

基于实践的 C++ 个人学习经验分享

上海大学中欧工程技术学院 19 级信息工程专业陈梓康

你好，我是信息工程陈梓康。首先是本人的个人情况介绍。本人其实很早就接触过 C++ 了，并且在正式的课程开始之前，已经学习过一遍 C++ 基本语法。此外，本人也熟悉其他编程语言，包括 MATLAB, C, Python, R, Ubuntu 操作系统, Julia。比如我的 C 语言课程成绩是 99 分，MATLAB 语言课程成绩是 A。因为有使用这些语言来建模、做算法、跑数据（手撸上千行的那种）的需求，因此在类似这样不断实践的过程中，我的编程水平勉强没有低于平均水平。总而言之，在上这门课程之前，C++ 对我而言已不是新鲜的东西。因此我的分享均是基于一定编程基础的，如果是小白的话，还是建议把基础语法应用熟练吧，脚踏实地更好。

关于应试方面。我认为考试难度是比较正常的，再难也无非就是一些 design patterns。很多同学觉得考试很难，其实当你真正沉下心来认认真真地做几遍 TD（包括课外的 TD），你能发现这个考试是非常程序化的。比如一开始是 introduction，然后就是让你首先 define 一些基本的 member 和 method；接着给你一个任务，让你根据题给 code 自己 declare & define 若干个 class。最后记得画一个好看的（bushi）UML 图。如果这些程序化的东西记不住，那就借用一些方法，比如思维导图、艾宾浩斯曲线之类的还记得当初我写 iterator, adaptor 写到吐，已经达到倒背如流的状态了。

不要害怕考试，因为这门课程的考试的性质是达标性的，只要会基础语法以及理解 C++ 的 idea 和 feature，就可以拿到很好的成绩。当然，这门考试其实也并没有我说的这么简单，因为其中的代码细节、背后的代码逻辑、试卷的结构以及负责的 Antoine 老师的出题意图是值得我们细细琢磨的。如果能在考试规定的时间内把这个做好，我觉得应该还是不错的。比如有些 class 之间的 association 是不确定的，它既可以是 composition 也可以是 aggregation，因为它其实是由代码的实现方式决定的。虽然这两种不同实现方法在功能上是一致的，但是 class 之间的关系就是不一样，可能一些同学并没有注意到这一点。当然，这个细节点，授课老师也没有提到过。还有比如，为什么有 observer 这个 design pattern，它解决了一个什么问题，为什么它能够解决这个问题？其实当你了解过 C++ 背后运行/编译代码的流程，就能找到这个答案。此外，当你阅读题给 code 时，如果达到平均水平，我认为你应该能够理解出题老师展示给你的代码的精彩之处（因为不那么写，会出现一些可能很难发现的致命 bug，这算是考试里的彩蛋了）

关于课程项目方面。这个 part 成绩占比 15-20%，不算很多，但是值得我们重视。这里我必须为我们组员感到骄傲，我们小组对我们当时做的项目还是异常满意的。当时把整个程序完全做出来时，已经是异常兴奋。这段经历，不能忘记。回到正题，这个项目有大约上千行的代码量，不知道有没有过万行，好像压缩之后也有 0.5G 大小左右。完全是独立设计，主题是一个桌游。这是一个 0 到 1 的过程，从游戏规则的设计，到代码框架，然后是算法的选择，最后是 GUI 界面的设计。首先我觉得关于规则的设计其实是非常精彩的，至少在我第一次看到这样的设计是非常兴奋的（就是很想玩这个游戏）。代码的架构，我们还是比较遗憾的，因为当初对各种设计模式并没有理解透彻（虽然后面懂了，但是来不及更改框架了），因此在这方面我们小组发生了一些小争执，走了不少弯路。幸运的是，我们没有放弃，求同存异，合作共赢。算法层面，即设计多个电脑玩家，主要由我负责。本来是采取另外小组成员的建议，使用 Markov Chain。但是因为种种原因，最后还是借用梯度优化的方法，设计了一个可以设定任意游戏喜好的电脑玩家。最后，我们还做了

GUI 界面的设计，很精美。还记得，当游戏初始界面的音乐开始播放，那具有史诗感的音乐，令人热血沸腾。值得一提的是，在这样的代码设计过程中，最大的收获便是写各种各样的 interface 了，这里面还是有很多技巧的，就不展开细说，期待各位花时间挖掘彩蛋。然而，很棒的事情是，这个项目拿到了很高很高的分数。说心里话，能够让我们所有人共同进步，这是多么开心的事啊。

关于上课方面，少数同学会觉得跟不上，这里主要指 TD 课。但是，无论如何，不要不来上课或者不听课。老师应该会多次强调，这门课程的教学内容和国内的不太一样。以我个人经历来说，确实是如此。本课程的教学内容偏向于工程，相比于国内，这门课程更多地从应用角度来向你介绍与解释 C++。因此我认为该课程更加实用，并且老师也都很负责。所以要好好听课，虽然好好听也不能代表能考好（划掉）。答疑的机会不要浪费，可以和法国人聊天，锻炼法语（划掉）。

最后说最重要的事，关于学习本门课程的个人经验。C++ 是一门细节比较多的语言，因为经常要考虑到内存管理，比较底层。当然这也是它的优势，它在做计算的时比较快，潜力很大。之前有用 C++ 模拟过一个 NP 的最佳路径问题，计算速度确实比较快。因此，为了学好这门课程，你有必要写一个 list，专门用来记录在学习过程中的心得。为了方便你学习，我把学习笔记放在附件里了。此外，多写代码，多写代码，多写代码。我当初就是因为比较忙，花的精力不够多，才这个下场呜呜呜。

ps: lunde 老师说我的考试成绩不好，突然觉得自己没有资格在这里给大家说三道四，我还是更脚踏实地一点吧。总之大家，多花时间好好准备考试。当然，不要过多在乎这些 GPA，多做实习、比赛、科研。因为一旦当你离开学校系统，进入社会，GPA 很难帮助你。

当然，最后的最后，本人并没有深入了解过 C++，如果读者觉得心理不适或者对你没有任何帮助，请不要阅读本文。（*¬¬）

附件

《学习资料》

1. constexpr 在编译时就可以声明，而不是运行的时方便候。这个东西在声明数组的长度很有用。

1. constexpr N=10;

2. array[N];

2. const 定义时，必须初始化一个值。指向常量的指针是不可改动的，包括对应指向的值。

1. 非法的：

1. const char * ='abcd';

- pc[3] = 'x';

2. 合法的：

1. char* const pc = 'abcd';

- pc[3] = 'x';

3. 在 struct 声明过后（在.h 文件里），后面在使用是可以不加上 struct 字符。struct 是默认 public 的，class 是 private 私有的。

4. 休止符 '\0'

5. 函数的重载 overload

1. C++根据函数的参数来分辨函数，当函数名相同时

6. *(p.nom)=0; *p.nom='\0'; (&p)->nom[0]=0;

1. Struct personne{

- int age;

- } //声明

2. void fonction_init(personne& p){ //p 是对象

- p.age=0;

- }//等价

- void onction_init(personne* p){ //p 是指针

- p->age=0;// 指针的用法

- }//等价

7. 初始化

- personne p1={.nom:"Adam", .age:20};

- personne table1[2]={ "Adam", 20, "Tom", 19};

8. 函数按引用传递

```
int& fonction(personne& p){  
  
    return p.age;  
  
} // 返回的是一个对象。如果是变量则可以进行变量的运算。
```

9. 在某些情况下, 如果不用 inline, 那么我们可以在 class 里面声明一个 function()。

10. this 是当前类对应的 object 的指针, *this 是当前类的 object 本身

11. const 可以放在函数声明的后面, 指不能对数据成员的值修改 (它是个 const)

12. class 里面可以有 namespace 和 class (通过双冒号的方式获取)

13. UML 图的绘制

1. 两种关系
2. 方框
3. 箭头 (数字)

14. 封装原则

1. 面向对象, 封装, 接口

15. public 内的内容可以通过 '.' 来访问。

16. 构造函数(constructor)、析构函数(destructor)与生命周期

1. classe 的 structor 是类的一种特殊的成员函数, 在创建类的新对象时执行。构造器 structor 与 class 同名, 没有确定的返回类型, 不需要 void, 没有 return。构造器可以有括号或者没有括号 (也可以带参数)
2. constructor 中 ":" 的是用来初始化的, {} 是用来赋值的。

在 CPP 文件中: Carte(Couleur c, Nombre v, Forme f, Remplissage r) :couleur(c), nombre(v),
forme(f), remplissage(r) {}

```
MATH::Fraction::Fraction(int n, int d) : numerateur(n),denominateur(d) {  
  
    setFraction(n, d);  
  
    simplification();  
  
}
```

3. 如果 constructor()=delete, 说明该 class 是禁止的。
4. 无参构造器是默认的 (可以不写) = default
5. 要声明 class 的 array 时, 如果包含有参构造器, 则一定要每个元素都要构造。

1. solution1: 声明 class 的指针数组 (class 的指针没有调用 constructor), 每个 point 指向一个 object (这里调用有参 constructor), 记得 delete

2. solution2: 手动添加一个无参 constructor, 手动添加 void setClass()的 method 来分别设置成员的值。

3. solution3: 使用 vector

6. class的~destructor 是类的一种特殊的成员函数, 在删除类的新对象时执行 (~Fraction() {};)。被析构时, 变量存在, 但是它指向的地址不变, 对应的值变了。class 的成员含有指针/数组时, 必须要 destructor (delete[] 数组成员)。

7. 程序的生命周期

1. 编码

2. 预处理

1. pre-processor 会处理#include, #define, #if

3. 编译

1. 静态检查

4. 运行

8. 变量的生命周期

1. 在构造器的花括号中, 变量以的形式存在, 在完成花括号内容后, 相关局部变量会被 destruct

2. 不要返回构造器的局部变量 (它会在 return 之前被删除), 必须得用 new (并且返回这个 new 对应的指针)

3. 注意不要有内存的泄露, 要删干净, new 出来的一定要删。

4. 全体删除: delete[] Pioche; 单个删除: for (something){delete something}

17. 期中考试 (周六 9:00-11:00)

1. 考试内容和 TD4 和 TD5 很接近, 书写规范、清晰明了

2. TD21: 操作符的重载, 尤其是<<, 写在 namespace 外面。(不考后缀操作符 surcharge 不考)

3. TD22: exceptions: throw SetException("something wrong");

4. TD23: 拷贝构造函数/构造器, 赋值构造函数/构造器, 不同类之间的关系, const 的作用 (英/法)

5. (对象的) 数组, 指针,

6. UML 图: (注意枚举类型 (这是一个 class)), composition (实心菱形), aggregation (空心菱形)

7. TD24: singleton pattern, iterator

8. 考二元操作符

9. 数组的扩容不考

10. inline, 类型转换不考

18. 期中复习

1. 创建 class 时, 一定在 namespace (大括号无分号) 中创建, 并且确定 private 和 public (大括号有分号), 再写 constructor, desstructor (~Fratoin(){};)。
2. void 函数可以返回 return 跳出函数。
3. déclarer 是在头文件中操作, définir 是在 cpp 中操作
4. 在 cpp 文件中, 记得加双冒号。
5. 在定义的时候, 记得加 const。
6. constructor 和 copy constructor 要记得用冒号赋值啊 (对于指针数组, 先初始化, 大括号内分别进行赋值)。
7. 只有 new 出来的才要在 cpp 文件中对 constructor 和 destructor 修改 (delete)
8. 注意函数参数类型和是不是有 &, 有没有 return。
9. 注意数组是不可以直接 copy 的, 要用 for 实现, 如果没有相关 copy 的声明的话。
10. **.h:** void Plateau::print(ostream& f =cout) const;

```
1. .cpp: void Plateau::print(ostream& f) const {  
  
    for (size_t i=1; i<nb; i++) {  
  
        if (i%4==0)    f << "\n";  
  
        f << *cartes[i] << " ";  
  
    }  
  
    f << "\n";  
  
}
```

11. 在创建新的数组时, 记得 delete 之前的数组;

```
1. auto old = cartes;  
  
for (size_t i=0; i<nb; i++) cartes[i]=newtb[i];  
  
delete[] old;
```

12. afficher 打印的意思, dupliquer 复制的意思, le constructeur de recopie et l'opérateur d'affectation 拷贝和赋值, en lecture 只读, parcourir 遍历
13. la classe *Controleur* compose à la fois la classe *Jeu* et la classe *Pioche* avec une multiplicité de 1 à chaque fois.(composition)
14. 画 UML 图时, num 可以看成 class。combinaison 只有一端有数字。

Fraction
-numérateur: int -denominateur: int
-simplification() +Fraction(in n:int=0,in d:int=1) +getNumerator(): int const +getDenominateur(): int const +setFraction(in n:int,in d:int)

1.

15. `std::cerr << "erreur : denominateur nul\n";`

16. 用 point array 初始化 constructor, 先在冒号后面 new, 再在大括号里逐个赋值。

19. surcharge 重载: 对于没有定义的操作符, 我们定义一些操作符。

1. 算术运算符不是一种 method, 要放在 class 外声明, 两个变量参数。在 class 里面, 只要第二个参数 (因为 surcharge 是 class 的一个 method, 所以默认有一个指向自己 class 类别 的 this 指针 (所以可以在后面加上 const))。当参数为一个 object 时, 应该传入它的 reference (const Fraction& f1)。

2. 后缀操作符 "++", 则需要再加一个参数 "int" 即可, 并且返回的类型是一个 class 本身 (先传指针, 再运算)。前缀操作符 "++" 返回的类型是一个 class 的指针 (先运算, 再传指针)。

1. 在 CPP 文件中

1. **+符号的 surcharge 返回参数没有&**

2. `MATH::Fraction* MATH::Fraction::operator++() { //加法, 指针, pre (错误写法, 一定要用&)`

```
numérateur+=numérateur+denominateur;
```

```
simplification();
```

```
return this;
```

```
}
```

3. `MATH::Fraction MATH::Fraction::operator++(int) { //指针, 加法, post`

```
Fraction frac(numérateur, denominateur);
```

```
numérateur+=denominateur;
```

```
simplification();
```

```
return frac;
```

```
}
```

4. `MATH::Fraction& MATH::Fraction::operator++() { //官方解答`

```
setFraction(getNumerator()+getDenominateur(),getDenominateur
```

```

    ());

    return *this;

}

```

5. `const MATH::Fraction MATH::Fraction::operator++(int){//官方解答`

```

    Fraction f(getNumerator(), getDenominateur()); // copie de la
fraction

    setFraction(getNumerator()+getDenominateur(),getDenominateur
());

    return f;

}

```

3. "<<"是在 `std::ostream` 中的，所以不能在 `namespace` 中声明。`std::ostream& operator<<(std::ostream& F, const MATH::Fraction& frac);`

在 `cpp` 文件中：

```

std::ostream& operator<<(std::ostream& F, const MATH::Fraction& frac){

    F<<frac.getNumerator();

    if (frac.getDenominateur()!=1){

        F<< '/' <<frac.getDenominateur();

    }

    return F;

}

```

4. `void afficher(ostream& f = cout) const { f<<toString(); }`

5. “==”“=”一定得在 `class` 中声明。

6. `operator[] const(...)`

7. `(".", ".*", "::")`是不可以被 `surcharge` 的

8. 二元操作符，自己读取两个变量“+”

9. `implicit conversion` 隐性转换

10. `friend` 空格后抄一遍在 `class` 外声明的 `surcharge`，也可以访问 `class private` 了。在哪个 `class` 里面就可以访问哪个 `class`。（可以放在 `private` 或 `public` 里）

20. Exception 异常处理

1. `Throw`

1. 可以接"字符串" (需要 try, catch)

2. 可以接 class (需要 try, catch)

2. Try: **try** 块中的代码标识将被激活的特定异常。

3. Catch: 关键字用于捕获异常, 紧跟在 try 之后, 并且传入 throw 返回的参数的 type (可以是 int i, 但是这个异常句柄, 可以用来调用异常值)。

```
1. const char* what() const noexcept { return info.c_str(); }
```

21. macro(宏) assert 断言

```
1. #include "assert.h"
```

2. 当判断为 1 时, 继续运行, 否则 (在编译的时候) 跳出。

22. enum 枚举

```
1. enum color_set2 { GREEN, RED, YELLOW, WHITE } color3, color4;
```

```
2. color3=RED;           //将枚举常量值赋给枚举变量
   color4=color3;        //相同类型的枚举变量赋值, color4的值为 RED
   int i=color3;         //将枚举变量赋给整型变量, i的值为 1
   int j=GREEN;          //将枚举常量赋给整型变量, j的值为 0
```

3. //比较同类型枚举变量 color3, color4 是否相等

```
1. if (color3==color4) cout<<"相等";
```

4. //输出的是变量 color3 与 WHITE 的比较结果, 结果为 1

```
1. cout<< color3<WHITE;
```

5. cout<< color3; //输出的是 color3 的整数值, 即 RED 的整数值 1

6. enum 变量只能参与赋值和关系运算以及输出操作, 参与运算时用其本身的整数值

7. ":" 作为遍历的用法

```
1. int couleurs[3]=[GREEN, RED, YELLOW, WHITE];
```

```
2. for (auto x : couleurs)
```

23. explicit 拒绝类型强制转换, 因此可以选择构造器。

24. 拷贝构造器 copy constructor: 深拷贝, 浅拷贝

1. 声明: cat(cat&);

2. 定义: cat::cat(cat& other) {

```
    age = other.age;
```

```
    ...
```

```
}
```

3. 自动调用拷贝器的情况:

25. association 联合, aggregation 聚合 and composition 关联/组合: 重点是谁负责谁的生命周期

1. composition 的 copy constructor 和 aggregation 的 copy constructor 不一样。一个要 new, 一个只需要传指针。
2. association: is a weaker form of relationship and in code terms indicates that a class uses another by parameter or return type.

```
1. class Foo{  
  
    public:  
  
        void Baz (Bar bar) {  
  
        }  
  
};
```

26. composition: implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.

```
1. class Foo{  
  
    private:  
  
        Bar* bar = new Bar();  
  
}
```

27. aggregation: implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

```
1. class Foo {  
    private:  
  
        Bar* bar;  
  
        Foo(Bar bar) {  
            this->bar = bar;  
        }  
  
}
```

28. Rules of three

1. destructor, copy, operator=.

29. 单例设计模式

1. 只提供唯一一个类的实例, 任何位置都可以通过接口获得该实例 `Jeu::getInstance()`; (这是一个对象)

```
1. static Jeu& Jeu::getInstance() {
```

```

        static Jeu jeu;

        return jeu;

    }

```

2. des interets de mettre en place le Design Pattern Singleton:

3. constructor, copy constructor(delete), operator=(delete), destructor()都是 private

4. 方法一 (.h) : static Jeu& getInstance();表示多个对象共同调用这个方法。getInstance()简单声明了 class jeu。

```

1. static Jeu& Jeu::getInstance() {

    static Jeu jeu;

    return jeu;

}

```

5. 方法二 (.h) : 用指针指向 jeu, 从而进行内存管理。

```

1. static Jeu& getInstance() {

    if (instance==nullptr) instance = new Jeu;

    return *instance;

}

static void libererInstance() {delete instance; instance = nullptr;}

```

6. 方法三 (.h) : 利用 struct Handler 和 point 共同进行生命周期管理。

1. private:

```

1. static struct Handler{

    Jeu* instance; //instance = new Jeu;这个代码是在声明
                    之后的内容

    Handler(): instance(nullptr) {}

    ~Handler(){

        delete instance;

        instance = nullptr; //凡是 delete instance, 还要
                             再把 nullptr 赋值给 instance

    }

};

```

2. static Handler handler;

2. public:

```

1. static Jeu& getInstance(){

    if (handler.instance==nullptr) handler.instance = new
    Jeu;

    return *handler.instance;

}

```

```

2. static void libererInstance() {

    delete handler.instance;

    handler.instance = nullptr;//赋值为 nullptr 是为了在
    调用的时候 (getInstance())方便判断是否是单例。

}

```

7. static 属性/方法可以直接通过 Jeu::getInstance() (这个就是 jeu, 只不过是单例) 这样通过 class 访问。

8. getInstance()返回类型只能为&

9. 注意单例模式和友元的题目的文法 (ex24-3)

30. 迭代器的设计模式

1. iterator 返回的没有&, 因为构造是私有的, 不得随便访问/调用。

2. class iterator 中的 method 前后的 const 是对称的。

3. 方法一:

```

1. class Iterator{

    private:

        size_t = 0;

        friend class Jeu;//Jeu::getCarte()是 private

        Iterator() = default;//记得 constructor 在 private 中声明, 注意
        初始化的方式 (根据 private member 来判断) 。

    public:

        bool isDone();

        void next();//要有 bool 判断函数和 throw

        const Carte& currentItem();//也要有判断函数和 throw

};

2. Iterator getIterator() const {return Iterator();} //在 Jeu 的 public 中

3. friend class Iterator;

```

4. 迭代器 iterator 是 the public member of Jeu

5. 方法二：par les constructor standards du C++(STL)，对指针的指针进行操作

```
1. class const_Iterator{
```

```
    private:
```

```
        Carte** current = nullptr;
```

```
        const_Iterator(Carte** c): current(c) {};
```

2. 在 const_instructor 的 private 里要声明 constructor

3. 在 const_instructor 的 public 里要 surcharge *, ++, !=

4. begin(), end()在大的 class 里面声明，并且返回对象是 iterator 本身，不是&

6. composition 关系中的 private 是否可以访问？是可以的，互相访问。

31. vect

```
1. vector<int> vect;
```

```
2. vect.push_back(10); //在最后添加一个值
```

```
3. vector<int> vect(3,10); //3个值为10的vector
```

```
4. vector<int> vect{10, 20, 30}; //赋值定义方法
```

```
5. vector<int> vect(ptr, ptr+n); //选定两个指针之间的序列
```

```
6. vectot<int> vect(vect1.begin(), vect1.end());
```

```
7. it = find(vect.begin(), vect.end(), vet); //如果没找到，为end()这是一个指针
```

32. inheritance

1. 继承类可以访问父 protected member 和父 public member，但是非继承类不可以访问。继承类不可以访问被继承类 private member。如果一个 member 想被继承类访问，不被非继承类访问，所以应该是 protected。如果希望永远是不可访问的，那么应该是 private。

2. hériter public: 继承的 private, protected, public 保持不变

3. hériter protected: 继承的 private member 为 private

4. hériter private: 继承的 private 为 private, public 为 protected

5. 被继承类的 destructor 一般是 virtual function。constructor 不可以是 virtual function。

6. 继承类要自己加上一个自己的 constructor，不能继承继承类的 constructor，destructor 和 copy constructor, surcharge, friend

7. 生命周期：最先初始化被继承类，最后释放被继承类，即 base-derive-dérive-base

8. 重载：(cpp)

```
1. this->Evt1jDur::operator=(r); //利用被继承类的 constructor 定义 private
```

member

2. `personne = r.personne;`

3. ...

9. 当实现一个在不同继承类不同定义的 method 时，要用虚函数 `virtual`。当被继承类被定义为 `virtual` 时，被继承类默认该 method 为 `virtual function`，反之不成立。

10. `surcharge<<` 时，函数传入的参数 `Evt1j e` 会根据被继承类 `Evt1j` 改变类型（隐式转换，有点像 `virtual`）。在继承中，如果不能分别是哪一个继承类，可以直接用被继承类，但是一定要加上 `&` 或 `*`。

11. 异构（传入参数不同），但是在代码层面，传入的参数都是 `Evt1j*`（如果是 `pointor` 或 `reference`，则会自动转化 `class` 类型）。

12. `iterator` 继承类

1. `class iterator: public std::vector<Evt*>::iterator{`

...

`iterator(std::vector<Evt*>::iterator& it):`

`std::vector<Evt*>::iterator(it){}`

`};`

33. `classe abstraite` 抽象类

1. 是一个 `class` 类型，可以有 `membre`, `method`，不能生成对象。

2. 虚函数在继承类中必须要被实现

3. `généralisation` 操作：提取公因式；`spelisation` 操作：不断丰富里面的方法

34. `creat pattern`

35. 模板设计方法 `patron`

1. 返回的一定是一个 `*`

1. `EvtPj* EvtPj::clone() const { return new EvtPj(*this); }`

2. `template<class T>`

36. 适配器模式 `pattern de l'adaptateur`

1. 把不同的继承类放在同一个 `vector` 里，包含的类型一定要用 `*`

2. `Log` 提供一个 `add` 方法和一个 `display` 方法（`adaptateur de classe`, `adaptateur d'objet`）

3. 在 `h` 文件中

`class MyLog : public Log, private Agenda { // 这个用得多一点`

`public:`

```

        void addEvt(const Date& d, const Horaire& h, const string& s);

        void displayLog(ostream& f) const;

};

class MyLog : public Log {

private:

    Agenda evts;

public:

    void addEvt(const Date& d, const Horaire& h, const string& s);

    void displayLog(ostream& f) const;

};

```

4. class 类型转换

1. 被继承类转化为继承类，用 `dynamic_cast<Rdv*>(ptrx)` (不成功返回 `nullptr`)
2. 这玩意可以判断这个对象是不是这对应的个类

```

1. inline Date getDate(const Evt& e){

    const Evt1j* pt1=dynamic_cast<const Evt1j*>(&e);

    const EvtPj* pt2=dynamic_cast<const EvtPj*>(&e);

    if (pt1) return pt1->getDate();

    if (pt2) return pt2->getDateDebut();

}

```

37. 观察者模式

1. 通过基函数来避免相互引用的问题。

38. set

1. `#include <algorithm>`

```
find()
```

39. virtual destructor 虚拟析构函数 (针对继承类里面有 new 的情况)

40. override 一般在虚函数后面。

41. 在继承类中，访问私有继承的被继承类，用 `using MyVector::ierator;`

1. 可以 `using member, methods(无参), constructor(无括号);`

42. 模板设计方法

1. A0 适配器模式

1. `template< class T, class CONT=Vector<T> >`//Design Pattern Stratégie 设计模式，这是对类似的模板的写法的偷懒

```
class StackP {
```

```
private:
```

```
    CONT cont;
```

```
public:
```

```
class iterator {
```

```
// Le type : type de l'iterateur du contenur dont on fait une composition
```

```
private:
```

```
    typename CONT::iterator courant;//借用 CONT 的 iterator, dependent scope
```

```
public:
```

```
    iterator():courant() {}
```

```
    iterator(typename CONT::iterator c):courant(c) {}
```

```
//因此没有继承以下的 method
```

```
    iterator& operator++() { courant++; return *this; }
```

```
    bool operator!=(const iterator& it) const { return courant!=it.courant; }
```

```
    T& operator*() const { return *courant; }
```

```
//需要定义 method, 因为这是对本 class 做操作, 半独立的 class。
```

```
    iterator operator++(int) { iterator tmp=*this; courant++; return tmp; }
```

```
    bool operator==(const iterator& it) const { return courant==it.courant; }
```



```
};
```

```
//不能是&, 因此return的东西是非参数, 非member
```

```
iterator begin() { return iterator(cont.begin());
```

```
iterator end() { return iterator(cont.end()); }
```

```
};
```

2. AC 适配器模式

```
template< class T, class CONT=Vector<T> >
```

```
class StackP : private CONT {
```

```
public:
```

```
class iterator : public CONT::const_iterator {
```

```
// Le type : type de l'iterateur du contenu dont on fait une composition
```

```
private:
```

```
typename CONT::iterator courant;//借用 CONT 的 iterator, dependent scope
```

```
public:
```

```
const_iterator():CONT::iterator() {}
```

```
const_iterator(typename CONT::iterator c):courant(c) {}
```

```
//不需要定义其他method, 因为是 herite public, 依赖的 class
```

```
};
```

```
//不能是&, 因此return的东西是非参数, 非member
```

```
iterator begin() { return iterator(CONT::begin()); }
```

```
iterator end() { return iterator(CONT::end()); }
```

```
};
```

3. 在模板实际方法中, 如果变量和模板有关, 一定要用 typename

```
1. typename CONT::iteratorcourant;
```

4. template function 兼容众多类似的 class

```
1. template<class Container, class<T>> bool Search (const Container&
    container, const T& value) {

    for (auto it=container.begin(); it!=container.end(); ++it){

        .....

    }

}
```

5. 在模板设计方法中, 如果返回定义模板, 都是 Vector<T> 这样的类型

6. 模板编程方法

```
template<class IT, class COMP>

IT TD::element_minimum(IT it1, IT it2, COMP comp){

    IT it_min=it1;

    ++it1;

    while(it1!=it2){

        if (comp(*it1,*it_min)) it_min=it1;

        ++it1;

    }

    return it_min;

}
```

43. 继承类要在 constructor 中定义被继承类