译

题目描述

- 现有一个工程目录, 其中有4套 `.cpp` 和 `.h` 文件（名称分别为 `f1.cpp` 和 `f1.h`， `f2.cpp` 和 `f2.h`， `f3.cpp` 和 `f3.h`， `f4.cpp` 和 `f4.h`），分别对应实现了一个能够在屏幕打印输出的功能函数 `f1`，`f2`，`f3`，`f4`。另外 `main.cpp` 实现了 `main` 函数。
- 文件内容见下载链接。
- 我们希望通过在make目录下编写一个 `Makefile` 文件实现条件编译。编译条件由 `control.mak` 文件（系统提供）给出。下面给出 `control.mak` 的一个样例：

```
# 以下设置使得编译链接后的main.cpp只用到f1和f2函数，而不会用到f3和f4函数
# f1函数开关：开
F1 = TRUE
# f2函数开关：开
F2 = TRUE
# f3函数开关：关
F3 = FALSE
# f4函数开关：关
F4 = FALSE
```

要求:

1. 根据题目所给文件编写完成 `Makefile` 文件，使得运行 `make` 命令可以在当前目录下生成 `main` 可执行文件（无后缀），系统会运行 `./main` 并根据提供的 `control.mak` 检查输出结果。
    - `control.mak` 中确保存在一个函数的开关状态为开
2. 编译时，需要采取每个源文件编译成目标文件，再进行链接的方式。即需在当前目录下生成**同前缀名的** `.o` 的目标文件（如，`f1.o`，`f2.o`，`main.o`），再将 `.o` 文件链接，形成 `main` (无后缀)可执行文件。该可执行文件在运行时会输出相关信息。
3. 使用 `make clean` 命令，能够清理 `Makefile` 生成的 `.o` 文件（包括历史生成的文件），还原成最初的样子(即没有 `.o` 文件和可执行文件)。
4. 如果 `control.mak` 中定义的函数开关发生变化，在不使用 `make clean` 的情况下，再次执行 `make` 命令需要能够生成相应的 `main` 可执行文件。

提交要求:

1. 你只需要提交 `Makefile` 文件, 满足上述需求。

提示:

1. 你需要使用 `include` 命令包含其他的 `makefile` 文件(`control.mak`)
2. `Makefile` 文件中可能涉及的 `if` 语句语法如下，用于判断 `var` 变量的值是否等于 `TRUE`

```
ifeq (TRUE, $(var))
    # do something
endif
```

3. `%`: `make` 命令允许对文件名，进行类似正则运算的匹配，主要用到的匹配符是 `%`。比如，假定当前目录下有 `f1.c` 和 `f2.c` 两个源码文件，需要将它们编译为对应的对象文件

```
%.o:%.cpp
```

相当于

```
f1.o:f1.cpp
f2.o:f2.cpp
```

其他更高级的操作可以参见 阮一峰的 `makefile` 教程 https://www.ruanyifeng.com/blog/2015/02/make.html

4. 可以在 `Makefile` 中进行简单的字符串操作，例如 `subst`

```
$(subst .cpp,.o, test.cpp)  # 结果是 test.o
```

5. 当然你可以不使用上面的提示，例如依次写下每个文件的编译命令，但是更建议你尝试练习 `%` 等方式进行批量编译。这不会作为评分标准。

C. 奶牛养殖场

题目描述

全自动化养殖场有n头奶牛，每头奶牛每天要吃不少于l kg且不多于u kg饲料，能够产奶m kg。养殖场为每头奶牛配备了一个**专属**智能饲料槽，每天定期自动往里补充a kg饲料。奶牛每天的进食在饲料补充后进行。每头牛总是提供多少饲料吃多少饲料，但智能饲料槽能够确保一头牛吃够u kg饲料后就不再提供饲料，剩余饲料留到之后使用（**仅限该头牛使用**）。产奶在进食之后进行，如果一头牛一天吃少于l kg饲料，则该天的产奶量减半；如果一头牛一天没有吃饲料，则该天的产奶量为0。你需要补充完成一个程序：根据k天的饲养情况，计算k天的产奶量之和。

输入格式

- 第一行是奶牛数n，接下来n行，每行包括一头奶牛的名字，以及相应的l，u和m，用空格隔开
- 接下来一行是天数k，后面k行，第i行行首是n'（$0 \leq n' \leq n$），表示第i天为n'头不同的奶牛补充了饲料，该行后面是用空格相间的n'项记录，每项纪录的格式为 `<奶牛名字> a`。比方说，`2 A 1 B 3`，意思是给名字为 `A` 的牛补充饲料1 kg，给名字为 `B` 的牛补充饲料3 kg
- 所有输入数字都是整数
- k天之前，饲料槽中的饲料剩余量为0

输出格式

- 输出k天的产奶量之和，保留一位小数

输入样例

```
2
a 2 5 6
b 3 4 7
2
2 b 6 a 2
1 a 1
```

输出样例

```
19.5
```

要求

- 在下列代码的基础上，编写Cow类（奶牛类）和Farm类（养殖场类）的代码，完成上述要求。
- main.cpp（可以在这里下载）

```
#include <iostream>
#include <string>
#include "Cow.h"
#include "Farm.h"
using namespace std;

int main(){
    int n;
    cin >> n;
    Farm farm(n);
    string name;
    int l, u, m;
    for(int i = 0; i < n; ++i){
        cin >> name >> l >> u >> m;
        Cow cow(name, l, u, m);
        farm.addCow(cow);
    }

    int k;
    cin >> k;
    int t;
    int a;
    for(int i = 0; i < k; ++i){
        cin >> t;
        for(int j = 0; j < t; ++j){
            cin >> name >> a;
            farm.supply(name, a);
        }
        farm.startMeal();
        farm.produceMilk();
    }
    printf("%.1f", farm.getMilkProduction());
    return 0;
}
```

提交格式

- 你需要提交多个文件，包含Makefile，上述文件调用的各种头文件及其cpp文件；可以不包括提供的main.cpp文件。使用Makefile必须要能生成可执行文件main（不带扩展名）。
- 你应该将你的文件打包成一个zip压缩包并上传。**注意：你的文件应该在压缩包的根目录下，而不是压缩包的一个子文件夹下。**评测时，OJ会将提供的main.cpp贴入你的目录下进行编译并执行。

评分标准

- $1 \leq k, n \leq 100$
- $1 \leq m, a \leq 100$
- $1 \leq l \leq u \leq 100$
- 时限1s，OJ评分占100%。

D. 命令行参数解析

题目描述

Python 语言中有一个十分好用的命令行参数解析库：`argparser`

通过如下代码，用户可以为程序 `parser.py` 添加想要的命令行选项：

```python
import argparse

parser = argparse.ArgumentParser()

parser.add_argument("--one", default=1, type=int, help="first number")
parser.add_argument("--two", default=1, type=int, help="second number")

args = parser.parse_args()

print(args.one)
print(args.two)
print(args.one + args.two)
```

使用时，可以通过命令行选项的名称指定相应的参数：

```
python parser.py --one 2 --two 3
```

输出：

```
2
3
5
```

本题的目标是使用 C++ 实现一个简化版的 `argparser`。测试样例代码在 `main.cpp` 文件的 `main` 函数中。测试代码如下（此样例代码可以从这里下载。）：

```cpp
#include <iostream>
#include "Parser.h"

int main(int argc, char *argv[]) {
    Parser parser = Parser();

    parser.add_argument("--one", 1, "First number");
    parser.add_argument("--two", 2, "Next number");
    parser.add_argument("--three", 3, "Third number");

    parser.init(argc, argv);

    int a = parser.get_argument("one");
    int b = parser.get_argument("two");
    int c = parser.get_argument("three");

    std::cout << a + b * c << std::endl;

    return 0;
}
```

我们会先构造一个 `Parser` 对象，向其中添加几个命令行选项。然后调用 `parser.init`，其中会处理程序真实的命令行输入，并根据选项名称提取出参数并保存下来。当然，这里还需要进行参数不合法时的错误处理。最后我们会访问设置的命令行选项。

为了简单，我们只考虑输入参数为整数的情况。你可以假设测试时的选项名称全部以 "--" 开头，并且 `get_argument` 中的参数全部是添加过的。

评测样例

以下的样例包含了我们考虑的所有正常和错误情况。

- `--help` 显示帮助信息（**注意：传入的参数都是小写，如 `one`；在下面输出的 `usage` 中用参数字符串的大写来代替它的值，如 `ONE`**）

```
OOP@ubuntu$ ./main --help
usage: ./main [--help] [--one ONE] [--two TWO] [--three THREE]

optional arguments:
 --help   show this help message and exit
 --one ONE   First number
 --two TWO   Next number
 --three THREE   Third number
```

- 正常输出结果

```
OOP@ubuntu$ ./main --one 3 --two 3 --three 2
9
```

- 未指定参数，使用默认值

```
OOP@ubuntu$ ./main --one 3 --three 2
7
```

- 使用未添加的参数

```
OOP@ubuntu$ ./main --fourth 3
usage: ./main [--help] [--one ONE] [--two TWO] [--three THREE]
./main: error: unrecognized arguments: --fourth
```

- 参数未提供值

```
OOP@ubuntu$ ./main --one
usage: ./main [--help] [--one ONE] [--two TWO] [--three THREE]
./main: error: argument --one: expected an argument
```

- 参数后面不是整数 #1

```
OOP@ubuntu$ ./main --one --two 3
usage: ./main [--help] [--one ONE] [--two TWO] [--three THREE]
./main: error: argument --one: invalid int value: '--two'
```

- 参数后面不是整数 #2

```
OOP@ubuntu$ ./main --one aa --two 3
usage: ./main [--help] [--one ONE] [--two TWO] [--three THREE]
./main: error: argument --one: invalid int value: 'aa'
```

- 多个错误，按第一个错误输出

```
OOP@ubuntu$ ./main --one aa --fourth 4
usage: ./main [--help] [--one ONE] [--two TWO] [--three THREE]
./main: error: argument --one: invalid int value: 'aa'
```

- 只要使用了 `--help` 选项，则忽略其他选项（即使其它选项有错误）

```
OOP@ubuntu$ ./main --help --one 1
usage: ./main [--help] [--one ONE] [--two TWO] [--three THREE]

optional arguments:
 --help   show this help message and exit
 --one ONE   First number
 --two TWO   Next number
 --three THREE   Third number
```

```
OOP@ubuntu$ ./main --fourth 4 --help
usage: ./main [--help] [--one ONE] [--two TWO] [--three THREE]

optional arguments:
 --help   show this help message and exit
 --one ONE   First number
 --two TWO   Next number
 --three THREE   Third number
```

提示

1. 在实现过程中，你可能需要实现字符串到整数和到字符串的映射。实现这种映射的方式有多种：（1）可以实现一个 `Argument` 类，维护加入到 `parser` 对象中的每个参数；（2）你也可以使用 C++ STL 库中的 `std::map`，通过 `std::map<std::string, int>` 与 `std::map<std::string, std::string>` 可以分别构造字符串到整数和到字符串的映射。具体使用方法可以查阅 http://www.cplusplus.com/reference/map/map/ 。当然你也可以采用其他的具体实现方式。**注意：本题的评判不对具体实现作硬性要求。**
2. 在处理错误的过程中可能需要直接退出程序。可以直接使用 `exit(0)`（注意，不管是否出错都应该返回0，否则会被判定为 `Runtime Error`）。

提交格式

- 你需要提交多个文件，包含Makefile，上述文件调用的各种头文件及其cpp文件；可以不包括提供的main.cpp文件。使用Makefile必须要能生成可执行文件main（不带扩展名）。
- 你应该将你的文件打包成一个zip压缩包并上传。**注意：你的文件应该在压缩包的根目录下，而不是压缩包的一个子文件夹下。**评测时，OJ会将提供的main.cpp贴入你的目录下进行编译并执行。

第二次作业

A. 选择

1. 下列说法错误的是 []

   A) 运算符重载要求编译器能够唯一地确定调用哪个运算符实现代码;

   B) 可以通过重载运算符修改运算的优先级

   C) 如果运用恰当，运算符重载可以使得代码变得更加直观简洁

   D) 若重载运算符时参数列表中仅有1个参数，则该运算符为双目运算符

2. 下列说法错误的是 []

   A) 输出流运算符重载函数的参数列表中使用目标对象的引用，可以避免由于对象复制而造成的额外开销

   B) 如果重载自定义类 `Test` 的输出流运算符，则函数签名必须是 `ostream & operator<<(ostream &, const Test &);`

   C) 若表达式 `a++` 中的"++"是作为成员函数重载的运算符,则与 `a++` 等效的运算符函数调用形式可以为 `a.operator++(n)` ，其中 `n` 为任意整数

   D) 运算符重载只能通过类成员函数实现

3. 下列说法错误的是 []

   A) 类的默认构造函数如果没有被用户显式定义，则一定会被编译器隐式合成

   B) 全局对象在 `main()` 函数调用之后立刻被构造，在 `main()` 函数执行完之前不会被析构

   C) 引用在声明后，可以像指针一样更改指向的对象

   D) 可以通过给析构函数传入参数实现在析构时期待的特定行为

4. 关于类A和类B的说法正确的是[]

```
class B {
public:
   B(int i) {}
};

class A {
private:
   int a = 1;
   B b;   // (2)
public:
   A() = default; // (1)
   A(int i):a(i), b(i) {}
};
```

   A) (1)处使用 `default` 关键字指定生成默认构造函数，故在 `main` 函数中可以使用 `A a();` 来构造类A的对象

   B) 由于B类没有默认构造函数，(2)处 `b` 不能在声明时初始化

   C) A类的一个对象 `a` 执行析构时，会先调用类 `B` 的析构函数来析构a中的数据成员 `b` ，再调用类 `A` 的析构函数

   D) `main` 函数中可以使用 `A a('c');` 来构造类A的对象

5. 关于友元friend，以下说法错误的是 []

   A) 友元类必须在想要访问其私有成员的类内声明并实现

   B) 被声明为当前类的友元的函数一定是全局函数

   C) 类的析构函数不能被声明为别的类的友元函数

   D) 如果在 `A` 的类内写有 `friend class B` ;那么 `A` 的所有成员函数均能访问 `B` 的所有成员。

6. 下列说法不正确的是 []:

   A) 常量对象既能调用常量成员函数，也能调用非常量成员函数，但在调用时不能修改对象状态

   B) 非常量对象的常量成员函数能访问不修改对象状态的非常量成员函数

   C) 如果在 `a.h` 中声明并定义全局静态变量 `static int v;` ， `a.h` 被 `b.cpp` 和 `c.cpp` 同时包含(即 `#include "a.h"`)，则同时编译这三个文件可能会因为多次定义同一个静态变量 `v` 而导致发生编译失败

   D) 常量静态数据成员都只能在类外初始化

7. 下列说法正确的是 []:

   A) 静态全局对象、常量全局对象都是在进入 `main` 函数之前构造，执行完 `main` 函数以后析构

   B) 若类A的对象a是类B的静态成员变量，则a在程序执行到它被第一次被访问时初始化

   C) 函数返回局部对象的引用可能导致运行时错误

   D) 用new[]构造的对象必须用delete[]释放内存，否则可能会造成内存泄露

8. 定义如下的Test类，下列说法正确的是 []

```
class Test{
   const int member1;
   static float member2;
public:
   Test(int mem):member1(mem){}
   int MyMember1() const {return member1;}
   static float Mymember2() {return member2;}
};
float Test::member2 = 0;
```

   A) member1可以像member2一样在类外初始化

   B) member1的值在不同的Test对象中可以不同，但不同的Test对象只能访问同一个member2

   C) 成员函数MyMember1的函数体内可以增加语句，修改member2的值

   D) 定义一个Test类的常量对象，可以调用MyMember1和MyMember2两个成员函数

```
BD
BD
ABCD
D
ABCD
ABCD
ACD
BCD
```

B. 类的声明、构造与析构

有一段不完整的程序，你需要在(1)~(9)处填入代码，并使得程序的输出和标准输出(stdout.txt)一致，文件内容见下载地址。

Example.h

```
#ifndef __EXAMPLE__
#define __EXAMPLE__

class Example {

private:

    (1)

public:

    Example(int data);
    void getData();
    ~Example();
};

#endif
```

Example.cpp

```
#include <iostream>
#include "Example.h"

using namespace std;

    (2)

Example::Example(int data) {
    (3)
}

void Example::getData() {
    (4)
}

Example::~Example() {
    (5)
}
```

main.cpp

```
#include <iostream>
#include "Example.h"

using namespace std;

void create_example(int n) {

    cout << "--------create_example is ready--------\n";

    Example* e[10];

    for (int i = 0; i < n; i++)
        e[i] = new Example(i);

    (6)

    cout << "--------create_example is over--------\n";
}

    (7)

int main() {

    cout << "--------main_function is ready--------\n";

    (8)

    create_example(3);

    (9)

    cout << "--------main_function is over---------\n";

    return 0;
}
```

stdout.txt

```
Example #1 is created
--------main_function is ready--------
Example #2 is created
--------create_example is ready--------
Example #3 is created
Example #4 is created
Example #5 is created
Example #6 is created
The data of Example #3 is 0
The data of Example #4 is 1
The data of Example #5 is 2
Example #3 is destroyed
Example #4 is destroyed
Example #5 is destroyed
The data of Example #6 is 2333
--------create_example is over--------
Example #7 is created
The data of Example #1 is 23
The data of Example #2 is 233
The data of Example #7 is 23333
--------main_function is over---------
Example #7 is destroyed
Example #2 is destroyed
Example #6 is destroyed
Example #1 is destroyed
```

说明：在类实例化时，构造函数需输出“Example #XXX is created”。在析构时，需输出“Example #XXX is destroyed”。在调用 getData() 时，需输出“The data of Example #XXX is XXXX”。其中“#XXX”是实例的编号，编号从1开始累计增长，XXXX是实例里的数据。

提示：建议在类中增加一个成员变量，记录类已经有了多少实例了。建议在类中再增加一个成员变量，记录实例的编号。

注意：我们会检测你是否只是添加了代码，所以切记不要在(1)~(9)之外的地方改动、添加代码。若系统检测到你修改了其他位置的代码，会显示例如 `Wrong Answer at Example.cpp` 的信息。

编译选项

g++ main.cpp Example.cpp -o main -lm -O2 -DONLINE_JUDGE --std=c++14

提交方式

提交三个文件：填好后的 Example.h、Example.cpp、main.cpp。

评分标准

OJ评分占100%。

C. Map封装

题目描述

你需要在这份代码的基础上帮忙完善 `Map` 类的实现（补充 `Map.h` 中 `TODO` 部分，并在 `Map.cpp` 中实现相关函数）。`Map` 类是一个能够实现 `std::string` 到 `int` 的映射的容器类，具有如下特性：

- 一个 `Map` 对象含有两个数据成员如下：

```
#pragma once
#include "Pair.h"

class Map{
    Pair * data;
    int sz;
public:
    Map(int n);
    // TODO
};
```

构造函数 `Map(int n);` 的形参 `int n` 指示被创建的 `Map` 对象需要能够容纳不多于 `n` 个键值对，一个键值对由 `Pair` 类的一个对象维护。你需要在构造函数内开辟一个大小为 `n` 个 `Pair` 对象的数组，并让数据成员 `data` 指向该数组的首位。

数据成员 `sz` 表明 `Map` 对象实际包含的键值对数量（必有$0 \leq$ `sz` $\leq$ `n`）。

- 假定 `s` 是一个 `string` 对象，`map` 是一个 `Map` 对象，则调用 `map[s]` 满足如下性质：
  - 如果 `map` 是常量且 `s` 是 `map` 的一个键：`map[s]` 返回 `s` 对应的值，只能读，不能写
  - 如果 `map` 是常量且 `s` 不是 `map` 的一个键：`map[s]` 返回默认值 `0`
  - 如果 `map` 不是常量且 `s` 是 `map` 的一个键：`map[s]` 返回 `s` 对应的值，能读写
  - 如果 `map` 不是常量且 `s` 不是 `map` 的一个键：`map[s]` 将 `s` 和默认值 `0` 配对添加到 `map` 中，并返回 `s` 对应的值，能读写

输入输出

- `main.cpp` 文件读取输入并构造一个 `Map` 类对象 `map`（及其常量引用 `cmap`）进行测试；
- 输入第一行是两个用空格隔开的整型数：`n` 和 `k`，其中 `n` 表示被测试的 `Map` 对象的最大容纳量为 `n` 个键值对，`n` > 0；
- 输入后面是 `k` 行，每行是如下三种格式中的一种：
  - `1 [key]`：查询并输出 `map` 中键为 `[key]`（字符串）对应的值，输出占一行
  - `2 [key] [val]`：将 `map` 中键为 `[key]` 的值改为 `[val]`（整型数）
  - `3 [key]`：查询并输出 `cmap` 中键为 `[key]`（字符串）对应的值，输出占一行
- 最后输出 `map` 实际包含的键值对数量。

提交格式

- 你需要提交多个文件，包含 `Map.h` 和 `Map.cpp`，可以不包括提供代码中除Map.h以外的其他文件。
- 你应该将你的文件打包成一个zip压缩包并上传。**注意：你的文件应该在压缩包的根目录下，而不是压缩包的一个子文件夹下。**评测时，OJ会将除 `Map.h` 和 `Map.cpp` 之外的评测所需的文件（包括 `main.cpp` 等）贴入你的目录下进行编译并执行。

D. 疫情防控系统

题目描述

由于新冠疫情的肆虐，学校要求每一位同学都上报自己的个人信息以及返校时间。并能够根据省份、学号、姓名进行信息查询。要求：

1. 从命令行读取每个同学的`[姓名]-[省份]-[年级]-[返校时间]-[学号]`，其中姓名是连续的英文字母，姓名长度不超过20，省份为一个大写的字母，年级为1-4数字中的一个（表示大一至大四），学号为6位数字（且**第一位不为0**），返校时间为4位数字（表示月份及日期，`0301`表示3月1号返校）。输入的姓名和学号分别都具有唯一性；
2. 能够根据给定省份、学号、或姓名，输出相关同学的信息

输入输出要求

- 输入第一行是学生人数`n`，接下来`n`行是学生的信息（$n>0$）；
- 然后是查询数`m`，接下来`m`行，每行一个数字，一个查询值（**保证不为空**）。对于每一个查询：
  - 数字`0`表示按学号查询，`1`表示按姓名查询，`2`表示按省份查询。**输入保证合法**
  - 当查询值不存在时，输出"Not Found"
  - 对符合查询信息的同学，输出格式为：`[姓名]-[返校时间]-[年级]-[学号]`
  - 符合查询信息的同学如果有多个，则输出**返校时间最晚的同学里学号最大**的同学的信息（学号可看作数字进行大小比较）

输入样例

```
4
ZangZY-A-1-0221-456789
AnBY-C-4-0302-211306
NaBA-A-4-0000-111222
EaaFG-B-3-0201-344233
3
2 A
1 AnBY
0 456781
```

输出样例

```
ZangZY-0221-1-456789
AnBY-0302-4-211306
Not Found
```

要求

在下列代码的基础上，编写`Student`类和`School`类，完成上述任务。要求实现**比较符重载和流运算符重载。**

`main.cpp`可从这里下载

```
#include <iostream>
#include "School.h"
#include "Student.h"

using namespace std;
int main(){
    int n;
    cin >> n;
    School tsinghua(n);
    for (int i = 0; i < n; ++ i){
        Student s;
        cin >> s;
        tsinghua.add_member(s);
    }

    int m;
    cin >> m;
    for (int i = 0; i < m; ++ i){
        int type;
        cin >> type;
        if (type == 0){
            int sid;
            cin >> sid;
            cout << tsinghua[sid];
        }
        else if (type == 1) {
            string name;
            cin >> name;
            cout << tsinghua[name];
        }
        else if (type == 2){
            char province;
            cin >> province;
            cout << tsinghua[province];
        }
    }
    return 0;
}
```

提交格式

你可以写多个文件，但必须包含一个Makefile文件、上述文件调用的各种头文件；可以不包括提供的`main.cpp`文件。你应该将你的文件打包成一个zip压缩包并上传。评测时，OJ会将提供的`main.cpp`贴入你的目录下进行编译并执行。

第三次作业

A. 选择

1. 下面说法正确的有:

   A) 若已定义类 `Obj`，则对于 `Obj a; Obj b=a;`，第二个语句首先调用 `Obj` 的默认构造函数（已定义）初始化 `b`，再调用 `Obj` 的赋值运算符（已定义）实现 `a` 到 `b` 的拷贝。

   B) 如果一个类没有显式定义拷贝构造函数但重载了赋值运算符，且编译器自动合成了拷贝构造函数，则合成的拷贝构造函数会调用拷贝赋值运算符完成拷贝。

   C) `std::move` 函数实际上只完成类型转换，不对对象做其余操作。

   D) 若有 `int x;`，则 `int *y = &x;` 和 `int *z = &(x+5)` 都是合法的语句。

2. 以下代码的输出是:

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


class T{
public:
    T(){}
    T(T& t){cout<<"A";}
    T(T&& t){cout<<"B";}
    T& operator =(T&& t){cout<<"C"; return *this;}
};

void swap(T& a, T b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}

int main(){
    T a;
    T b;
    swap(a,b);
    return 0;
}
```

   A) AABCC

   B) ACCC

   C) ABCC

   D) ABBCBC

3. 关于以下代码说法正确的有:

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

class Complex
{
public:
    int real;
    int imag;
    Complex():real(0),imag(0){}
 Complex(int r, int i):real(r),imag(i){}
 Complex(int r): real(r),imag(0){cout<<"A";}
 operator int(){
     return real;
 }
 Complex operator+(const Complex& c){
     return Complex(this->real+c.real,this->imag+c.imag);
 }
};

int main()
{
    Complex c(3);
    c=1.2;
    c=c+3.4;
    cout<<c.real;
    return 0;
}
```

   A) 该代码在第24行产生歧义会导致编译错误。可以将12-14行的代码注释或15-17行的代码注释，两种情况均可编译通过。

   B) 如果注释15-17行，并且在12行前加 `explict` 关键字，则第24行不可以通过编译。

   C) 如果注释12-14行，输出结果是 `AAA4`。

   D) 如果注释15-17行，输出结果是 `AAA4`。

4. 以下代码的输出是:

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

class A {
    char c = 'c';
public:
    A() { cout << "A"; }
    A(char X):c(X){ cout << "X"; }
    ~A() { cout << c; }
};

class B: public A {
    A a{'x'};
public:
    B() { cout << "B"; }
    ~B() { cout << "b"; }
};

int main() {
    B b;
    return 0;
}
```

A) XABbcx

B) AXBbxc

C) BXabxc

D) XABxcb

5. 关于以下代码说法正确的有:

```
#include <iostream>
using namespace std;
class A {
public:
    int a=1;
protected:
    int b=2;
private:
    int c=3;
};

class B {
public:
    int d=4;
protected:
    int b=5;
private:
    int e=6;
};

class C: public A, private B{
public:
    void print() {
        cout << [1] << endl;
    }
};

int main() {
    C obj_c;
    obj_c.print();
    cout << [2] << endl;
    return 0;
}
```

A) 在[1]处，填 `a` 或 `d` 均可编译通过。

B) 在[2]处，填 `obj_c.a` 或 `obj_c.d` 均可编译通过。

C) 在[1]处，可以通过 `A::b` 和 `B::b` 分别访问 `A` 类和 `B` 类中的成员变量 `b` 。

D) 在[2]处，可以通过 `obj_c.A::b` 来访问 `A` 类中的成员变量 `b` 。

6. 关于下面这段代码运行结果说法正确的有 (在编译选项含有 `-std=c++11`):

```
#include <iostream>
using namespace std;
class A {
    int data;
public:
    static int count;
    A():data(2019){count += 1; cout << count << endl;}
    A& operator = (const A & a){
        if (this != &a) data = a.data;
       return *this;
    }
    A(int i):data(i){count += 2; cout << count << endl;}
    ~A(){cout << "destructor "<< data << endl;}
};

class B {
    int data{2020};
    A a1,a2;
public:
    B(){}
    B(int i):a2(i){a1 = a2;}
    ~B(){cout<<"destructor "<< data <<endl;}
};

int A::count = 0;

int main() {
  B obj1;
  B obj2(2021);
  return 0;
}
```

A) 将第8-11行去掉对程序运行结果不会有影响。

B) 输出的前4行是 `1\n2\n4\n5\n` 。

C) 将第20行去掉对程序编译不会有影响。

D) 输出中前缀是 `destructor` 的行，后面的数字按顺序是 `2020` , `2021` , `2021` , `2020` , `2019` , `2019` 。

7. 关于下面代码说法正确的有:

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

class Base0{
public:
    Base0(){};
    Base0(int d){};
    void f(double d) { cout << "Base0::f(" << d << ")\n"; }
    void g(double d) { cout << "Base0::h(" << d << ")\n"; }
};

class Base1: public Base0{
public:
    using Base0::f;
    void f() { cout << "Base1::f()\n"; }
    void f(int i) { cout << "Base1::f(" << i << ")\n"; }
};

class Derive : public Base1{
public:
    using Base1::f;
    void f(int i) { cout << "Derive::f(" << i << ")\n"; }
};

int main() {
    Derive d;
    d.f(10);
    d.f(4.9);
    d.g(4.8);
    d.f();
    return 0;
}
```

A) 如果将6行注释掉, 程序将不能通过编译。

B) 如果将第14行注释掉, 程序仍能通过编译,但是输出有变化。

C) 如果将第21行注释掉, 程序仍能通过编译,但是输出有变化。

D) 如果将第12行改为 `class Base1: protected Base0{` , 只需要去掉第29行就可以成功编译运行。

8. 以下说法正确的有:

A) 派生类的成员函数不能访问通过私有继承的基类的保护成员。

B) 继承时，除了构造函数，析构函数之外的基类成员函数都可以被派生类继承。

C) 类 `D` 直接继承类 `B` 和类 `C` ，而类 `B` 和类 `C` 分别直接继承类 `A` ，则类 `D` 的对象访问类 `A` 的数据成员时可能出现二义性。

D) 派生类的构造函数的成员初始化列表中，不能包含基类数据成员的初始化。

```
C
C
ABCD
B
AC
AD
ABD
CD
```

B. 引用？复制？

1.有一段缺损的程序，其中缺少三个全局函数f1,f2和f3的实现。本题需要观察stdout.txt中输出结果隐含的逻辑关系，然后根据这些关系将这三个函数的返回值类型、参数类型和函数体补充至func.h中。程序的其他部分（包括main.cpp，Test.h）均不能修改，补充的答案不唯一，任意一种即可。

文件下载：下载链接

文件说明：main.cpp中主要是对函数f1,f2和f3的调用，部分代码如下：

```
#include <iostream>
#include "Test.h"
#include "func.h"

using namespace std;

int main()
{
    cout << "------entering main------" << endl;
    Test a;
    Test b;

    cout << "------before call f1------" << endl;

    cout << "f1():" << endl;
    Test A = f1(a);

    cout << "------after f1 return------" << endl;
......
```

stdout.txt包含程序的输出，主要反映了各类构造函数、析构函数的调用情况，部分内容如下：

```
------entering main------
Test(): this->buf @ 0x1e1550
Test(): this->buf @ 0x1e1570
------before call f1------
f1():
Test(const Test&) called. this->buf @ 0x1e1590
a.buf @ 0x1e1590
Test(Test&&) called. this->buf @ 0x1e1590
Test(Test&&) called. this->buf @ 0x1e1590
~Test(): this->buf @ 0
~Test(): this->buf @ 0
......
```

2.下列程序是否存在编译错误或潜在风险？若没有错误，请说明构造函数、复制构造函数、移动构造函数、赋值运算、移动赋值运算被调用的位置及次数。若有编译错误或潜在风险，请指出**所有**错误和风险，说明理由。

说明：Test类的定义和题目1相同，F是函数名

```
(1)
Test F(Test a){
    Test b = std::move(a);
    return b;
}
int main(){
    Test a;
    a = 1;
    Test A = F(a);
    return 0;
}

(2)
Test F(const Test& a){
    Test b = std::move(a);
    return b;
}
int main(){
    Test A = F(1);
    return 0;
}

(3)
Test F(Test &&a){
    Test b = std::move(a);
    return b;
}
int main(){
    Test A = F(1);
    return 0;
}

(4)
Test&& F(Test &a){
    Test b = a;
    return std::move(b);
}
int main(){
    Test A = F(Test(1));
    return 0;
}

(5)
const Test& F(const Test& a){
    Test b = a;
    return Test(1);
}
int main() {
    Test a;
    const Test &A = F(std::move(a));
    return 0;
}
```

编译选项

```
g++ main.cpp -o main -lm -O2 -DONLINE_JUDGE -fno-elide-constructors -std=c++11
```

提交文件

提交两个文件。

第一个是 func.h，要求该文件能和提供的其他文件共同编译，且输出结果中的内存申请情况应与标准输出相同（具体的内存地址可以不同）。

第二个是一个txt文件，回答第二题。

C. Vector类的简易实现与封装

本题需要你在给定代码的基础上，实现类 `Vector`。

`Vector` 是一种动态数组，可以数据储存的过程中动态调整其容量大小。具体来说，给定 `Vector` 的接口如下：

```
class Vector {
private:
    int capacity; //容量，即该对象最多可以储存多少个元素
    int len;      //当前元素长度
    Node* array;  //指向元素数组头的指针
public:
    Vector(int n); //初始化Vector，容量设定为n。
    ~Vector();
    Vector(const Vector & other);
    Vector(Vector && other);
    Vector & operator=(const Vector & other);
    Vector & operator=(Vector && other);
    Node& operator [] (int pos);
    void append(int value);   //在数组的尾部添加一个指定值的元素，即Node(value)。保证插入后长度不会大于容量。
    void insert(int pos, int value);  //在数组的指定位置插入新元素(即Node(value))，其他元素顺次右移。保证插入后长度不会大于容量。
    void swap(int pos1, int pos2);    //交换数组中指定位置的元素
    void dilatation();                //将目前Vector中的数组容量翻倍。提示：你需要先申请更大的数组，然后将当前元素信息移动过去。
    int getlen();                     //获取当前Vector的长度。
};
```

`Vector` 类的元素是 `Node` 类的对象，具体可见下载文件中的 `Node.h` 和 `Node.cpp`。

我们将使用测试函数来判断你的实现是否正确，样例测试代码见 `main.cpp` 中 `test1` 函数和 `test2` 函数，从标准输入读入，处理并产生相应输出，最后由输出结果判定你的结果是否正确。

在测试函数结束时，我们会调用 `Node::outputResult()` 函数来输出所有 `Node` 类对象的普通构造1（`Node()`）、普通构造函数2（`Node(int v)`）、拷贝构造、移动构造、拷贝赋值、移动赋值、析构的次数。通过这些次数来验证你的实现的正确性和效率。实现正确的 `Vector` 类不应该存在内存泄露，即满足：

普通构造1+普通构造函数2+拷贝构造+移动构造=析构

我们假定普通构造函数1由于没有进行显式赋值操作（默认值），该函数耗时较短。普通构造2、拷贝构造和拷贝赋值操作消耗时间较长，移动构造和移动赋值消耗时间较短，我们将使用如下公式简单地验证效率：

(普通构造2+拷贝构造+拷贝赋值)*10+(普通构造1+移动构造+移动赋值)≤参考答案

我们将按照如下方式对你的程序评分：

1.程序输出结果正确，但存在内存泄漏，获得50%的分数；

2.程序输出结果正确，不存在内存泄漏，但效率不满足要求，获得70%的分数；

3.程序输出结果正确，不存在内存泄漏，且满足效率要求，获得100%的分数。

提示：只要合理利用移动构造和移动赋值，即可通过效率测试。

输入格式

第一行包含一个正整数 `ref_ans`，表示效率的参考值。

第二行包含一个正整数 `n`，表示 `Vector` 的初始容量。

第三行包含一个正整数 `m`。

接下来 `m` 行，每行一个整数，按顺序加入 `Vector` 数组中。

第 `m+4` 行包含一个正整数 `k`，表示操作的个数。

接下来的 `k` 行，每行1到3个整数：`p (q) (v)`，表示操作的参数。

当 `p=0` 时，该操作为在尾部添加元素，将 `q` 加入到数组尾部。

当 `p=1` 时，该操作为插入，将 `v` 插入到数组的第 `q` 个位置。

当 `p=2` 时，该操作为交换，将数组中第 `q` 个元素和第 `v` 个元素交换位置。

当 `p=3` 时，该操作为长度加倍。

输入保证所有的操作都合法。

输入样例

```
272
6
5
0
1
2
3
4
4
0 5
3
1 4 8
2 3 0
```

样例解释：

```
272      // 效率参考值
6        // 初始容量
5        // 接下来是5个初始值，初始值为 0 1 2 3 4
0
1
2
3
4
4        // 接下来是4个操作
0 5      // 将5加入数组尾部，此时数组为 0 1 2 3 4 5
3        // 容量扩大为原来的两倍
1 4 8    // 将8插入数组的第4个位置，此时数组为 0 1 2 3 8 4 5
2 3 0    // 将数组的第3个元素和第0个元素交换位置，此时数组为 3 1 2 0 8 4 5
```

输出样例

```
0 1 2 3 4 5
0 1 2 3 4 5
0 1 2 3 8 4 5
3 1 2 0 8 4 5
pass
3 1 2 0 8 4 5
pass
3 1 2 0 8 4 5
44 7 0 1 14 17 52
YES
```

编译指令

编译指令为：`g++ main.cpp Vector.cpp Node.cpp -o main -std=c++11 -fno-elide-constructors`

提交要求

在已有代码的基础上编写`Vector.cpp`，**不能使用STL标准库中的vector容器。**

D. 机器人组装

题目描述

有两种机器人 `Alice` 和 `Bob`，它们的制造分别需要若干零件 `Part`（对于同一种类，每个机器人需要的零件数量也可能不一样）。零件 `Part` 存储一个整型的数字，`Alice` 在运行的过程中会计算它所有 `Part` 零件的数字之和，`Bob` 在运行过程中会计算它所有 `Part` 零件数字的平方和。本题需要你编写几个类，来模拟这两种机器人的制造和运行过程。测试代码可从这里下载。

要求如下：

- 分别用一个类实现 `Alice` 和 `Bob`，它们继承自基类 `Robot`。
- 用一个类实现零件 `Part`。
- 使用动态内存分配实现由零件 `Part` 组合成 `Alice` 与 `Bob`。
- 除了必要的构造函数和析构函数之外，机器人相关的类需要如下成员函数（**省略了传入的参数**）：
  - `bool is_full()`：机器人制造完成返回 `True`，否则返回 `False`
  - `void add_part()`：为机器人添加零件
  - `int run()`：运行机器人
- 对机器人类重载标准输出流运算符，使用该运算符时输出 `Build robot Alice` 或 `Build robot Bob`。具体调用方式见评测代码。

输入格式

第一行一个正整数 `ma`，表示要制造几个 `Alice`。

第二行 `ma` 个正整数，表示每个 `Alice` 分别需要几个零件。

第三行一个正整数 `mb`，表示要制造几个 `Bob`。

第四行 `mb` 个正整数，表示每个 `Bob` 分别需要几个零件。

第五行一个正整数 `m`，表示提供的零件是什么。

接下来的 `m` 行每行两个正整数，第一个正整数为 `0` 时代表零件被分配于制造机器人 `Alice`，为 `1` 时代表零件被分配于制造机器人 `Bob`，第二个正整数为零件存储的整型数字。

输出格式

输出若干行，随着零件的提供按照输入的顺序制造并运行机器人。如果一个机器人制造完成，则马上运行机器人，并输出相关信息和运行结果，具体信息内容和格式见测试代码和样例。

输入样例

```
3
5 4 5
2
2 7
16
1 2
0 5
0 2
1 0
1 8
1 2
0 3
1 3
0 5
0 2
0 6
0 1
0 10
0 5
0 6
0 7
```

输出样例

```
Build robot Bob run: 4
Build robot Alice run: 17
Build robot Alice run: 22
```

数据规模

$1 \leq ma, mb \leq 100$

$1 \leq m \leq 1000$

保证提供的零件数 $\leq$ 制造所有机器人所需的零件数

保证最终输出数字在32位整数范围内

提交格式

- 你需要提交多个文件，包含Makefile，上述文件调用的各种头文件及其cpp文件；可以不包括提供的main.cpp文件。使用Makefile必须要能生成可执行文件main（不带扩展名）。
- 你应该将你的文件打包成一个zip压缩包并上传。注意：你的文件应该在压缩包的根目录下，而不是压缩包的一个子文件夹下。评测时，OJ会将提供的main.cpp贴入你的目录下进行编译并执行。

评分标准

OJ评分占70%，人工评分占30%。

人工评价考查以下内容： 1. 类的组合继承合理性 2. 程序的可拓展性 3. 正确的多文件实现方式 4. 正确的权限使用方式 5. 不会出现内存泄露或其他逻辑错误

第四次作业

A. 选择

1. 关于以下代码，说法正确的有：

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

class Base {
    int x;
public:
    Base(){}
    Base(const Base & other){x = other.x;}
    void f(){ cout << "Base::f()" << endl;}
};

class Derive: public Base {
    int y;
public:
    Derive(){}
    Derive(const Derive & other):Base(other){y = other.y;}
    void f(){ cout << "Derive::f()" << endl;}
};

void g1(Base & obj){
    obj.f();
}

void g2(Base obj){
    obj.f();
}

Base g3(Derive obj){
    return obj;
}

int main(){
    Derive a;
    g1(a); // (1)
    g2(a); // (2)
    Derive b(a);
    Base c = g3(a); // (3)
    return 0;
}
```

A. 一共发生了3次向上类型转换，分别发生在 `(1)`，`(2)`，和 `(3)`，其中，`(3)` 的函数调用 `g3(a)` 在返回的过程中把 `Derive` 类的对象 `obj` 对象切片成 `Base` 对象

B. 一共发生了1次对象切片,发生在 `(2)`

C. `(1)` 处 `g1(a);` 调用的是 `Derive::f()`

D. 如果把 `Derive` 公有继承 `Base` 改成私有继承 `Base`，`(1)`，`(2)` 处涉及私有继承的向上类型转换，将发生编译错误

2. 关于以下代码，说法正确的有：

```
class Base {
    int x;
public:
    virtual void f1(){}
    virtual void f2() final {}
    void g(int){}
    void g(float){}
    void g(){}
};

class Derive: public Base {
    int y;
public:
    void f1(){}
    void g(){}
};
```

A. `Derive::g()` 是 `Base::g(int)` 和 `Base::g(float)` 的重载函数

B. `Derive::g()` 重写隐藏 `Base::g(int)` 和 `Base::g(float)`

C. `Derive::f1()` 重写覆盖 `Base::f1()`； `Derive::g()` 重写覆盖 `Base::g()`

D. `Base::f2()` 由于被 `final` 关键字修饰而不能被重写覆盖，但 `Derive` 类中仍然可以定义成员函数 `void f2()`，并重写隐藏 `Base::f2()`

3. 关于以下代码，说法正确的有：

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

class Base {
    int* x;
public:
    Base(){x = new int[10];}
    Base(const Base& other){fn();} // (1)
    virtual void fn(){}
    virtual void g1(){}
    virtual Base& g2(){}
    ~Base(){delete [] x;}
};

class Derive: public Base {
    int* y;
public:
    Derive(){y = new int[10];}
    Derive(const Derive& other):Base(other){fn();}
    void fn(){}
    void g1() const {}
    Derive& g2(){}
    ~Derive(){delete [] y;}
};

void fn(){
    Base* b = new Derive();
    delete b; // (*)
}

int main(){
    fn();
    return 0;
}
```

A. `(1)` 处调用的是 `Base::fn()`，这是因为虚机制在**拷贝构造函数**中不起作用

B. `Derive::g1()` 被const修饰， `Base::g1()` 没有被const修饰。 `Derive::g1()` 仍能重写覆盖 `Base::g1()`，不会出现编译错误。

C. `Derive::g2()` 的返回值为Derive&， `Base::g2()` 的返回值为Base&。 `Derive::g2()` 仍能重写覆盖 `Base::g2()`，不会出现编译错误。

D. 由于 `Base` 类析构函数不是虚函数， `(*)` 处析构 `b` 指向的 `Derive` 类对象时不会调用 `Derive` 类的析构函数，故存在内存泄漏的问题

4. 关于抽象类，以下说法正确的有：

A. 抽象类必定存在一个成员函数没有函数体

B. 如果在一个抽象类的派生类中将该抽象基类的所有纯虚函数都重写覆盖，则该派生类不再是抽象类

C. 当尝试将派生类对象向上转换为抽象类对象时，会产生编译错误，这是因为抽象类不允许定义对象

D. 抽象类不允许有数据成员

5. 关于类型转换，以下说法正确的有：

A. `dynamic_cast` 只能实现引用或指针的向下类型转换，不能实现向上转换。

B. 若有 `Derive* a = dynamic_cast<Derive*>(pt)` 类型转换成功（即a指针不为空），其中 `pt` 类型为 `Base*`， `Derive` 类是 `Base` 类的派生类，则 `Base` 类一定有虚函数

C. 若有 `a` 是 `Base` 类的对象，且 `Derive` 类是 `Base` 类的派生类，则类型转换 `static_cast<Derive>(a)` 一定可以成功，运行不会出现错误

D. 对指针和引用进行向下类型转换时，使用 `dynamic_cast` 比 `static_cast` 更安全

6. 关于虚机制，以下说法正确的有：

A. 一个派生类对象可以继承多个抽象基类

B. 现有 `Derive` 类是 `Base` 类的派生类，且 `b` 是指向 `Derive` 类对象的 `Base` 类指针，即使派生类 `Derive` 重写覆盖了基类 `Base` 的一个虚函数 `void fn();`，仍然可以通过 `b->Base::fn()` 调用 `Base::fn()`

C. 在类的析构函数中调用虚成员函数，被调用的是该函数的本地版本

D. 如果派生类保护继承或私有继承基类，则基类的虚成员函数将不能被重写覆盖

7. 关于多态，以下说法正确的有：

A. 对象、指针、引用对虚函数的调用总是晚绑定，即运行时绑定

B. 多态可以提高代码接口的复用性

C. C++语言编译时的多态性可通过函数重载或模板实现

D. 若派生类 `Derive` 重写覆盖基类 `Base` 的一个虚函数 `void fn();`，且 `Derive` 类对象 `d` 通过对象切片的方式转换成 `Base` 类对象 `b`，因为虚函数的多态性， `b.fn()` 将调用 `Derive::fn()`

8. 关于以下代码，说法正确的有：

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

// (1)
template<typename T0> class A {
    T0 value;
public:
    void set(const T0 & v){
        value = v;
    }
    T0 get();
};

// (2)
template<typename T1>
T1 A<T1>::get(){ return value;}

template<typename T0, typename T1>
T1 sum(T0 a, T0 b){
    return T1(a.get() + b.get());
}

int main() {
    A<double> a;
    a.set(4.3);
    cout << a.get() << endl;
    cout << sum<A<double>, int>(a, a) << endl;
    // (3)
   return 0;
}
```

A. `(2)` 处 `A::get()` 的定义采用的类型名 `T1` 与 `(1)` 处类 `A` 采用的类型名 `T0` 不同，因此编译错误

B. 程序的执行结果为 `4.3\n8\n`

C. **根据函数模板 `sum` 的定义**，实参类型需要定义成员函数 `get`，且 `sum` 的返回值类型必定与参数类型不同

D. 如果 `(3)` 处添加代码 `sum(9, 2);`，程序依然能正常编译运行

```
D
B
ACD
BC
BD
ABC
BC
B
```

B. 线性数据结构

题目描述

栈和队列是两种应用广泛的数据结构。它们都属于线性数据结构，并且都具有插入、删除两种基本操作。不同的是，栈采用“后进先出”的策略，即先被插入栈中的元素会后被从栈中删除；而队列采用“先进先出”的策略，即先被插入队列的元素也会先被从队列中删除。

C++本身对于这些数据结构就有一定的支持。同时，为了更方便使用，C++还提供了“迭代器”（`Iterator`）的概念。简单来说，“迭代器”（`Iterator`）类似于指向数据结构内部元素的指针，使得用户可以在以上提到的两种操作之外，方便地访问数据结构中的每个元素。但是，迭代器比裸的指针更加安全。

这里我们给出一个迭代器的例子，假设某数据结构包含 `1, 2, 3, 4, 5` 五个数，迭代器可以认为是指向该数据结构元素的指针。参考以下代码体会它的功能：

```
Iterator it = s.begin(); //迭代器it1指向第一个数
cout << *it << endl; // 1
cout << (it!=s.begin()? "yes":"no") << endl; //两个迭代器指向同一位置，输出no
it++; //迭代器it指向第二个数
cout << *it << endl; //2
cout << (it!=s.begin()? "yes":"no") << endl; //两个迭代器指向不一样的位置，输出yes
```

同时，数据结构应该支持 `begin()` 和 `end()` 两个函数，分别返回开始和结束位置的迭代器。值得注意的是，`end()` 应指向最后一个元素的下一个位置，而不是最后一个元素。用数学语言来说，`begin()` 和 `end()` 是一个左闭右开的区间。具体可以参考以下代码：

```
for (Iterator it = s.begin(); it != s.end(); it++) {
    std::cout << *it << " "; //1 2 3 4 5
}
```

最后一步循环里 `it` 指向5，`it++` 后，`it == s.end()`，退出循环。

C++中的“迭代器”（`Iterator`）由模板实现，但在本题中，希望你能利用C++继承的多态和运算符重载的特性实现栈和队列，以及访问数据结构中每个元素的迭代器（`Iterator`）。

要求

1. 实现 `Stack` 和 `Queue` 类，它们都继承自一个**抽象基类** `LinearDataStruct`。为简单起见，两个数据结构中只存储 `int` 类型。
2. 为每个数据结构实现 `Iterator` 类，都继承自 `IteratorBase` 类，可以按只读的方式访问 `Stack` 和 `Queue` 中的元素。
3. 为数据结构实现以下接口，你需要确定哪些成员函数可以在基类中实现，哪些必须在子类中实现，以最大可能地复用代码。

   ```
   void push(int n); // 插入元素
   int pop(); // 删除元素并返回被删除的元素：对于栈，会删除最后插入栈的元素；对于队列，将删除最先插入队列的元素
   int max_size(); // 数据结构的最大容量
   int size(); // 数据结构的当前容量

   Iterator begin(); // 返回指向数据结构开始元素的迭代器
   Iterator end(); // 返回指向数据结构结束元素的下一个位置的迭代器
   ```

4. 为 `Iterator` 类重载下列运算符，同样地，你需要根据你的实现确定哪些成员函数可以在基类中实现，哪些必须在子类中实现：
   - `!=`：两个 `Iterator` （逻辑上）指向同一个位置则相等，否则不等
   - `*`：类似于指针的取值运算符
   - `++`：两种形式的自增运算符，假设数据结构中数据按照进入的时间排序。

   具体来说，你编写的迭代器可以通过以下代码输出整个数据结构：

   ```
   //对于Stack的指针S
   for (Stack::Iterator it = S->begin(); it != S->end(); it++) {
       std::cout << *it << " ";
   }
   ```

5. 和已经写好的文件一起编译、运行输出正确结果。**注意：可以在提供代码的基础上增加代码文件，但提供的代码中 `TODO` 以外的部分不允许修改，不允许使用STL。如果系统检测到你修改了其他位置的代码或使用了STL，会显示例如 `Wrong Answer at IteratorBase.h` 的信息。**

输入说明

第一行一个正整数 `m`，代表要构造的数据结构个数。

接下来 `m` 行，每行两个整数 `t s`，当 `t = 0` 时，构造 `Stack`，`t = 1` 时，构造 `Queue`。`s` 为该数据结构的最大容量。

第 `m+1` 行一个正整数 `k`，代表操作个数。

接下来 `k` 行，每行若干整数。

首先读入第一个整数 `o`。

如果 `o = 0`，代表插入操作，则该行还会有两个整数 `i e`，`i` 为需要进行操作的数据结构的编号索引，`e` 为插入的整数。

如果 `o` 为 `1~4`，则分别代表"删除元素并输出该元素"，"输出最大容量"，"输出当前容量"，"遍历容器输出"这四个操作。这时该行还会有一个整数 `i`，代表需要进行操作的数据结构的编号索引。

输入样例

```
5
0 5
1 5
0 3
0 4
1 2
19
0 1 10
0 1 2
0 2 3
0 3 8
0 0 9
0 0 8
0 0 7
0 0 6
0 0 5
0 0 4
0 0 3
1 0
2 0
3 0
4 0
2 1
3 1
1 1
4 1
```

输出样例

```
Full!
Full!
5
5
4
9 8 7 6
5
2
10
2
```

数据规模

$2 \leq m \leq 10$

$2 \leq s \leq 1000$

$1 \leq k \leq 10000$

$|e| \leq 100000$

保证 $o \in \{1,2,3,4,5\}, 0 < i < m$

提示

对于队列的实现，可以考虑采用循环队列的实现方式以增加程序效率。该要求并不强制，你也可以采取其他的实现方式。

在这题的实现中，我们使用了类的嵌套，即可以在Stack类中定义Iterator类，访问Iterator类时应使用 `Stack::Iterator`，其他和普通类并无区别。更多相关可以查看嵌套类和局部类相关资料。

提示：你不能修改除TODO以外的位置。

提交格式

- 你需要提交多个文件，包含Makefile，上述文件调用的各种头文件及其cpp文件，可以不包括main.cpp。使用Makefile必须要能生成可执行文件main（不带扩展名）。
- 你应该将你的文件打包成一个zip压缩包并上传。注意：你的文件应该在压缩包的根目录下，而不是压缩包的一个子文件夹下。评测时，OJ会将提供的main.cpp贴入你的目录下进行编译并执行。

C. 树洞管理

题目描述

树洞是一个校内同学互相匿名交流的公共平台，作为树洞的管理员，你管理了若干个帖子。现在你要写一个程序，（1）支持封禁帖子的操作，即 `ban x y z`，代表第 `x` 个帖子的第 `y` 个回复将不会被展现，该操作由 `id` 为 `z` 的管理员提交；（2）查询封禁情况的操作，即 `query x y`，返回第 `x` 个帖子的第 `y` 个回复的封禁状况。管理员的 `id` 可能为 `string` 类型，也可能是 `int` 类型，程序需要支持这两种类型。

我们已经实现了测试文件 `main.cpp`，你需要实现相应代码满足测试文件的需求。测试文件下载地址：下载链接

输入样例

- 第一行是管理员的 `id` 类型，为 `int` 或 `string`；第二行是一个整数 `n`，表示操作的总数。接下来 `n` 行，每行描述一个操作。首先是一个字符串，有两种可能：（1）`ban` 代表一个封禁操作；（2）`query`，代表查询操作。字符串之后是帖子的序号和回复的序号。

```
string
8
ban 5 3 Wang
ban 3 4 Huang
ban 6 2 Musk
ban 4 1 Shi
query 3 2
query 5 3
query 4 1
query 6 1
```

输出样例

```
open
Wang
Shi
open
```

- 对每个查询操作，输出一行为查询的答案，若帖子未被封禁，返回 `open`；反之，直接返回执行该操作的管理员的 `id`。

限制与约定

- $0 < n \leq 1000$
- 帖子的序号和回复的序号均为整数，在 `int` 类型范围内，均不超过100000。所有帖子的回复的初始状态均为未封禁。管理员 `id` 可能为 `int` 或是 `string`，字符串的长度不超过 25。

要求

- 我们提供了测试文件main.cpp。
- 你需要提交多个文件，包含Makefile，上述文件调用的各种头文件及其cpp文件，可以不包含 `main.cpp`。使用Makefile必须要能生成可执行文件main（不带扩展名）。
- 你应该将你的文件打包成一个zip压缩包并上传。注意：你的文件应该在压缩包的根目录下，而不是压缩包的一个子文件夹下。评测时，OJ会将提供的main.cpp贴入你的目录下进行编译并执行。

D. 简单计算图2 //隐藏

第五次作业

A. 选择

1. 【单选】下列关于命名空间和STL的说法正确的是:

   A. 如下的代码片段是有问题的，即不可以声明和定义同一个命名空间多次，每次添加新的成员。

```
namespace A{int i;}
namespace A{int j;}
```

   B. 使用 `using std::cout` 会暴露命名空间 `std` 中其他的变量名例如 `cin`;

   C. 命名空间可以嵌套, 例如 `namespace A{namespace B{int i;}}`。

   D. `STL` 中的**序列**容器包括 `vector`, `list`, `set`。

2. 【单选】下列关于简单容器类型及其函数说法**错误**的是:

   A. 从 `std::make_tuple` 可以接受任意个数的参数，可以看出 `tuple` 对象的长度（即包含的元素个数）在运行时才确定。

   B. `std::make_pair(a,b)` 中，`a`,`b` 的变量类型可以不相同。

   C. `std::tuple` 类重载了赋值运算符 `=`。

   D. `std::tie` 的形参和返回值均为引用。

3. 【单选】以下代码展示了一些vector类型上的操作, 说法【错误】的是:

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main(){
6       vector<int> vec = {1,2,3,4,5};
7       auto a = vec.end();
8       auto b = vec.begin();
9       vec.erase(b+1);
10      auto c = vec.begin() + 2;
11      cout<<*c;
12      vec.insert(vec.begin(),6);
13      vec.insert(vec.end(),6);
14      return 0;
15  }
```

   A. 运行到第9行之前，`*a` 的值是5。

   B. 第11行的输出是4。

   C. `vector` 类型本身没有重载流运算符，需要自己重载 `<<` 才能使用 `cout<<vec;`。

   D. 由于在 `vec` 插入数据需要把插入位置及之后的元素后移，第12行比第13行执行更多的元素后移操作。

4. 【单选】下列关于STL容器说法正确的是:

   A. 使用 `list` 插入删除数据，一定不会使任何的迭代器失效，包括指向被插入或被删除的元素的迭代器。

   B. `list` 的 `push_back` 可能会发生整体的内存拷贝。

   C. 对 `map` 用 `[]` 进行下标访问如果元素不存在则返回 `nullptr`。

   D. 下面的代码输出是 `312`。

```cpp
#include <iostream>
#include <map>

int main ()
{
    std::map<char,int,std::greater<char>> mymap;
    mymap['b'] = 1;
    mymap['a'] = 2;
    mymap['c'] = 3;
    for(auto it=mymap.begin(); it!=mymap.end(); ++it)
        std::cout << it->second;
    return 0;
}
```

5. 【多选】若在 `vector`，`list`，`set`，`map` 中选择，下列关于说法合适的有:

A. 小明想构建一个词表，仅记录一篇文章中所有用到的词，最好选用 `set`。

B. 小兰想建立一个班级通讯录，需要通过同学姓名检索TA的相关信息，最好选用 `map`。

C. 小红想跟踪编号为 `1-100` 的被试的动态收支情况(用当前资产表示)，需要修改和查询某一编号被试的信息，最好选用 `list`。

D. 小新想维护一个排队购票系统，记录每个排队者的信息，需要从队尾加入新的排队者或从队头删去排队者信息，最好选用 `vector`。

6. 【多选】下列关于C++中迭代器的说法，正确的有:

A. STL中 `vector` 类的迭代器支持迭代器之间的减法。

B. 调用 `push_back` 等修改 `vector` 大小的方法时，原有迭代器肯定都不会失效。

C. 迭代器指向的内容删除后，该迭代器失效。

D. 可以通过自定义迭代器，实现用同样操作符（例如自增操作符）执行不一样的元素访问顺序。

7. 【单选】关于下面的bash代码(linux环境下保存为 `main.sh`)说法正确的是:

```bash
#!/bin/bash
filename=test
for par in $@;
do
   mkdir output-$par && python ${filename}.py $par
done
```

A. 为了美观可以在 `filename=test` 的等号边加上空格，即修改为 `filename = test`，bash脚本仍能运行、功能不变。

B. 如果该目录下已有 `output-1` 子目录，再使用 `./main.sh 1` 执行该脚本，将不会运行 `python` 语句。

C. 如果该bash脚本有两个参数, 则 `$@` 等价于 `$0 $1 $2`。

D. 如果执行bash脚本的命令为 `./main.sh 0.5 1.0`，则 `$par` 是浮点类型。

8. 【单选】下列关于git的说法正确的是:

A. 如果不小心把项目中文件 `hello.cpp` 的内容破坏了, 使用 `git reset - hello.cpp` 恢复到上一次提交的版本。

B. `git branch test` 会创建名为 `test` 的分支并且切换到 `test` 中

C. 如果在 `.gitignore` 文件写入一行 `*.xx`, 这可以保证只提交以 `.xx` 为后缀的文件。

D. 与他人使用远程仓库合作时, 使用 `git fetch` 会比较远程仓库中的内容和本地仓库的内容，用户在检查是否有冲突以后决定是否合并到本地仓库中；而 `git pull` 会直接尝试合并。

```
C
A
A
D
AB
ACD
B
D
```

B. 消息列表维护

题目描述

日常生活中，我们常常会使用线上聊天工具与好朋友们交流。现要求你维护一个简易的聊天列表，需要满足以下需求：

- 每个好友有一个id（整数，不超过10000），该id是唯一的。
- 当用户接收到消息时，如果消息列表里没有该好友，则将该好友加到消息列表，将消息显示到该好友的聊天窗口，并将该好友置顶；如果该好友已经存在于消息列表中，则将消息显示到该好友的消息记录的最上方，同时将该好友置顶。
- 输出最终的消息列表。

输入

第一行为整数N，表示初始消息列表中好友的数量。接下来N行，每行包括两项内容，第一项为用户id，第二项是一系列聊天内容（聊天内容中不包含空格），其中每句按照斜杠（/）分隔。**其中，用户的顺序为消息从旧到新的顺序，即输入越靠后的用户，在消息列表中显示越靠前。聊天内容中，消息的顺序也为从旧到新的顺序，输入越后的，为新聊天记录，在显示时也越靠前。**

接下来为一个整数M，表示按时间顺序的新接收的消息的数量。接下来M行，每行第一个为用户id，之后跟一句话作为聊天内容（聊天内容中不包含空格）。

输出

输出最终消息列表。首先按好友从新到旧进行输出，每个好友的聊天记录同样也按从新到旧输出。输出格式为:

```
用户id（最新的聊天好友）
该用户的消息1（最新）
该用户的消息2
...
该用户的消息n（最旧）
用户id（次新的聊天好友）
该用户的消息1（最新）
该用户的消息2
...
该用户的消息n（最旧）
...
用户id（最旧的聊天好友）
该用户的消息1（最新）
该用户的消息2
...
该用户的消息n（最旧）
```

样例输入：

```
2
15 Today_is_Thursday.
47 Hi/I_love_OOP.
2
15 I_am_sad.
13 Glad_to_see_you.
```

样例输出：

```
13
Glad_to_see_you.
15
I_am_sad.
Today_is_Thursday.
47
I_love_OOP.
Hi
```

要求

- 你需要提交多个文件，包含Makefile，上述文件调用的各种头文件及其cpp文件。使用Makefile必须要能生成可执行文件main（不带扩展名）。
- 你应该将你的文件打包成一个zip压缩包并上传。注意：你的文件应该在压缩包的根目录下，而不是压缩包的一个子文件夹下。

C. 模拟外星人

你需要使用C++继承和多态的特性来模拟外星人。有以下一段代码（`main.cpp`）：

```
void doSpeak(WhatCanSpeak* obj) {
  obj->speak();
  obj->stop();
}
void doMotion(WhatCanMotion* obj) {
  obj->motion();
  obj->stop();
}

int main()
{
  std::string name;
  std::cin >> name;
  Alien alien(name);
  doSpeak(&alien);
  doMotion(&alien);
  return 0;
}
```

其中`WhatCanSpeak`和`WhatCanMotion`类都已经写好：

```
class WhatCanSpeak {
public:
  virtual ~WhatCanSpeak() {std::cout << "obj cannot speak" << std::endl; }
  virtual void speak() = 0;
  virtual void stop() = 0;
};
class WhatCanMotion {
public:
  virtual ~WhatCanMotion() {std::cout << "obj cannot move" << std::endl; }
  virtual void motion() = 0;
  virtual void stop() = 0;
};
```

你需要补充头文件`alien.h`，使得程序能够编译通过。

文件下载地址：下载链接

输入说明

输入是一个字符串，是alien的名字。

输出说明

输出一共六行，请观察样例。

输入样例

```
alice
```

输出样例

```
alice is speaking
alice stops
alice is moving
alice stops
obj cannot speak
obj cannot move
```

提交格式

你只能提交头文件`alien.h`我们会将你提交的文件和我们预先设置好的`main.cpp`，`what.h`一起编译运行。

D. 运送动物

题目描述

动物园新接收了一批动物。由于接收过程比较匆忙，整个过程分为了两步：

- 园方依次接收所有的动物（`Animal`），先暂存在一块区域（`buffer`）。
- 之后，园方再按照动物种类，包括狗（`Dog`）和鸟（`Bird`），按照接收的顺序将动物**移动**到相应的区域。

整个过程由以下代码（`main.cpp`）描述。

```
int main()
{
    Animal* buffer[100];
    vector<Dog> dogzone;
    vector<Bird> birdzone;

    int n;
    cin >> n;

    // step 1: store all animals in buffer
    for(int i = 1; i <= n; i++){
        string type, name;
        cin >> type >> name;
        if(type == "dog")
            buffer[i] = new Dog(name);
        else if(type == "bird")
            buffer[i] = new Bird(name);
    }

    // step2: move animal to the corresponding zone
    for(int i = 1; i <= n; i++){
        action(buffer[i], dogzone, birdzone);
        delete buffer[i];
    }

    // output animals' name and type
    for(auto &dog : dogzone){
        cout << dog << endl;
    }
    for(auto &bird : birdzone){
        cout << bird << endl;
    }

    return 0;
}
```

其中你需要补充头文件`action.h`，在其中实现

```
void action(Animal* animal, std::vector<Dog> & dogzone, std::vector<Bird> & birdzone)
```

该函数将判断当前animal的类型，并将该动物移动到相应的区域中（利用`dogzone.push_back`或`birdzone.push_back`）。

`Animal`类、`Dog`类和`Bird`类都已经写好：

```cpp
class Animal {
private:
    std::string name, type;
public:
    Animal(const std::string &_name, const std::string &_type): name(_name), type(_type) {}

    //Cannot copy animals
    Animal(const Animal &other) = delete;
    //Can move animals
    Animal(Animal &&other): name(other.name), type(other.type) {
        other.name = ""; other.type = "empty";
    }

    //Cannot copy animals
    Animal& operator=(const Animal &other) = delete;
    //Can move animals
    Animal& operator=(Animal &&other){
        if(type != "empty"){
            std::cout << "There has already been an animal here. Cannot move." << std::endl;
        }
        name = other.name; type = other.type;
        other.name = ""; other.type = "empty";
        return *this;
    }

    friend std::ostream& operator<<(std::ostream& out, const Animal& animal){
        out << animal.type << " " << animal.name;
        return out;
    }

    //pure virtual destruction
    virtual ~Animal()=0;
};

inline Animal::~Animal()
{
    if(type != "empty"){
        std::cout << "An animal deleted." << std::endl;
    }
}

class Dog: public Animal {
public:
    Dog(const std::string &_name): Animal(_name, "dog") {};
};

class Bird: public Animal {
public:
    Bird(const std::string &_name): Animal(_name, "bird") {};
};
```

提示：

- `Animal` 中 `type` 变量为 `private`，无法从外部访问。
- `dynamic_cast`。
- 动物只能**移动**，不能**复制**。

文件下载地址：下载链接

输入说明

输入第一行是n（$n \leq 100$），代表接受的动物数量。

接下来n行，每行两个字符串，分别是动物的种类和动物的名字。

输出说明

输出一共$2n$行。

前n行，每行两个字符串，分别经过分类后动物的种类和动物的名字。

后n行为Animal类的析构信息。

输入样例

```
4
bird Jack
dog DogKing
bird Jone
dog BigDog
```

输出样例

```
dog DogKing
dog BigDog
bird Jack
bird Jone
An animal deleted.
An animal deleted.
An animal deleted.
An animal deleted.
```

提交格式

你只能提交头文件 `action.h`，我们会将你提交的文件和我们预先设置好的 `main.cpp`，`animal.h` 一起编译运行。

评分标准

OJ评分占100%。

第六次作业

A. 选择

1. 【单选】以下哪份代码可以编译通过，且运行时一定能够完成对arr数组从大到小的正确排序:

A.

```
#include <algorithm>
bool comp(int a, int b) {return a > b;}
int main(){
    double arr[5] = { 5.0, 2.0, 2.2, 1.0, 7.0 };
    std::sort(arr, arr + 5, comp);
    return 0;
}
```

B.

```
#include <algorithm>
#include <functional>
int main(){
    double arr[5] = { 5.0, 2.0, 2.2, 1.0, 7.0 };
    std::sort(arr, arr + 5, std::less<double>);
    return 0;
}
```

C.

```
#include <algorithm>
class Greater {
public:
    bool operator()(T &a, T &b) const    {
        return a > b;
    }
};
int main(){
    double arr[5] = { 5.0, 2.0, 2.2, 1.0, 7.0 };
    std::sort(arr, arr + 5, Greater<double>());
    return 0;
}
```

D.

```
#include <algorithm>
int comp(double a, double b) {return a > b;}
int main(){
    double arr[5] = { 5.0, 2.0, 2.2, 1.0, 7.0 };
    std::sort(arr, arr + 5, comp);
    return 0;
}
```

2. 【单选】以下说法正确的是:

A. 一个裸指针可以初始化多个智能指针，这几个智能指针将共享同一个辅助指针

B. 不能将多个智能指针指向同一个对象

C. 使用类对象的指针 `ptr` 构造 `shared_ptr` 后，则 `delete ptr;` 是合法操作，不会造成潜在的问题

D. 不能使用智能指针维护对象数组

3. 【单选】以下说法正确的是:

A. 函数对象不能定义析构函数

B. 可以使用 `unique_ptr` 对象作为初始化参数构造 `shared_ptr` 对象

C. 只要重载了 `operator()` 运算符，无论其访问该运算符是 `public` 还是 `private`，该类对象都可以认为是函数对象

D. `unique_ptr` 指针不能复制，只能移动

4. 【单选】下列说法正确的是（为简便省略了头文件，注意局部变量初始化不保证为0）：

A. 执行下列代码片段后，`strlen(a)` 将为 10

```
void func() {
  char a[20];
  int i = 0;
  while (strlen(a) < 10) {
      a[i] = char(i + int('a'));
      i++;
  }
  std::cout << a << std::endl;
}
```

B. 下列代码片段可以修改字符串 `a` 中的字符，不存在产生编译错误

```
void func() {
  std::string a = "abcdefg";
  char* b = a.c_str();
  for (int i = 0; i < a.size(); ++i) {
      b[i] = char(i + 'A');
  }
  std::cout << a << std::endl;
}
```

C. 下列代码片段可以将字符串 `b` 拷贝至 `a`，最终 `strlen(a)` 等于 `strlen(b)`

```
void func() {
  char a[20];
  const char* b = "abcdefg";
  int len = strlen(b);
  for (int i = 0; i < len; ++i) {
      a[i] = b[i];
  }
  std::cout << a << std::endl;
}
```

D. 下列代码片段可以正常修改字符串 `a` 中的字符，不产生编译错误

```
void func() {
  std::string a = "abcdefg";
  for (int i = 0; i < a.size(); ++i) {
      a.at(i) = char(i + 'A');
  }
  std::cout << a << std::endl;
}
```

5. 【多选】关于下列函数，说法正确的有（为简便省略了头文件）：

A. 以下函数的功能是将 `target` 的前缀修改为 `prefix`，但是算法复杂度可以进一步降低

```
void replace_prefix(const char * prefix, char * target){
    for (int i = 0; i < min(strlen(prefix), strlen(target)); ++i){
        target[i] = prefix[i];
    }
}
```

B. 以下函数的功能是将字符串数组 `suffixes` 中的字符串依次拼接到 `target` 后面，但是在不改变函数签名的情况下，可以通过优化内部实现，减少对象创建和拷贝等操作

```
void append_suffixes(string & target, vector<string> & suffixes){
    for(auto & suffix: suffixes){
        target = target + suffix;
    }
}
```

C. 以下函数的功能是将字符串数组 `suffixes` 中的字符串依次拼接到 `target` 后面，其中 `target += suffix;` 相比 `target = target + suffix;` 更加高效

```
void append_suffixes(string & target, vector<string> & suffixes){
    for(auto & suffix: suffixes){
        target += suffix;
    }
}
```

D. 以下函数的功能是将字符串数组 `suffixes` 中的字符串依次拼接到 `target` 后面，其中 `target.append(suffix)` 相比 `target = target + suffix;` 更加高效

```
void append_suffixes(string & target, vector<string> & suffixes){
    for(auto & suffix: suffixes){
        target.append(suffix);
    }
}
```

6. 【单选】对比 `std::vector<char> a` 和 `std::string a`，下列哪种操作 `std::vector` 和 `std::string` 共同具有：

   A. `std::cin >> a`

   B. `a.append("abc")`

   C. `std::sort(a.begin(), a.end())`

   D. `a.length()`

   注意：第7、8题涉及第14课的内容，相关课程的讲授安排在本次作业结束前一周

7. 【多选】以下关于迭代器模式的说法正确的是：

   A. 迭代器模式是为了提供一种方法顺序访问一个聚合对象中各个元素，而又无须暴露该对象的内部表示。

   B. 迭代器类通常设计为存储类的友元，从而迭代器类可以访问存储类内部的数据。

   C. 存储器类需要是迭代器类的友元。

   D. 迭代器模式实现了算法和数据存储的隔离。

8. 【单选】以下关于模板方法和策略模式的说法不正确的是：

   A. 在设计思路上，模板方法模式优先考虑继承，策略模式优先考虑组合。

   B. 策略模式在策略组合时封装性较差，较难处理业务场景十分复杂的情况。

   C. 当算法的每一个步骤都有多种解决方案时，模板方法优势较大。

   D. 使用接口类而不是具体类的实现进行编程有助于避免具体类的改变造成整个程序的大规模变化。

```
D
D
D
D
ABCD
C
ABD
C
```

B. 容器的可查询化 II

题目描述

请在头文件myqueriable.h中，实现一个序列容器的模板类MyQueriable。它需要支持

- 使用 `obj.size()` 得到容器的长度
- 使用 `obj[i]` 访问第i个元素

同时需要支持where、apply、sum三个成员函数。其中：

- where接受一个函数对象f作为参数，返回值为 `{x|f(x)==true, x属于原来的MyQueriable}表示的MyQueriable（元素顺序与原来的MyQueriable一致）`；
- apply以一个函数对象f作为参数，返回值为 `{f(x)|x属于原来的MyQueriable}表示的MyQueriable（元素顺序与原来的MyQueriable一致）`；
- sum不带参数，返回值为MyQueriable中各元素之和。

此外，你还需要实现模板函数from

- 它以一个vector或list为参数,返回值为对应的MyQueriable

示例1：

```
#include<iostream>
#include<list>
#include"myqueriable.h"

using namespace std;

bool op1(int x){return x > 2 && x < 10;}
class Op2{
public:
    int operator()(int x) {return x * 2;}
}op2;

int main(){
    list<int> li = {1, 4, 6, 2, 10};
    auto out = from(li)
        .where(op1)  // 4, 6
        .apply(op2)  // 8, 12
        .sum();
    cout << out << endl;
}
```

示例1输出：

```
20
```

示例2：

```
#include<iostream>
#include<vector>
#include<cmath>
#include<functional>
#include"myqueriable.h"

using namespace std;

float fn(float x){return sqrt(x);}
function<float(float)> op = fn;

int main(){
    vector<float> vec = {2.0, 3.0, 4.0, 5.0};
    auto li = from(vec)
        .apply(op);
    for(int i = 0; i < li.size(); i++){
        cout << li[i] << endl;
    }
}
```

示例2输出：

```
1.41421
1.73205
2
2.23607
```

本题可以使用C++14。编译选项为 `g++ code.cpp -o code -lm -O2 -DONLINE_JUDGE --std=c++14`。网页上显示的可能是C++11，但实际上编译使用的C++14。

数据说明

1. 对于浮点数的测试用例，容许1e-5内的相对误差
2. 我们保证测试数据中不会出现MyQueriable对象中长度为0的情况
3. apply,sum返回值的元素类型应和原MyQueriable的元素类型相同：例如原来是int，sum后也该是int

分数设置

一共设置5个测试点，每个测试点占20%。

1. 只包含from(vector为参数)，sum。类型为int。
2. 只包含from(vector为参数)，需要size和下标访问。类型为int。
3. 只包含from(list为参数)，sum，需要size和下标访问。类型为int。
4. 包含所有方法，类型为int。
5. 包含所有方法，类型为int\float。

数据范围和样例类似，正常实现不需要考虑时限或内存限制。

C. 小明的动态类型

题目描述

小明最近了解了动态语言（python和javascript），这些语言中变量的类型是动态的，因此可以为同一个变量赋予不同类型的值。小明想在C++中也做到类似的事情，于是它设计了一个类 `Object`，任意的变量都可以储存进该类中。具体来说：

```
//subtask 1
Object x;
x = 1;
std::cout << x << std::endl; //1
x = std::string("text");
std::cout << x << std::endl; //text
x = CustomClass("obj1"); //a user-defined class
```

`Object` 一般都是以引用的形式储存数据，这种方式可以避免多余的复制操作。这也使得有时候对某一 `Object` 的修改导致另一 `Object` 的变化，举例来说：

```
//subtask 2
Object x, y, z;
x = 1;
y = x; // y point to the same value of x
z = CustomClass("obj2");
std::cout << x << " " << y << std::endl; //1 1

x += 1;
std::cout << x << " " << y << std::endl; //2 2

x = std::string("text"); // x points to "text", but y still points to 2
std::cout << x << " " << y << std::endl; //text 2

y = x;
x += std::string("1");
std::cout << x << " " << y << std::endl; //text1 text1
x = z;
```

注意这里 `=` 和 `+=` 的不同，`=` 是让 `Object` 改为储存另一个数据，`+=` 是修改储存的数据。

接下来，小明考虑让 `std::vector` 能够装进不同类型的对象。

```
//subtask 3
std::vector<Object> arr;
arr.push_back(Object(1));
arr.push_back(Object("text"));
arr.push_back(Object(CustomClass("obj3")));
std::cout << arr[0] << " " << arr[1] << std::endl; //1 text
arr[0] = arr[1];
std::cout << arr[0] << " " << arr[1] << std::endl; //text text
arr[1] += std::string("1");
std::cout << arr[0] << " " << arr[1] << std::endl; //text1 text1
arr[0] = arr[2];
std::cout << arr[1] << std::endl; //text1
```

如果将 `std::vector` 赋给 `Object`，甚至能够做到在 `std::vector` 中包含 `std::vector`。对于以下代码，如果存在理解困难，下载文件中提供了一个 `instruction.pdf` 解释了操作的过程。

```
//subtask 4
std::vector<Object> inner_arr1 = {Object(1), Object("text1"), CustomClass("obj4")};
std::vector<Object> inner_arr2 = {Object(2), Object("text2")};
std::vector<Object> arr = {Object(inner_arr1), Object(inner_arr2), CustomClass("obj5")};

std::cout << arr[0][0] << " " << arr[0][1] << std::endl; // 1 text1
std::cout << arr[1][0] << " " << arr[1][1] << std::endl; // 2 text2

arr[1] = arr[0];
std::cout << arr[1][0] << " " << arr[1][1] << std::endl; // 1 text1

arr[0][0] += 2;
std::cout << arr[0][0] << " " << arr[1][0] << std::endl; // 3 3

arr[0][1] = arr[1][0]; // arr[1][1] and arr[0][1] is the same Object
std::cout << arr[0][1] << " " << arr[1][1] << std::endl; // 3 3

arr[0][0] = arr[2];
```

- 为了简单考虑，我们对题目做出以下限制：
  - 只要求 `Object` 能够储存以下类型 `int`，`std::string`，`std::vector<Object>`，`CustomClass`。其中 `CustomClass` 是一个用户自定义类，该类会统计其对象构造与析构的次数，并测试你的代码是否存在内存泄露。
  - `+=` 运算右侧只会出现 `int` 和 `std::string`，并且保证类型与 `Object` 中存储的类型一致。
  - 中括号运算 `[]` 只会对储存了 `std::vector<Object>` 的 `Object` 使用，并且保证不会越界访问。

小明已经完成了部分代码，但是调试一直没法通过。你需要帮助他改好相应的功能（或者自己重新实现），并通过测试程序。`main.cpp` 为测试程序，`Object.h` 是小明实现的部分代码。文件下载地址：下载链接

如果觉得太困难，你可以选择完成一部分的功能以获取一部分的分数，请看评分标准。

对小明代码的解释

你可以采用小明的代码，**也可以完全不使用小明的代码**，他的代码只作为本题的提示。你只用提交 `Object.h`，通过测试即可。

需要注意的是，**小明的已完成的代码里包含一些错误和未完成的部分**，你需要进行修改。**提示：包含STL的程序往往编译错误较多，建议每次先解决第一个error，然后重新编译。**

这里对小明的设计做一些解释。小明首先发现，`Object` 储存的都是引用，而不应该是真实的值（因为两个 `Object` 可能指向同一个值，分开储存无法做到同时修改）。因此，小明新增了一个 `Content` 类，该类作为真正储存数据的位置。`Object` 储存了指向 `Content` 的指针。由于 `Content` 可以储存不同类型的数据，因此该类应该是一个基类，其派生类 `IntContent`，`StringContent`，`VectorContent`，`CustomContent` 分别储存 `int`，`std::string`，`std::vector<Object>`，`CustomClass` 数据。

该设计的示意图可以参考 `instruction.pdf`。小明在大部分存在问题的地方添加了注释，请仔细阅读注释中给予的**提示**完成代码。

提交格式

你只能提交头文件 `Object.h`

我们会将你提交的文件和我们预先设置好的 `main.cpp` 和 `CustomClass.h` 一起编译运行。

评分标准

我们会有1个样例测试点，即下发的 `main.cpp`。另外有1个隐藏测试点，会相应的更改 `main.cpp` 中内容进行测试。一般来说，如果正确实现了题目要求，并能通过样例测试点（而不是通过某种方法直接输出标准答案），也应该能够通过隐藏测试点。（若提交后存在问题，请及时联系助教。）

本题按照以上描述分为了4个子任务，每个子任务各占25分。只有你通过了同一个子任务的样例测试点和隐藏测试点，你才能获得该子任务的分数。如果你的程序在样例测试点或隐藏测试点中出现了内存泄露，该子任务只会获得15分。

注意你不用同时通过4个子任务再提交，我们会将每一个子任务的代码拆开，分别编译。（但是后一个子任务会依赖于前一个子任务，例如：如果想要第三个子任务正确，必须先保证第一、二个子任务正确。）

考试100%为OJ评分。

D. 发年终奖

题目描述

注意：本题涉及第14课的内容，相关课程的讲授安排在本次作业结束前一周

某公司年终要给所有员工发奖金，按照职级的不同，有不同的基础年终奖金额；根据绩效的评估有不同的bonus。该公司总共有3个职级：`P1`、`P2`、`P3`，其中 `P1` 基础年终奖金额为2000元，`P2` 的基础年终奖金额为5000元，`P3` 的基础年终奖金额为10000元。

绩效为任务的完成程度，按照绩效的大小，也有不同的bonus策略。绩效为0到1的小数，当绩效为[0,0.5)时，在已有的基础上增加1000元。当绩效为[0.5, 0.8)时，年终奖增加金额为绩效乘以基础年终奖。当绩效为[0.8, 1.0]时，年终奖增加金额为两倍的绩效乘以基础年终奖。

输入样例

第一行一个整数n，代表需要计算年终奖金额的人数

接下来n行，每行一个字符串，一个小数，分别代表职级和绩效。

```
2
P1 0.3
P3 0.9
```

输出样例

n个整数，表示每个人最终的年终奖金额。（如果为小数，则向下取整）

```
3000
28000
```

要求

- 不修改 `main.cpp`（`main.cpp` 在这里下载）
- 使用策略模式
- 你需要提交多个文件，包含 `Makefile`，上述文件调用的各种头文件及其 `cpp` 文件。使用 `Makefile` 必须要能生成可执行文件 `main`（不带扩展名）。
- 你应该将你的文件打包成一个 `zip` 压缩包并上传。注意：你的文件应该在压缩包的根目录下，而不是压缩包的一个子文件夹下。

评分标准

OJ评分 30%，人工评分70%