Table of Contents

[Explicit Functions 2](#_Toc18359619)

[Design Pattern 2](#_Toc18359620)

[Command design pattern 2](#_Toc18359621)

[Const Keyword in C++ 4](#_Toc18359622)

[PIMPL Idiom (Pointer to Implementation) 4](#_Toc18359623)

[Smart Pointers 6](#_Toc18359624)

[auto and decltype 7](#_Toc18359625)

[Map Compare function 8](#_Toc18359626)

[To Do’s 9](#_Toc18359627)

# Explicit Functions

Explicit function specifier controls unwanted implicit type conversions. It can only be used in declarations of constructors within a class declaration. For example, except for the default constructor, the constructors in the following class are conversion constructors

class A

{ public:

A();

A(int);

A(const char\*, int = 0);

};

The following declarations are legal.

A c = 1;

A d = "Venditti";

If you declare the constructor of the class with the explicit keyword, the previous declarations would be illegal.

class A

{ public:

explicit A();

explicit A(int);

explicit A(const char\*, int = 0);

};

statements are legal:

A a1;

A a2 = A(1);

A a3(1);

A a4 = A("Venditti");

A\* p = new A(1);

A a5 = (A)1;

A a6 = static\_cast<A>(1);

# Design Pattern

## Command design pattern

<https://www.oodesign.com/command-pattern.html>

<https://www.geeksforgeeks.org/command-pattern/>

1. Create a class that encapsulates some number of the following:
   * a "receiver" object
   * the method to invoke
   * the arguments to pass
2. Instantiate an object for each "callback"
3. Pass each object to its future "sender"
4. When the sender is ready to callback to the receiver, it calls execute()
5. **#include <iostream> #include <string> using namespace std;**
6. **class** **Person**;
7. **class** **Command**
8. {
9. *// 1. Create a class that encapsulates an object and a member function*
10. *// a pointer to a member function (the attribute's name is "method")*
11. Person \*object; *//*
12. **void**(Person:: \*method)();
13. **public**:
14. Command(Person \*obj = 0, **void**(Person:: \*meth)() = 0)
15. {
16. object = obj; *// the argument's name is "meth"*
17. method = meth;
18. }
19. **void** execute()
20. {
21. (object-> \*method)(); *// invoke the method on the object*
22. }
23. };
24. **class** **Person**
25. {
26. string name;
27. *// cmd is a "black box", it is a method invocation*
28. *// promoted to "full object status"*
29. Command cmd;
30. **public**:
31. Person(string n, Command c): cmd(c)
32. {
33. name = n;
34. }
35. **void** talk()
36. {
37. *// "this" is the sender, cmd has the receiver*
38. cout << name << " is talking" << endl;
39. cmd.execute(); *// ask the "black box" to callback the receiver*
40. }
41. **void** passOn()
42. {
43. cout << name << " is passing on" << endl;
45. *// 4. When the sender is ready to callback to the receiver,*
46. *// it calls execute()*
47. cmd.execute();
48. }
49. **void** gossip()
50. {
51. cout << name << " is gossiping" << endl;
52. cmd.execute();
53. }
54. **void** listen()
55. {
56. cout << name << " is listening" << endl;
57. }
58. };
59. **int** **main**()
60. {
61. *// Fred will "execute" Barney which will result in a call to passOn()*
62. *// Barney will "execute" Betty which will result in a call to gossip()*
63. *// Betty will "execute" Wilma which will result in a call to listen()*
64. Person wilma("Wilma", Command());
65. *// 2. Instantiate an object for each "callback"*
66. *// 3. Pass each object to its future "sender"*
67. Person betty("Betty", Command(&wilma, &Person::listen));
68. Person barney("Barney", Command(&betty, &Person::gossip));
69. Person fred("Fred", Command(&barney, &Person::passOn));
70. fred.talk();
71. }

#### Output

Fred is talking

Barney is passing on

Betty is gossiping

Wilma is listening

# Const Keyword in C++

<https://www.codesdope.com/cpp-const-keyword/>

<https://www.studytonight.com/cpp/const-keyword.php>

# PIMPL Idiom (Pointer to Implementation)

Example-

class X

{

private:

C c;

D d;

} ;

could be changed to:

class X

{

private:

struct XImpl;

XImpl\* pImpl;

};

and, in the CPP, the definition:

struct X::XImpl

{

C c;

D d;

};

Example 2

// my\_class.h

class my\_class {

// ... all public and protected stuff goes here ...

private:

class impl; unique\_ptr<impl> pimpl; // opaque type here

};

// my\_class.cpp

class my\_class::impl { // defined privately here

// ... all private data and functions: all of these

// can now change without recompiling callers ...

};

my\_class::my\_class(): pimpl( new impl )

{

// ... set impl values ...

}

**Own interpretation** -

* Dependencies need not to recompile their code as they have used the pointer of the class. We can do any changes in the definition of pointer in class
* Also, it helps in data hiding

**Reasons for using the PIMPL idiom: (stackoverflow)**

**Binary compatibility**

When you're developing a library, you can add/modify fields to XImpl without breaking the binary compatibility with your client (which would mean crashes!). Since the binary layout of X class doesn't change when you add new fields to Ximpl class, it is safe to add new functionality to the library in minor versions updates.

Of course, you can also add new public/private non-virtual methods to X/XImpl without breaking the binary compatibility, but that's on par with the standard header/implementation technique.

**Data hiding**

If you're developing a library, especially a proprietary one, it might be desirable not to disclose what other libraries / implementation techniques were used to implement the public interface of your library. Either because of Intellectual Property issues, or because you believe that users might be tempted to take dangerous assumptions about the implementation or just break the encapsulation by using terrible casting tricks. PIMPL solves/mitigates that.

**Compilation time**

Compilation time is decreased, since only the source (implementation) file of X needs to be rebuilt when you add/remove fields and/or methods to the XImpl class (which maps to adding private fields/methods in the standard technique). In practice, it's a common operation.

With the standard header/implementation technique (without PIMPL), when you add a new field to X, every client that ever allocates X (either on stack, or on heap) needs to be recompiled, because it must adjust the size of the allocation. Well, every client that doesn't ever allocate X *also* need to be recompiled, but it's just overhead (the resulting code on the client side will be the same).

What is more, with the standard header/implementation separation XClient1.cpp needs to be recompiled even when a private method X::foo() was added to X and X.h changed, even though XClient1.cpp can't possibly call this method for encapsulation reasons! Like above, it's pure overhead and is related with how real-life C++ build systems work.

Of course, recompilation is not needed when you just modify the implementation of the methods (because you don't touch the header), but that's on par with the standard header/implementation technique

# Smart Pointers

[Smart pointer](http://en.wikipedia.org/wiki/Smart_pointer) is a wrapper class over a pointer with operator like \* and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can’t like automatic destruction (yes, we don’t have to explicitly use delete), reference counting and more

// A generic smart pointer class

template <class T>

class SmartPtr

{

   T \*ptr;  // Actual pointer

public:

   // Constructor

   explicit SmartPtr(T \*p = NULL) { ptr = p; }

   // Destructor

   ~SmartPtr() { delete(ptr); }

   // Overloading dereferencing operator

   T & operator \* () {return \*ptr;}

   // Overloading arrow operator so that members of T can be accessed

   // like a pointer (useful if T represents a class or struct or

   // union type)

   T \* operator -> () {return ptr;}

};

int main ()

{

    SmartPtr<int> smpt(new int());

    \*smpt = 20; // smpt = \*ptr (returning the dereferencing value) = T & (returning the \*ptr and not the copy of \*ptr)

    cout << \*smpt;

    return 0;

}

The dereference operator (\*) overload works like any other operator overload. If you want to be able to modify the dereferenced value, you need to return a non-const reference. This way \*smpt = value will actually modify the value pointed to by smpt.ptr and not a temporary value generated by the compiler.

\*smpt = value (need to modify the dereference value of ptr so in place of smpt it will return the address of \*ptr)

The structure dereference operator (->) overload is a special case of operator overloading. The operator is actually invoked in a loop until a real pointer is returned, and then that real pointer is dereferenced. I guess this was just the only way they could think of to implement it and it turned out a bit hackish.

smpt->member = 10; // interpreted as (smpt.operator->())->member

# auto and decltype

auto keyword specifies that the type of the variable that is being declared will be automatically deducted from its initializer. In case of functions, if their return type is auto then that will be evaluated by return type expression at runtime

#include <bits/stdc++.h>

using namespace std;

int main()

{

    auto x = 4;

    auto y = 3.37;

    auto ptr = &x;

    cout << typeid(x).name() << endl

         << typeid(y).name() << endl

         << typeid(ptr).name() << endl;

    return 0;

}

Outputs:  
i   
d  
Pi

**Note:** auto becomes int type if even an integer reference is assigned to it. To make it reference type, we use auto &

// function that returns a 'reference to int' type  
int& fun() { }

// m will default to int type instead of int& type  
auto m = fun();

// n will be of int& type because of use of extra & with auto keyword  
auto& n = fun();

**decltype**

It inspects the declared type of an entity or the type of an expression. Auto lets you declare a variable with particular type whereas decltype lets you extract the type from the variable.

#include <bits/stdc++.h>

using namespace std;

// A generic function which finds minimum of two values

// return type is type of variable which is minimum

template <class A, class B>

auto findMin(A a, B b) -> decltype(a < b ? a : b)

{

    return (a < b) ? a : b;

}

// driver function to test various inference

int main()

{

    // This call returns 3.44 of double type

    cout << findMin(4, 3.44) << endl;

    // This call returns 3 of double type

    cout << findMin(5.4, 3) << endl;

    return 0;

}

# Map Compare function

// constructing maps

#include <iostream>

#include <map>

bool fncomp (char lhs, char rhs) {return lhs<rhs;}

struct classcomp {

bool operator() (const char& lhs, const char& rhs) const

{return lhs<rhs;}

};

int main ()

{

std::map<char,int> first;

first['a']=10;

first['b']=30;

first['c']=50;

first['d']=70;

std::map<char,int> second (first.begin(),first.end());

std::map<char,int> third (second);

std::map<char,int,classcomp> fourth; // class as Compare

bool(\*fn\_pt)(char,char) = fncomp;

std::map<char,int,bool(\*)(char,char)> fifth (fn\_pt); // function pointer as Compare

return 0;

}

# Template Inheritance

<https://blog.feabhas.com/2014/06/template-inheritance/>

# To Do’s

* Make file (GNU Make) and make command
* Operator overloading
* Smart pointers
* Qsc compiler
* Static and shared library
* Typecasting operator
* Csh and sh shell scripiting
* Do research about languages to learn and why?