1. C++设计模式之单例模式。

单例模式：只能实例化一个类的模式，通过将构造函数私有和使用static来实现该功能

1. /\*
2. \* 关键代码：构造函数是私有的，不能通过赋值运算，拷贝构造等方式实例化对象。
3. \*/
4. ​
5. //懒汉式一般实现：非线程安全，getInstance返回的实例指针需要delete
6. **class** Singleton
7. {
8. **public**:
9. **static** Singleton\* getInstance();
10. ~Singleton(){}
11. ​
12. **private**:
13. Singleton(){}                                        //构造函数私有
14. Singleton(**const** Singleton& obj) = **delete**;            //明确拒绝
15. Singleton& operator=(**const** Singleton& obj) = **delete**; //明确拒绝
17. **static** Singleton\* m\_pSingleton;
18. };
19. ​
20. Singleton\* Singleton::m\_pSingleton = NULL;
21. ​
22. Singleton\* Singleton::getInstance()
23. {
24. **if**(m\_pSingleton == NULL)
25. {
26. m\_pSingleton = **new** Singleton;
27. }
28. **return** m\_pSingleton;
29. }

饿汉模式：在单例类定义的时候就进行实例化，在访问量较大或者访问的线程较多的时候采用饿汉模式实现。

1. //饿汉式：线程安全，注意一定要在合适的地方去delete它
2. **class** Singleton
3. {
4. **public**:
5. **static** Singleton\* getInstance();
6. **private**:
7. Singleton(){}                                    //构造函数私有
8. Singleton(**const** Singleton&) = **delete**;            //明确拒绝
9. Singleton& operator=(**const** Singleton&) = **delete**; //明确拒绝
10. ​
11. **static** Singleton\* m\_pSingleton;
12. };
13. ​
14. Singleton\* Singleton::m\_pSingleton = **new** Singleton();
15. ​
16. Singleton\* Singleton::getInstance()
17. {
18. **return** m\_pSingleton;
19. }
20. 设计模式之观察者模式

定义对象之间的一种一对多的依赖关系，当一个对象的状态发生改变时所有依赖于它的对象都要得到通知并更新

1. /\*
2. \* 关键代码：在目标类中增加一个ArrayList来存放观察者们。
3. \*/
4. #include <iostream>
5. #include <list>
6. #include <memory>
7. ​
8. **using** **namespace** std;
9. ​
10. **class** View;
11. ​
12. //被观察者抽象类   数据模型
13. **class** DataModel
14. {
15. **public**:
16. **virtual** ~DataModel(){}
17. **virtual** **void** addView(View\* view) = 0;
18. **virtual** **void** removeView(View\* view) = 0;
19. **virtual** **void** notify() = 0;   //通知函数
20. };
21. ​
22. //观察者抽象类   视图
23. **class** View
24. {
25. **public**:
26. **virtual** ~View(){ cout << "~View()" << endl; }
27. **virtual** **void** update() = 0;
28. **virtual** **void** setViewName(**const** string& name) = 0;
29. **virtual** **const** string& name() = 0;
30. };
31. ​
32. //具体的被观察类， 整数模型
33. **class** IntDataModel:**public** DataModel
34. {
35. **public**:
36. ~IntDataModel()
37. {
38. m\_pViewList.clear();
39. }
40. ​
41. **virtual** **void** addView(View\* view) override
42. {
43. shared\_ptr<View> temp(view);
44. auto iter = find(m\_pViewList.begin(), m\_pViewList.end(), temp);
45. **if**(iter == m\_pViewList.end())
46. {
47. m\_pViewList.push\_front(temp);
48. }
49. **else**
50. {
51. cout << "View already exists" << endl;
52. }
53. }
54. ​
55. **void** removeView(View\* view) override
56. {
57. auto iter = m\_pViewList.begin();
58. **for**(; iter != m\_pViewList.end(); iter++)
59. {
60. **if**((\*iter).get() == view)
61. {
62. m\_pViewList.erase(iter);
63. cout << "remove view" << endl;
64. **return**;
65. }
66. }
67. }
68. ​
69. **virtual** **void** notify() override
70. {
71. auto iter = m\_pViewList.begin();
72. **for**(; iter != m\_pViewList.end(); iter++)
73. {
74. (\*iter).get()->update();
75. }
76. }
77. ​
78. **private**:
79. list<shared\_ptr<View>> m\_pViewList;
80. };
81. ​
82. //具体的观察者类    表视图
83. **class** TableView : **public** View
84. {
85. **public**:
86. TableView() : m\_name("unknow"){}
87. TableView(**const** string& name) : m\_name(name){}
88. ~TableView(){ cout << "~TableView(): " << m\_name.data() << endl; }
89. ​
90. **void** setViewName(**const** string& name)
91. {
92. m\_name = name;
93. }
94. ​
95. **const** string& name()
96. {
97. **return** m\_name;
98. }
99. ​
100. **void** update() override
101. {
102. cout << m\_name.data() << " update" << endl;
103. }
104. ​
105. **private**:
106. string m\_name;
107. };
108. 设计模式之工厂模式

通过使用一个共同的接口来指向新创建的对象。

1. #include <iostream>
2. **using** **namespace** std;
3. //定义产品类型信息
4. **typedef** **enum**
5. {
6. Tank\_Type\_56,
7. Tank\_Type\_96,
8. Tank\_Type\_Num
9. }Tank\_Type;
10. ​
11. //抽象产品类
12. **class** Tank
13. {
14. **public**:
15. **virtual** **const** string& type() = 0;
16. };
17. ​
18. //具体的产品类
19. **class** Tank56 : **public** Tank
20. {
21. **public**:
22. Tank56():Tank(),m\_strType("Tank56")
23. {
24. }
25. ​
26. **const** string& type() override
27. {
28. cout << m\_strType.data() << endl;
29. **return** m\_strType;
30. }
31. **private**:
32. string m\_strType;
33. };
34. ​
35. //具体的产品类
36. **class** Tank96 : **public** Tank
37. {
38. **public**:
39. Tank96():Tank(),m\_strType("Tank96")
40. {
41. }
42. **const** string& type() override
43. {
44. cout << m\_strType.data() << endl;
45. **return** m\_strType;
46. }
47. ​
48. **private**:
49. string m\_strType;
50. };
51. ​
52. //工厂类
53. **class** TankFactory
54. {
55. **public**:
56. //根据产品信息创建具体的产品类实例，返回一个抽象产品类
57. Tank\* createTank(Tank\_Type type)
58. {
59. **switch**(type)
60. {
61. **case** Tank\_Type\_56:
62. **return** **new** Tank56();
63. **case** Tank\_Type\_96:
64. **return** **new** Tank96();
65. **default**:
66. **return** nullptr;
67. }
68. }
69. };
70. C++智能指针shared\_ptr, weak\_ptr, unique\_ptr

智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。

shared\_ptr可以用来实现共享所有权的概念。多个智能指针可以引用同一个对象，当最后一个智能指针销毁时，对象销毁。每个shared\_ptr对象在内部指向两个内存位置（1.指向对象的指针。2.用于控制引用计数的指针）

1. #include <iostream>
2. #include  <memory> // 需要包含这个头文件
4. **int** main()
5. {
6. // 使用 make\_shared 创建空对象
7. std::shared\_ptr<**int**> p1 = std::make\_shared<**int**>();
8. \*p1 = 78;
9. std::cout << "p1 = " << \*p1 << std::endl; // 输出78
11. // 打印引用个数：1
12. std::cout << "p1 Reference count = " << p1.use\_count() << std::endl;
14. // 第2个 shared\_ptr 对象指向同一个指针
15. std::shared\_ptr<**int**> p2(p1);
17. // 下面两个输出都是：2
18. std::cout << "p2 Reference count = " << p2.use\_count() << std::endl;
19. std::cout << "p1 Reference count = " << p1.use\_count() << std::endl;
21. // 比较智能指针，p1 等于 p2
22. **if** (p1 == p2) {
23. std::cout << "p1 and p2 are pointing to same pointer\n";
24. }
26. std::cout<<"Reset p1 "<<std::endl;
28. // 无参数调用reset，无关联指针，引用个数为0
29. p1.reset();
30. std::cout << "p1 Reference Count = " << p1.use\_count() << std::endl;
32. // 带参数调用reset，引用个数为1
33. p1.reset(**new** **int**(11));
34. std::cout << "p1  Reference Count = " << p1.use\_count() << std::endl;
36. // 把对象重置为NULL，引用计数为0
37. p1 = nullptr;
38. std::cout << "p1  Reference Count = " << p1.use\_count() << std::endl;
39. **if** (!p1) {
40. std::cout << "p1 is NULL" << std::endl; // 输出
41. }
42. **return** 0;
43. }

unique\_ptr可用用来实现互斥所有权的概念，在一段时间内，只有一个智能指针可以指向对象。当然，这种所有权是可以转移的。这种智能指针可以有效地避免资源泄漏，比如忘记调用delete来释放堆对象。

1. Static 关键字总结

5.1当同时编译多个文件时，所有未加static前缀的全局变量和函数都具有全局可见性。

5.2存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。共有两种变量存储在静态存储区：全局变量和static变量，只不过和全局变量比起来，static可以控制变量的可见范围.

5.3在类中声明static变量或者函数时，初始化时使用作用域运算符来标明它所属类，因此，静态数据成员是类的成员，而不是对象的成员，这样就出现以下作用：

(1): 类的静态成员函数是属于整个类而非类的对象，所以它没有this指针，这就导致 了它仅能访问类的静态数据和静态成员函数。

(2): 不能将静态成员函数定义为虚函数。

(3): 初始化在类体外进行，而前面不加static

1. C++ Volatile

用来修饰变量的，表明某个变量的值可能会随时被外部改变，因此这些变量的存取不能被缓存到寄存器，每次使用需要重新读取。

1. C++ 指针和引用的区别

* 指针只是一个变量，只不过这个变量存储的是一个地址；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已，不占用内存空间。
* 引用必须在定义的时候初始化，而且初始化后就不能再改变；而指针不必在定义的时候初始化，初始化后可以改变。
* 指针可以为空，但引用不能为空（这就意味着我们拿到一个引用的时候，是不需要判断引用是否为空的，而拿到一个指针的时候，我们则需要判断它是否为空。这点经常在判断函数参数是否有效的时候使用。）

1. 纯虚函数和抽象类

在很多情况下，基类本身生成对象是不合情理的。为了解决这个问题，方便使用类的多态性，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;）纯虚函数不能再在基类中实现，编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。

1. Const 的作用
2. 限定变量不可更改
3. 限定成员函数不能更改任何数据成员
4. const char \*p 表示 指向的内容不能改变，char \* const p，就是将P声明为常指针，它的地址不能改变，是固定的，但是它的内容可以改变。
5. C++的深拷贝和浅拷贝

深拷贝与浅拷贝：

浅拷贝：默认的复制构造函数只是完成了对象之间的位拷贝，也就是把对象里的值完全复制给另一个对象，如A=B。这时，如果B中有一个成员变量指针已经申请了内存，那A中的那个成员变量也指向同一块内存。

这就出现了问题：当B把内存释放了（如：析构），这时A内的指针就是野指针了，出现运行错误。

深拷贝：自定义复制构造函数需要注意，对象之间发生复制，资源重新分配，即A有5个空间，B也应该有5个空间，而不是指向A的5个空间。

1. vector中的size()和capacity()

size()指容器当前拥有的元素个数(对应的resize(size\_type)会在容器尾添加或删除一些元素，来调整容器中实际的内容，使容器达到指定的大小。)；capacity()指容器在必须分配存储空间之前可以存储的元素总数。

1. 快速排序
2. //快速排序，随机选取哨兵放前面
3. **void** QuickSort(**int**\* h, **int** left, **int** right)
4. {
5. **if**(h==NULL) **return**;
6. **if**(left>=right) **return**;
8. //防止有序队列导致快速排序效率降低
9. srand((unsigned)time(NULL));
10. **int** len=right-left;
11. **int** kindex=rand()%(len+1)+left;
12. Swap(h[left],h[kindex]);
14. **int** key=h[left],i=left,j=right;
15. **while**(i<j)
16. {
17. **while**(h[j]>=key && i<j) --j;
18. **if**(i<j) h[i]=h[j];
19. **while**(h[i]<key && i<j) ++i;
20. **if**(i<j) h[j]=h[i];
21. }
23. h[i]=key;
25. //QuickSort(&h[left],0,i-1);
26. //QuickSort(&h[j+1],0,right-j-1);
28. QuickSort(h,left,i-1);
29. QuickSort(h,j+1,right);
30. }
31. 冒泡排序
32. //冒泡排序
33. **void** BubbleSort(**int**\* h, **size\_t** len)
34. {
35. **if**(h==NULL) **return**;
36. **if**(len<=1) **return**;
37. //i是次数，j是具体下标
38. **for**(**int** i=0;i<len-1;++i)
39. **for**(**int** j=0;j<len-1-i;++j)
40. **if**(h[j]>h[j+1])
41. Swap(h[j],h[j+1]);
43. **return**;
44. }
45. 选择排序
46. //选择排序
47. **void** SelectionSort(**int**\* h, **size\_t** len)
48. {
49. **if**(h==NULL) **return**;
50. **if**(len<=1) **return**;
52. **int** minindex,i,j;
53. //i是次数，也即排好的个数;j是继续排
54. **for**(i=0;i<len-1;++i)
55. {
56. minindex=i;
57. **for**(j=i+1;j<len;++j)
58. {
59. **if**(h[j]<h[minindex]) minindex=j;
60. }
61. Swap(h[i],h[minindex]);
62. }
64. **return**;
65. }
66. 插入排序
67. //插入排序
68. **void** InsertSort(**int**\* h, **size\_t** len)
69. {
70. **if**(h==NULL) **return**;
71. **if**(len<=1) **return**;
73. **int** i,j;
74. //i是次数，也即排好的个数;j是继续排
75. **for**(i=1;i<len;++i)
76. **for**(j=i;j>0;--j)
77. **if**(h[j]<h[j-1]) Swap(h[j],h[j-1]);
78. **else** **break**;
80. **return**;
81. }
82. 归并排序
83. //归并排序
84. **void**  MergeArray(**int**\* arr, **size\_t** left, **size\_t** mid, **size\_t** right, **int**\* temp)
85. {
86. **if**(arr==NULL) **return**;
88. **size\_t** i=left,j=mid+1,k=0;
89. **while**(i<=mid && j<=right)
90. {
91. **if**(arr[i]<=arr[j])
92. {
93. temp[k++]=arr[i++];
94. **continue**;
95. }
97. temp[k++]=arr[j++];
98. }
100. **while**(i<=mid)
101. temp[k++]=arr[i++];
103. **while**(j<=right)
104. temp[k++]=arr[j++];
106. memcpy(&arr[left],temp,k\***sizeof**(**int**));
108. **return**;
109. }
111. **void** MMergeSort(**int**\* arr, **size\_t** left, **size\_t** right, **int**\* temp)
112. {
113. **if**(left<right)
114. {
115. **size\_t** mid=(left+right)/2;
116. MMergeSort(arr, left, mid, temp);
117. MMergeSort(arr, mid+1,right, temp);
118. MergeArray(arr,left, mid, right, temp);
119. }
120. }
122. **void** MergeSort(**int**\* h, **size\_t** len)
123. {
124. **if**(h==NULL) **return**;
125. **if**(len<=1) **return**;
126. **int**\* temp=(**int**\*)calloc(len,**sizeof**(**int**));
127. MMergeSort(h, 0, len-1, temp);
129. memcpy(h,temp,**sizeof**(**int**)\*len);
131. free(temp);
133. **return**;
134. }
135. 构造函数析构函数可以作为内联函数吗

构造函数、析构函数、虚函数可以声明为内联函数，这在语法上是正确的。

编译器并不真正对声明为inline的构造和析构函数内联，因为编译器会在构造和析构函数中添加额外的操作（申请/释放内存，构造/析构对象等），致使构造函数/析构函数并不像看上去的那么精简。

1. Assert()

其作用是如果它的条件返回错误，则终止程序执行。可以通过定义 NDEBUG 来关闭 assert，但是需要在源代码的开头，include <assert.h> 之前。

#define NDEBUG // 加上这行，则 assert 不可用

#include <assert.h>

assert( p != NULL ); // assert 不可用

1. Sizeof()

* sizeof 对数组，得到整个数组所占空间大小。
* sizeof 对指针，得到指针本身所占空间大小。

1. extern “C”

* 被 extern 限定的函数或变量是 extern 类型的
* 被 extern "C" 修饰的变量和函数是按照 C 语言方式编译和链接的

extern "C" 的作用是让 C++ 编译器将 extern "C" 声明的代码当作 C 语言代码处理，可以避免 C++ 因符号修饰导致代码不能和C语言库中的符号进行链接的问题。

#ifdef \_\_cplusplus

extern "C" {

#endif

void \*memset(void \*, int, size\_t);

#ifdef \_\_cplusplus

}

#endif

1. C++ 中的struct和class

* 默认的继承访问权限。struct 是 public 的，class 是 private 的。
* struct 作为数据结构的实现体，它默认的数据访问控制是 public 的，而 class 作为对象的实现体，它默认的成员变量访问控制是 private 的。

1. explicit 关键字

* explicit 修饰构造函数时，可以防止隐式转换和复制初始化
* explicit 修饰转换函数时，可以防止隐式转换，但 按语境转换 除外

1. C++ 成员初始化列表

* 少了一次调用默认构造函数的过程
* 有些场合必须要用初始化列表：

常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面

引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面。

没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化

1. 抽象类，接口类，聚合类

* 抽象类：含有纯虚函数的类
* 接口类：仅含有纯虚函数的抽象类
* 聚合类：用户可以直接访问其成员，并且具有特殊的初始化语法形式。满足如下特点：

所有成员都是 public

没有定义任何构造函数

没有类内初始化

没有基类，也没有 virtual 函数

1. Linux内核的同步方式

* 原子操作
* 信号量
* 自旋锁
* 读写锁

1. 线程和进程之间拥有和共享的资源

* 进程

私有：地址空间、堆、全局变量、栈、寄存器

共享：代码段，公共数据，进程目录，进程 ID

* 线程

私有：线程栈，寄存器，程序计数器

共享：堆，地址空间，全局变量，静态变量

1. 数据结构的堆和栈

堆：堆是一种经过排序的树形数据结构，每个节点都有一个值，通常我们说堆的数据结构是指二叉树。所以堆在数据结构中通常可以看作一颗树的数组对象。堆需要满足以下的两个性质。

* 堆中的某个节点的值总是不大于或不小于其父节点的值。
* 堆是一个完全二叉树。

堆分为两种情况，有最大堆和最小堆。将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。下图图一就是一个最大堆，图二就是一个最小堆。在一个摆放好元素的最小堆中，可以看到，父结点中的元素一定比子结点的元素要小，但对于左右结点的大小则没有规定谁大谁小。

栈：是限定仅在表尾进行插入和删除操作的线性表。我们把允许插入和删除的一端称为栈顶，另一端称为栈底，不含任何数据元素的栈称为空栈。栈的特殊之处在于它限制了这个线性表的插入和删除位置，它始终只在栈顶进行。

1. 函数调用的过程

* 参数拷贝（压栈），注意顺序是从又到左
* 保存函数调用的下一条指令的位置
* 移动新的ebp,esp形成新的栈帧结构
* 压栈形成临时变量并执行相关的操作
* 日返回一个值
* 出栈

1. 数字证书的颁发过程

用户产生了自己的密钥对，并将公共密钥及部分个人身份信息传送给一家认证中心。认证中心在核实身份后，将执行一些必要的步骤，以确信请求确实由用户发送而来，然后，认证中心将发给用户一个数字证书，该证书内附了用户和他的密钥等信息，同时还附有对认证中心公共密钥加以确认的数字证书。当用户想证明其公开密钥的合法性时，就可以提供这一数字证书。

1. 数字证书里的内容

证书的版本信息

证书的序列号

证书的签名算法

证书的有效期

证书所有人的签名

证书所有人的公开密钥

证书发行者对证书的签名

1. TLS 握手的过程



