**Functors, Functor Classes, Functions, etc.**

Functor class have state where as function don’t.

class IncrementFunctor {

public:

IncrementFunctor() :num(0){}

void operator() (int &x){

x=num+x;

num++;

}

int num;

};

std::vector<int>v = { 1,2,3,4,5,6,7,8,9 };

std::for\_each(v.begin(), v.end(), IncrementFunctor());

But no ways can perform same with the function since it don’t have the state.

**Item 38. Design functor classes for pass-by-value.**

*Note: functors class not functors argument.*

**Item 39. Make predicates pure functions.**

* A ***predicate***is a function that returns bool . Predicates are widely used in the STL. The comparison functions for the standard associative containers are predicates, and predicate functions are commonly passed as parameters to algorithms like find\_if and the various sorting algorithms.
* A ***pure function***is a function whose return value depends only on its parameters. If f is a pure function and x and y are objects, the return value of f(x, y) can change only if the value of x or y changes.

Note: If a pure function consulted data that might change between calls, invoking the function at different times with the same parameters might yield different results, and that would be contrary to the definition of a pure function.

* A ***predicate class***is a functor class whose operator() function is a predicate, i.e., its operator() returns true or false. The STL expects a predicate, it will accept either a real predicate or an object of a predicate class.

Function objects are always passed by value. So, function objects that are predicate always should behave well when they are copied.

Algorithms may make copies of functors and hold on to them a while before using them, and some algorithm implementations take advantage of this freedom. A critical observation is that ***predicate functions must be pure functions.***

Before Proceed further, check below implementation:

***find\_if()***

Returns an iterator to the first element in the range [first,last) for which pred returns true. If no such element is found, the function returns last.

template<class InputIterator, class UnaryPredicate>

InputIterator find\_if (InputIterator first, InputIterator last, UnaryPredicate pred)

{

while (first!=last) {

if (pred(\*first)) return first;

++first;

}

return last;

}

Consider the below bad predicate class. Regardless of the arguments that are passed, it returns true exactly once: the third time it is called. The rest of the time it returns false.

class BadPredicate : public unary\_function<int, bool> {

public:

BadPredicate() : timesCalled(0) {}

bool operator()(const Widget& ) {

return ++timesCalled == 3;

}

private:

size\_t timesCalled;

};

// Suppose we use this class to eliminate the third element from vector

vector<int> vw; // create vector and put some Widgets into it

vw.erase(remove\_iff(vw.begin(),// eliminate the third Widget;

vw.end(), BadPredicate()), vw.end())

And consider below in our own remove\_iff() implementation:

template <typename Fwdlterator, typename Predicate>

Fwdlterator remove\_iff(Fwdlterator begin, Fwdlterator end, Predicate p)

{

begin = find\_if(begin, end, p);

if (begin == end) return begin;

else {

Fwdlterator next = begin;

return remove\_copy\_if(++next, end, begin, p);

}

}

vector<int> v{ 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

v.erase(remove\_iff(v.begin(), v.end(), BadPredicate ()),v.end());

for\_each(v.begin(), v.end(), displayEle);

Output is: 

It will eliminate not just the third element from v, it will also eliminate the sixth! But Why?

Here we can ignore the implementation of remove\_iff method. The predicate p is passed first to find\_if, then later to remove\_copy\_if. In both cases, p is passed by value —

is *copied* — into those algorithms.

The initial call to remove\_iff creates an anonymous BadPredicate object, one with its internal timesCalled member set to zero. This object (known as p inside *remove\_iff)* is then copied into find\_if, so find\_if also receives a BadPredicate object with a timesCalled

value of 0.

find\_if "calls" that object until it returns true, so it calls it three times, find\_if then returns control to remove\_iff. Remove\_iff continues to execute and ultimately calls remove\_copy\_if, passing as a predicate another copies of p. But p's timesCalled member is still 0!

Find\_if never called p. it called only a copy of p. As a result, the third time remove\_copy\_iff calls its predicate, it, too, will return true. And that's why remove\_iff will ultimately remove two int values from v instead of just one.

We Can avoid this problem by rewriting predicate like this:

class BadPredicate : public unary\_function<int, bool> {

public:

BadPredicate() : timesCalled(0) {}

bool operator()(const Widget& )const {

return ++timesCalled == 3; //Error, now method is const.

}

private:

size\_t timesCalled;

};

Note: But only const operator() is not enough. Even const member functions may access mutable data members, non-const local static objects, non-const class static objects, non-const objects at namespace scope, and non-const global objects. A well-designed predicate class ensures that its operator() functions are independent of those kinds of objects, too.

Declaring operator() const in predicate classes is *necessary* for correct behavior, but it's not *sufficient.* A well-behaved operator() is certainly const, but it's more than that. It's also a pure function.

We now understand that operator() functions in predicate classes should be pure functions, so this restriction extends to predicate functions, too. This function is as bad a predicate as the objects generated from the Bad-Predicate class:

bool anotherBadPredicate(const Widget&, const Widget&){

static int timesCalled = 0; // No! No! No! No! No! No! No!

return ++timesCalled == 3; // Predicates should be pure functions,

} // and pure functions have no state.

**Item 40. Make functor classes adaptable.**

Before proceeds with next explanation need to know blow thing:

* *Unary/Binary function object base class*
* *pointer\_to\_unary\_function/binary\_function*
* *Ptr\_func*
* *Unary\_negate/Binary\_negate*
* *adapters (not1, not2, bind1st, and bind2nd)*
* ***Unary/Binary function object base class***

This is a base class for standard unary/binary function objects.

Function objects are instances of a class with member function operator() defined.

And we can call operator() using function object like any other method call.

*unary\_function* is just a base class, from which specific unary function objects are derived. It has no operator()member defined. It simply has two public data members that are typedefs of the template parameters. It is defined as:

template <class Arg, class Result>

struct unary\_function {

typedef Arg argument\_type;

typedef Result result\_type;

};

// unary\_function example

#include <functional> // std::unary\_function

struct IsOdd : public std::unary\_function<int,bool> {

bool operator() (int number) {return (number%2!=0);}

};

int main () {

IsOdd isOddObject;

IsOdd::argument\_type input;

IsOdd::result\_type result;

std::cout << "Please enter a number: ";

std::cin >> input;

result = isOddObject (input);

}

* ***pointer\_to\_unary\_function/binary\_function***

Generates a unary function object class from a pointer to a function that takes a single argument of type Arg and returns a value of type Result.  
  
pointer\_to\_unary\_function is generally used as a type. The function [ptr\_fun](http://www.cplusplus.com/ptr_fun) can be used to directly construct an object of this type. This class is derived from [unary\_function](http://www.cplusplus.com/unary_function) and is typically defined as:

template <class Arg, class Result>

class pointer\_to\_unary\_function : public unary\_function <Arg,Result>{

protected:

Result(\*pfunc)(Arg);

public:

explicit pointer\_to\_unary\_function ( Result (\*f)(Arg) ) : pfunc (f) {}

Result operator() (Arg x) const { return pfunc(x); }

};

#include <functional>

#include <algorithm>

#include <cmath>

using namespace std;

int main () {

pointer\_to\_unary\_function <double,double> LogObject (log);

double numbers[] = {10.0, 20.0, 40.0, 80.0, 160.0};

double logs[5];

transform (numbers, numbers+5, logs, LogObject);

for (int i=0; i<5; i++)

cout << logs[i] << " ";

cout << endl;

return 0;

}

* ***Ptr\_func***

It Convert function pointer to function object (having member function operator()) and Returns a function object that encapsulates function f. Several standard [algorithms](http://www.cplusplus.com/algorithm)

and [adaptors](http://www.cplusplus.com/functional) are designed to be used with function objects. It is defined as:

template <class Arg, class Result>

pointer\_to\_unary\_function<Arg,Result> ptr\_fun (Result (\*f)(Arg)){

return pointer\_to\_unary\_function<Arg,Result>(f);

}

template <class Arg1, class Arg2, class Result>

pointer\_to\_binary\_function<Arg1,Arg2,Result> ptr\_fun (Result (\*f)(Arg1,Arg2)){

return pointer\_to\_binary\_function<Arg1,Arg2,Result>(f);

}

Return Value: A function object equivalent to f (function pointer).

int main () {

double numbers[] = {10.0, 20.0, 40.0, 80.0, 160.0};

double logs[5];

transform (numbers, numbers+5, logs, ptr\_fun<double,double>(log));

for (int i=0; i<5; i++)

cout << logs[i] << " ";

cout << endl;

return 0;

}

* ***Unary\_negate/Binary\_negate***

Negate unary function object class. Unary function object class whose call returns the opposite of another unary function, passed to its constructor. Object of type unary\_negate are generally constructed using function [not1](http://www.cplusplus.com/not1).

template <class Predicate> class unary\_negate

: public unary\_function <typename Predicate::argument\_type,bool>{

protected:

Predicate fn\_;

public:

explicit unary\_negate (const Predicate& pred) : fn\_ (pred) {}

bool operator() (const typename Predicate::argument\_type& x) const {return !fn\_(x);}

};

struct IsOdd\_class {

bool operator() (const int& x) const {return x%2==1;}

typedef int argument\_type;

} IsOdd\_object;

int main () {

std::unary\_negate<IsOdd\_class> IsEven\_object (IsOdd\_object);

int values[] = {1,2,3,4,5};

int cx;

cx = std::count\_if ( values, values+5, IsEven\_object );

std::cout << "There are " << cx << " elements with even values.\n";

return 0;

}

**Binary\_negate**: Negate binary function object class. Binary function object class whose call returns the opposite of another binary function, passed to its constructor. Object of type binary\_negate are generally constructed using function [not2](http://www.cplusplus.com/not2).

template <class Predicate> class binary\_negate : public binary\_function <typename Predicate::first\_argument\_type, typename Predicate::second\_argument\_type, bool>

{

protected:

Predicate fn\_;

public:

explicit binary\_negate ( const Predicate& pred ) : fn\_ (pred) {}

bool operator() (const typename Predicate::first\_argument\_type& x,

const typename Predicate::second\_argument\_type& y) const {

return !fn\_(x,y);

}

};

int main () {

std::equal\_to<int> equality;

std::binary\_negate < std::equal\_to<int> > nonequality (equality);

int foo[] = {10,20,30,40,50};

int bar[] = {0,15,30,45,60};

std::pair<int\*,int\*> firstmatch,firstmismatch;

firstmismatch = std::mismatch (foo,foo+5,bar,equality);

firstmatch = std::mismatch (foo,foo+5,bar,nonequality);

std::cout << "First mismatch in bar is " << \*firstmismatch.second << "\n";

std::cout << "First match in bar is " << \*firstmatch.second << "\n";

return 0;

}

Output:

|  |
| --- |
| First mismatch in bar is 0  First match in bar is 30 |

* ***adapters (not1, not2, bind1st, and bind2nd)***

**not1:** Return negation of unary function object

template <class Predicate>

unary\_negate<Predicate> not1 (const Predicate& pred){

return unary\_negate<Predicate>(pred);

}

Constructs a unary function object (of a [unary\_negate](http://www.cplusplus.com/unary_negate) type) that returns the opposite of *pred* (as returned by operator !).

struct IsOdd {

bool operator() (const int& x) const {return x%2==1;}

typedef int argument\_type;

};

int main () {

int values[] = {1,2,3,4,5};

int cx = std::count\_if (values, values+5, std::not1(IsOdd()));

std::cout << "There are " << cx << " elements with even values.\n";

return 0;

}

**Not2:** Return negation of binary function object

Constructs a binary function object (of a [binary\_negate](http://www.cplusplus.com/binary_negate) type) that returns the opposite of *pred* (as returned by operator !). A binary function object with the opposite behavior of pred.

template <class Predicate>

binary\_negate<Predicate> not2 (const Predicate& pred){

return binary\_negate<Predicate>(pred);

}

int main () {

int foo[] = {10,20,30,40,50};

int bar[] = {0,15,30,45,60};

std::pair<int\*,int\*> firstmatch,firstmismatch;

firstmismatch = std::mismatch (foo, foo+5, bar, std::equal\_to<int>());

firstmatch = std::mismatch (foo, foo+5, bar, std::not2(std::equal\_to<int>()));

std::cout << "First mismatch in bar is " << \*firstmismatch.second << '\n';

std::cout << "First match in bar is " << \*firstmatch.second << '\n';

return 0;

}

----------------------------------------------------------------------------------------------

Suppose We have a list of Widget\* pointers and a function to determine whether such a pointer identifies a Widget that is interesting:

list<Widget\*> widgetPtrs;

bool islnteresting(const Widget \*pw);

list<Widget\*>::iterator i = find\_if(widgetPtrs.begin(), widgetPtrs.end(), islnteresting);

if ( i != widgetPtrs.end()) {

// process the first interesting pointer-to-widget

}

Not if we want to find out first Widget pointer which is not interesting: not(islnteresting).

However below line of code not will compile.

list<Widget\*>::iterator i=find\_if(widgetPtrs.begin(), widgetPtrs.end(), not1(islnteresting))

Instead of that need to apply this:

i = find\_if(widgetPtrs.begin(), widgetPtrs.end(), not1(**ptr\_func**( islnteresting))); //fine

Consider the below example:

struct IsOdd{

bool operator() (const int& x) const { return x % 2 == 1; }

};

int values[] = { 1,2,3,4,5 };

int cx = std::count\_if(values, values + 5, std::not1(IsOdd()));

std::cout << "There are " << cx << " elements with even values.\n";

Above example will not work. If we change the definition of IsOdd() like below then will start working:

struct IsOdd:public unary\_function<const int& ,bool > {

bool operator() (const int& x) const { return x % 2 == 1; }

};

OR,

struct IsOdd {

bool operator() (const int& x) const { return x % 2 == 1; }

typedef int argument\_type;

};

Why?

Not1 demands the typedefs, which is absent in IsOdd() method. Either we extend the interface unary\_function or define typedef both will work. Similarly, islnteresting lacks the typedefs that not1 demands.

Other ways we can do one thing applay ptr\_fun on IsOdd() method, which convert normal function to slandered function object (implemented with unary\_function or binary\_function).

Note: Each of the four standard function adapters (not1, not2, bind1st, and bind2nd) requires the existence of certain typedefs. So, either we write the correct function object with all required typedef or convert any predicate (function which returns bool) to standard predicate class (function object having operator () and well defined typedef like Arg and Result) using ptr\_fun.

Function objects that provide the necessary typedefs are said to be *adaptable,* while function objects lacking these typedefs are not adaptable. Adaptable function objects can be used in more contexts than can function objects that are not adaptable, so we should make our function objects adaptable whenever we can.

So, what are those typedefs?

Ans: argument\_type, first\_argument\_type, second\_argument\_type, and result\_type e.t.c.

Different kinds of functor classes are expected to provide different subsets of these names

which not easy to remember.

Any ways we don't need to know anything about these typedefs. That's because the conventional way to provide them is to inherit them from a base class, or. more precisely, a base struct. For functor classes whose operator() takes one argument, the struct to inherit from is std::unary\_function. For functor classes whose operator() takes two arguments, the struct to inherit from is std::binary\_function.

unary\_function and binary\_function are templates, so we can't inherit from them directly. Instead that we can inherit from structs they generate, and that requires to specify some type arguments. Here are a couple of examples:

template<typename T>

class MeetsThreshold: public std::unary\_function<Widget, bool>{

private:

const T threshold;

public:

MeetsThreshold(const T& threshold);

bool operator()(const Widget&) const;

…

};

struct WidgetNameCompare: std::binary\_function<Widget, Widget, bool>{

bool operator()(const Widget& lhs, const Widget& rhs) const;

};

In above example we can see how argument type and return type passed in unary and binary functions. Also we can notice, MeetsThreshold is a class, while WidgetNameCompare is

a struct. MeetsThreshold has internal state (its threshold data member), and a class is

the logical way to encapsulate such information. WidgetNameCompare has no state,

hence no need to make anything private. Declaration of such functors as classes or structs is purely a matter of personal style.

Look again at WidgetNameCompare:

struct WidgetNameCompare: std::binary\_function<Widget, Widget, bool> {

bool operator()(cost Widget& lhs, const Widget& rhs) const;

}

Even though operator's arguments are of type const Widget&, the type passed to binary\_function is Widget. In general, non-pointer types passed to unary\_function or binary\_function have consts and references stripped off.

The rules change when operator() takes pointer parameters. Here's a struct analogous to WidgetNameCompare, but this one works with Widget\* pointers:

struct PtrWidgetNameCompare: std::binary\_function<const Widget\*, const Widget\*, bool> {

bool operator()(const Widget\* lhs, const Widget\* rhs) const;

}

Here, the types passed to binary\_function are the *same* as the types taken by operator(). The general rule for functor classes taking or returning pointers is to pass to unary\_function or binary\_function whatever types operator() takes or returns.

**Note: Functor class inherited with binary or unary base class makes “class adoptable”.**