



Himanshu Mishra <himanshumishra3112@gmail.com>

Learning

43 messages

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Thu, May 28, 2015 at 4:11 PM

State and Strategy design:

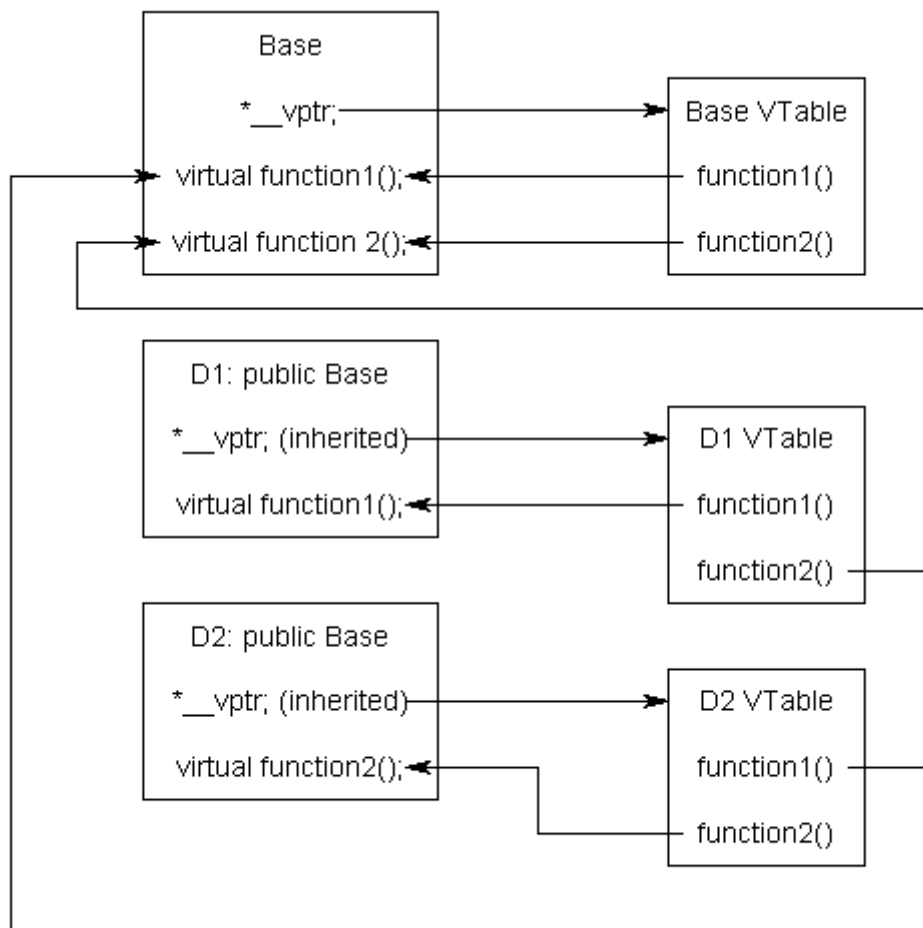
The Strategy pattern is really about having a different implementation that accomplishes (basically) the same thing, so that one implementation can replace the other as the strategy requires. For example, you might have different sorting algorithms in a strategy pattern. The callers to the object does not change based on which strategy is being employed, but regardless of strategy the goal is the same (sort the collection).

The State pattern is about doing different things based on the state, while leaving the caller relieved from the burden of accommodating every possible state. So for example you might have a `getStatus()` method that will return different statuses based on the state of the object, but the caller of the method doesn't have to be coded differently to account for each potential state.

The difference simply lies in that they solve different problems:

- The *State* pattern deals with **what** (state or type) an object is (in) -- it encapsulates state-dependent behavior, whereas
- the *Strategy* pattern deals with **how** an object performs a certain task -- it encapsulates an algorithm.

vTable Concept:



Himanshu Mishra <himanshumishra3112@gmail.com>
 To: Himanshu Mishra <himanshumishra3112@gmail.com>

Fri, May 29, 2015 at 1:30 PM

STL

->Sequence containers

->vector:

- * Dynamic array which grows in one direction
- * Direction (--->)
- * Elements can be randomly accessed
- * Recommended way of traversing is through iterator(faster)
- * Dynamically allocated contiguous block of memory
- * fast insert/remove at the end: O(1)
- * slow insert/remove at the beginning or in the middle: O(n)
- * slow search: O(n)
- * Cache advantage

->deque

- * Direction <--- or --->
- * Random access.
- * fast insert/remove at the both ends
- * Slow insert/remove at the middle: O(n)
- * slow search: O(n)

->list

- * Doubly linked list.
- * Fast insert/remove at any place inside list
- * Slow search(slower than vector)
- * cache misses
- * take more memory
- * No random access

* big advantage: splice()

->forward list
* one direction

->array
* size can not be changed
* two array of integer may have different type

-> **Associative**

->set
->map

-> **Unordered**

->set
->map

Common API:

```
cont.begin()
cont.end()
cont.empty()
cont.size()
cont.swap()
cont.clear()
```

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Tue, Jun 2, 2015 at 5:01 PM

Shared object/library:

```
gcc -c -Wall -Werror -fpic <source file>
gcc -shared -o lib<shared file>.so <object file>.o
gcc -L<path> -Wall -o <executable file> <client source file> -l<shared file>
```

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Fri, Jun 5, 2015 at 12:29 AM

Storage qualifiers in C++:

- i.) Const - This variable means that if the memory is initialised once, it should not be altered by a program.
- ii.) Volatile - This variable means that the value in the memory location can be altered even though nothing in the program code modifies the contents.
- iii.) Mutable - This variable means that a particular member of a structure or class can be altered even if a particular structure variable, class, or class member function is constant.

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Fri, Jun 5, 2015 at 12:40 AM

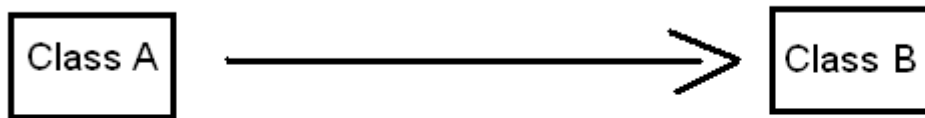
- Constructors can not be virtual
- Destructors can be virtual.

[Quoted text hidden]

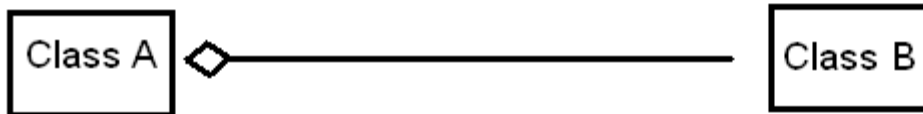
Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Thu, Jun 11, 2015 at 8:29 AM

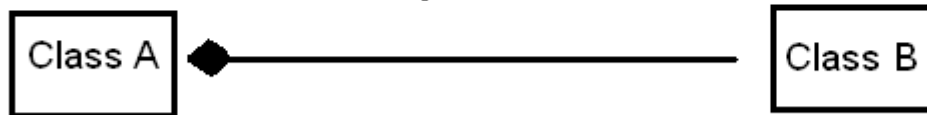
Association



Aggregation



Composition

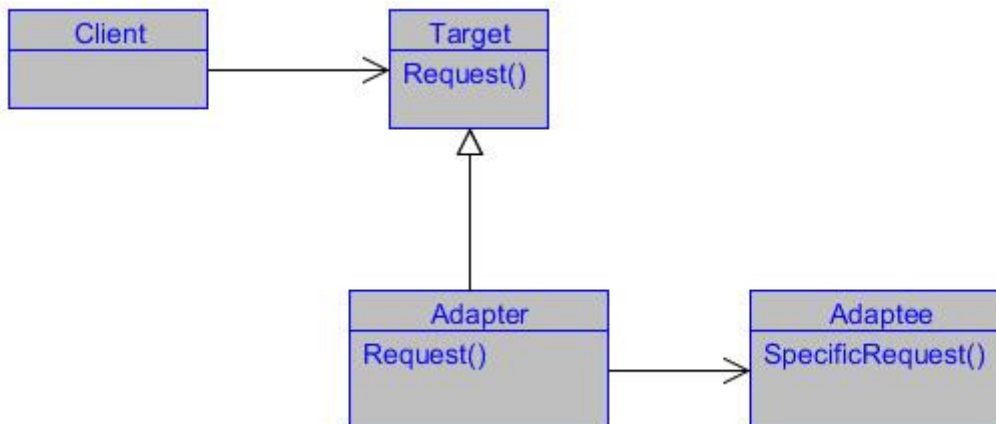


[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Fri, Jul 17, 2015 at 4:15 PM

Adapter Design



- **Target:** Defines the interface that the client uses.
- **Adapter:** It uses (adapts) the `Adaptee`'s interface and exposes the `Target` interface to the client.
- **Adaptee:** This is the object whose interface needs to be adapted.
- **Client:** It uses the `Adaptee` functionality via the adapter interface, i.e., `Target`.

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Fri, Jul 17, 2015 at 4:16 PM

<http://www.codeproject.com/Articles/342082/Understanding-and-Implementing-the-Adapter-Pattern>

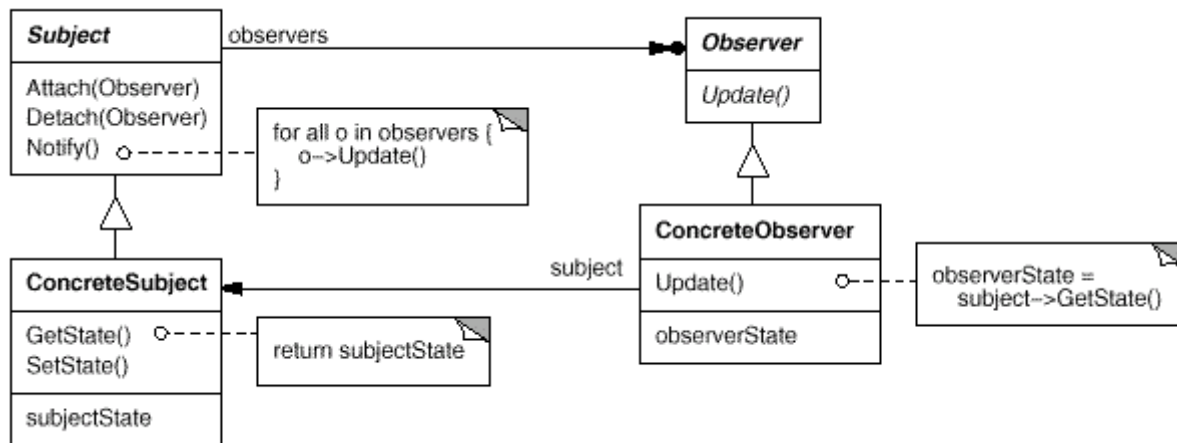
[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Tue, Aug 4, 2015 at 3:48 PM

Observer Design Pattern

Observer Pattern's intent is to define a **one-to-many** dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The **subject** and **observers** define the one-to-many relationship. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the notification, the observers may also be updated with new values.



[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Fri, Sep 18, 2015 at 12:33 AM

"The need for **dynamic_cast** generally arises because you want to perform derived class operation on a derived class object, but you have only a pointer or reference-to-base" said Scott Meyers in his book "Effective C++".

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Wed, Oct 28, 2015 at 12:59 PM

What is Visitor Pattern?

The **"visitor"** design pattern is a way of separating an algorithm from an object structure. The basic idea is that you have a set of element classes that form an object structure. Each of these element classes has an **"accept"** method that takes a visitor object as an argument. The visitor is an interface that has a different **"visit"** method for each element class. The "accept" method of an element class calls back the "visit" method for its class. Separate concrete visitor classes can then be written that perform some particular operations.

You can think of these visit methods as methods not of a single class, but rather methods of a pair of classes: the concrete visitor and the particular element class. Thus the visitor pattern simulates double dispatch in a conventional single dispatch object-oriented language such as Java or C++.

The **Visitor Design Pattern** allows you to decouple the logics and the data structures and while applying the logics to the data structures. With this pattern, you can build classes that focus only on the data structures without knowing the logics that will be applied to the structure. At the same time, you can build classes that concentrate solely on the logics that will be applied to the structure without knowing what the structure looks like. The benefit is that the evolution of the logics and the structures can vary independently.

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Wed, Oct 28, 2015 at 3:57 PM

Builder Pattern

- Builder pattern is useful when you want to build a complex object. Intent of this pattern is to separate the construction of object from its representation.
- Abstract the construction of object and then derived concrete implementations will give respective construction parts.
- The director makes sure that the product (complex object) is properly created using the builder interface and the parts are built with correct order.

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Wed, Oct 28, 2015 at 4:06 PM

```
#include <iostream>
#include <string>
#include <queue>

using namespace std;

// Product
class Meal {
public:
    Meal() {}
    ~Meal() {}

    void setMealItem(string mealItem) { mMeal.push(mealItem); }
    void serveMeal() {
        int i = 1;
        while(!mMeal.empty()) {
            cout << " Serve item " << i++ << ":" << mMeal.front() << endl ;
            mMeal.pop();
        }
    }

private:
    queue <string> mMeal;
};

// Builder
class MealBuilder {
public:
    MealBuilder() {}
    ~MealBuilder() {}

    const Meal& getMeal() { return mMeal; }
    virtual void buildStarter() = 0;
    virtual void buildMainCourse() = 0;
    virtual void buildDessert() = 0;

protected:
```

```
    Meal mMeal;
};

// Director: Makes sure the right sequence of food is prepared and served.
class MultiCuisineCook {
public:
    MultiCuisineCook():mMealBuilder(NULL) {}
    ~MultiCuisineCook() { if (mMealBuilder) delete mMealBuilder;}

    void setMealBuilder(MealBuilder *mealBuilder) {
        if (mMealBuilder) delete mMealBuilder;
        mMealBuilder = mealBuilder;
    }

    const Meal& getMeal() { return mMealBuilder->getMeal(); }

    void createMeal() {
        mMealBuilder->buildStarter();
        mMealBuilder->buildMainCourse();
        mMealBuilder->buildDessert();
    }

private:
    MealBuilder *mMealBuilder;
};

// Concrete Meal Builder 1
class IndianMealBuilder : public MealBuilder {
public:
    IndianMealBuilder() {}
    ~IndianMealBuilder() {}

    void buildStarter() { mMeal.setMealItem("FriedOnion");}
    void buildMainCourse() { mMeal.setMealItem("CheeseCurry");}
    void buildDessert() { mMeal.setMealItem("SweetBalls");}

};

// Concrete Meal Builder 2
class ChineseMealBuilder : public MealBuilder {
public:
    ChineseMealBuilder() {}
    ~ChineseMealBuilder() {}

    void buildStarter() { mMeal.setMealItem("Manchurian");}
    void buildMainCourse() { mMeal.setMealItem("FriedNoodles");}
    void buildDessert() { mMeal.setMealItem("MangoPudding");}

};

// Concrete Meal Builder 3
class MexicanMealBuilder : public MealBuilder {
public:
```

```
MexicanMealBuilder() {}
~MexicanMealBuilder() {}

void buildStarter() { mMeal.setMealItem("ChipsNSalsa");}
void buildMainCourse() { mMeal.setMealItem("RiceTacoBeans");}
void buildDessert() { mMeal.setMealItem("FriedIcecream");}

};

int main()
{
    MultiCuisineCook cook;

    cout << "Build a Chinese Meal!" << endl;
    cook.setMealBuilder(new ChineseMealBuilder());
    cook.createMeal();

    Meal chineseMeal = cook.getMeal();
    chineseMeal.serveMeal();

    cout << "Build a Mexican Meal!" << endl;
    cook.setMealBuilder(new MexicanMealBuilder());
    cook.createMeal();

    Meal mexicanMeal = cook.getMeal();
    mexicanMeal.serveMeal();

    return 0;
}
```

Output:-

```
Build a Chinese Meal!
Serve item 1:Manchurian
Serve item 2:FriedNoodles
Serve item 3:MangoPudding
Build a Mexican Meal!
Serve item 1:ChipsNSalsa
Serve item 2:RiceTacoBeans
Serve item 3:FriedIcecream
```

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Thu, Oct 29, 2015 at 10:10 AM


```

1 #include <iostream>
2 using namespace std;
3
4 namespace mon{
5     typedef enum _month{jan=1000, fab, mar, apr} month;
6 }
7
8 using namespace mon; //Place after namespace definition
9 int fn(month _mon){
10     cout << "Jan : " << _mon << endl;
11 }
12
13 int main(){
14     cout << "size of enum : " << sizeof(mon::month) << endl;
15     fn(mon::jan);
16 }

```

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
 To: Himanshu Mishra <himanshumishra3112@gmail.com>

Thu, Oct 29, 2015 at 10:26 AM

What is template design pattern?

- Template pattern is a behavioral design pattern.
- This has nothing to do with C++ templates as such.
- Template patterns is a common form in object oriented programming. Having an abstract base class (one or more pure virtual functions) is a simple example of template design pattern.
- In the template pattern, parts of program which are well defined like an algorithm are defined as a concrete method in the base class. The specifics of implementation are left to the derived classes by making these methods as abstract in the base class.
- The method which implements the algorithm is also referred as template method and the class which implements this methods as the template class.

Demonstrate the usage of template design pattern

```

#include <iostream>
using namespace std;

// Base class
// Template class
class Account {
public:
    // Abstract Methods
    virtual void Start() = 0;

    virtual void Allow() = 0;

    virtual void End() = 0;

    virtual int MaxLimit() = 0;

    // Template Method
    void Withdraw(int amount) {

```

```
        Start();

        int limit = MaxLimit();
        if ( amount < limit ) {
            Allow();
        }
        else {
            cout << "Not allowed" << endl;
        }

        End();
    }
};
```

// Derived class

```
class AccountNormal : public Account {
public:
    void Start() {
        cout << "Start ..." << endl;
    }

    void Allow() {
        cout << "Allow ..." << endl;
    }

    void End() {
        cout << "End ..." << endl;
    }

    int MaxLimit() {
        return 1000;
    }
};
```

// Derived class

```
class AccountPower : public Account {
public:
    void Start() {
        cout << "Start ..." << endl;
    }

    void Allow() {
        cout << "Allow ..." << endl;
    }

    void End() {
        cout << "End ..." << endl;
    }

    int MaxLimit() {
        return 5000;
    }
};
```

```
int main() {
    AccountPower power;
    power.Withdraw(1500);

    AccountNormal normal;
    normal.Withdraw(1500);
}
```

OUTPUT:-

```
Start ...
Allow ...
End ...

Start ...
Not allowed
End ...
```

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

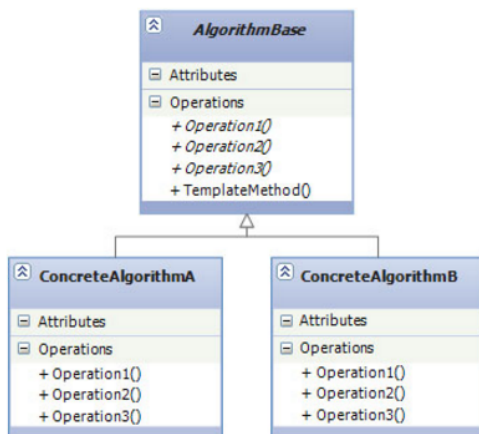
Thu, Oct 29, 2015 at 1:48 PM

Template method pattern

The template method pattern is a design pattern that allows a group of interchangeable, similarly structured, multi-step algorithms to be defined. Each algorithm follows the same series of actions but provides a different implementation of the steps.

This design pattern allows you to change the behavior of a class at run-time. It is almost the same as Strategy design pattern but with a significant difference. While the Strategy design pattern overrides the entire algorithm, Template method allows you to change only some parts of the behavior algorithm using abstract operations (the entire algorithm is separated into several operations).

Structural code example



The UML diagram below describes an implementation of the template method design pattern. This diagram consists of two parts:

- **AlgorithmBase:** defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm. Implements a template method which defines the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in **AbstractClass** or those of other objects.
- **ConcreteAlgorithm:** implements the primitive operations to carry out subclass-specific steps of the algorithm. When a concrete class is called the template method code will be executed from the base class while for each method used inside the template method will be called the implementation from the derived class.

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

[Quoted text hidden]

[Quoted text hidden]

[Quoted text hidden]

```

try
{
    throw myex;
    throw 20;
}
catch (exception& e)
{
    myexception& myexceptionObj = dynamic_cast<myexception&> (e); // Reference
    myexception* myexceptionObj2 = dynamic_cast<myexception*> (&e); // Pointer
    //myexception myexceptionObj3 = dynamic_cast<myexception> (e); // Error: target is not pointer or reference
    cout << "1. " << e.what() << " - Base reference to derived" << '\n';
    cout << "1. " << myexceptionObj.what() << " - Type casted reference to derived" << '\n';
    cout << "1. " << myexceptionObj2->what() << " - Type casted pointer to derived" << '\n';
}
catch (myexception& e)
{
    cout << "2. " << e.what() << '\n';
}
catch (...)
{
    cout << "3. Unknown exception" << '\n';
}
return 0;
}

```

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
 To: Himanshu Mishra <himanshumishra3112@gmail.com>

Mon, Nov 16, 2015 at 4:52 PM

Setting Linux Environment for Personal Practice:

1. Create a global(common) make file for compilation rule, boost libraries etc.
2. Set a environment variable pointing to this file and add this in .bashrc file.
3. use this common make to compile c++ code using advanced libraries.

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
 To: Himanshu Mishra <himanshumishra3112@gmail.com>

Sat, Nov 21, 2015 at 7:41 PM

Valgrind:
 =====

Tool = Memcheck

This tool can detect the following memory related problems :

- Use of uninitialized memory
- Reading/writing memory after it has been freed
- Reading/writing off the end of malloc'd blocks
- Memory leaks
- Mismatched use of malloc/new/new[] vs free/delete/delete[]
- Doubly freed memory

```
valgrind --tool=memcheck --leak-check=full ./val
```

Key checks:

- Invalid read
- Invalid write
- LEAK SUMMARY
- HEAP SUMMARY
- Mismatched free
- Invalid free
- Error
- Uninitialized

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Sat, Nov 21, 2015 at 8:38 PM

<http://stackoverflow.com/questions/4172722/what-is-the-rule-of-three>

<http://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom>

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Mon, Jan 11, 2016 at 5:22 PM

Amadeus C++ class.

[Quoted text hidden]

 **AdvancedCPP_Workshop.7z**
3K

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Tue, Jan 12, 2016 at 5:02 PM

time size nm objdump strace ltrace proc/file (man prock)

shaik4consulting@gmail.com

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Tue, Apr 12, 2016 at 11:26 AM

<http://easycplusplus.com/cpp/cpptutorial22a.html>

by John Kopp

Support this site at no cost to you

Introduction

Welcome to EasyCPlusPlus.com's free tutorial on C++ programming. This lesson deals with two important keywords, [const](#) and [mutable](#), and their use with class objects. Const provides a way to declare that an object is not modifiable.

It can also be used with class methods to indicate to they will not modify their objects. The use of "const" can reduce bugs within your code by allowing the compiler to catch unintended modification of objects that should remain constant. The keyword mutable provides a way to allow modifications of a particular data members of a constant objects. By the way, I love tabloid style headlines, which is why this lesson is titled "Mutable Members" rather than something dull like "Const and Mutable"

Const

In an earlier lesson, the keyword const was used to declare that an object of built-in type, that is, an ordinary variable, was a constant. Attempts to modify a "const" result in a compilation error.

```
const int limit;
limit = 25; // Results in compilation error
```

Assigning to a const variable is not permitted. It must be initialized to provide a value.

```
const int limit = 25;
```

This provided a way to have a constant and to be sure that the constant was not modified unintentionally. C++ also allows the declaration of const class objects.

```
// Class Definition
class Employee {
public:
    string getName(){return _name;}
    void setName(string name) {_name = name;}
    string getId(){return _id;}
    void setId(string id) {_id = id;}
private:
    string _name;
    string _id;
};
```

```
const Employee john;
```

This declares the object john of class Employee to be constant. But there are problems with this code. First, since the object is const, we need a way to initialize it. Its members cannot be assigned either directly or indirectly via methods. The compiler's default constructor is insufficient, so we must define constructors that can initialize all the data members. A default constructor that initializes all members is required. Other constructors may be written. Second, C++ allows methods to be declared as const. By declaring a method "const" we are in effect stating that the method cannot change its object. *Only methods declared const can be called be a const object.* This has real benefit. The compiler can check that methods declared const do not modify the object. Attempts to modify an object within a const method will be flagged by the compiler. Since a const object may invoke only const methods, it will not be modified unintentionally. *Objects that are not const can invoke both const and non-const methods.* Which methods should be declared as const? Certainly, any method that is intended to simple return the value of a data member should be. Depending upon their purpose, the const keyword may be appropriate for other methods as well. Let's correct the Employee class to allow its proper use with const Employee objects and see its use in a simple program.

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
 To: Himanshu Mishra <himanshumishra3112@gmail.com>

Wed, Apr 27, 2016 at 10:34 AM

Operator "<<" overloading - For Debugging purpose

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

class Test
{
    private:
        int i;
        string s;
    public:
        Test(int _i, string _s): i(_i), s(_s)
        {}
        friend ostream &operator<<(ostream &out, const Test& t)    //output
        {
            out<<"i: "<<t.i<<"\n";
            out<<"s: "<<t.s<<"\n";
            return out;
        }
};

int main()
{
    Test t1(10,"himanshu");
    cout << t1 << endl;
    Test t2(20,"himanshu2");
    cout << t2 << endl;
}
```


[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Fri, May 6, 2016 at 12:50 PM

GDB useful reference DOC.

[Quoted text hidden]

 **gdb_coredump1.pdf**
20K

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Thu, May 26, 2016 at 12:48 PM

=====

Let me explain this way:

Your dad has a bike and ur his son want to use his bike but it is outdated and u want to modify it the way you want to and ride it...that is, ur dad(class) has a function called Bike and ur using the same Bike(function) but modified/changed and ur using it(overriding). Same bike but now it is changed.... and when we say son it is ur modified bike that shud come into picture ..which has overriden ur dad's old bike

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Thu, May 4, 2017 at 12:53 PM

Ctag and Taglist tools for vim

<http://www.thegeekstuff.com/2009/04/ctags-taglist-vi-vim-editor-as-source-code-browser>

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Sat, May 6, 2017 at 9:47 PM

<https://packagecontrol.io/packages/CTags>

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Sat, May 6, 2017 at 10:34 PM

GDB Excellent material:

<https://www.youtube.com/watch?v=-n9Fkq1e6sg>

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Tue, May 16, 2017 at 5:07 PM

tmux install from source code

[Quoted text hidden]

 **3106801-c3382b8bda16706f530614329b0af7d0e90075aa.zip**
2K

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Thu, May 25, 2017 at 2:57 PM

VIM tips and tricks

<https://www.cs.swarthmore.edu/help/vim/vim7.html>

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Wed, May 31, 2017 at 3:09 PM

Switch to TUI mode: Ctrl+x+a

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Fri, Jun 16, 2017 at 11:00 AM

```
gcc -H -fsyntax-only <C/C++ file>
```

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>

Sun, Jul 23, 2017 at 11:37 PM

To: Himanshu Mishra <himanshumishra3112@gmail.com>

ObjectCount.cpp

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

template <class T>
class ObjectCount
{
    static int count;
protected:
    ObjectCount() {
        count++;
    }
public:
    void static showCount() {
        cout << count << endl;
    }
};

template <class T>
int ObjectCount<T>::count = 0;

class Employee : public ObjectCount<Employee>
{
public:
    Employee(){}
    Employee(const Employee & emp) {}
};

class Manager : public ObjectCount<Manager>
{
public:
    Manager(){}
    Manager(const Manager & man) {}
};

class Director : public ObjectCount<Director>
{
public:
    Director(){}
    Director(const Director & dir) {}
};

int main()
{
    Employee e[10];
    cout << "Employee's count: ";ObjectCount<Employee>::showCount();
    Manager m[5];
    cout << "Manager's count: ";ObjectCount<Manager>::showCount();
    Director d1;
    cout << "Director's count: ";ObjectCount<Director>::showCount();
}
```

Output:

Employee's count: 10
Manager's count: 5
Director's count: 1

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: Himanshu Mishra <himanshumishra3112@gmail.com>

Thu, Aug 10, 2017 at 4:04 PM

The PIMPL idiom

<http://www.cppsamples.com/common-tasks/pimpl.html>

[Quoted text hidden]

Himanshu Mishra <himanshumishra3112@gmail.com>
To: himanshumishra3112 <himanshumishra3112@gmail.com>

Fri, Aug 11, 2017 at 1:02 PM

Sorting algorithm comparison

<https://www.toptal.com/developers/sorting-algorithms/>

[Quoted text hidden]