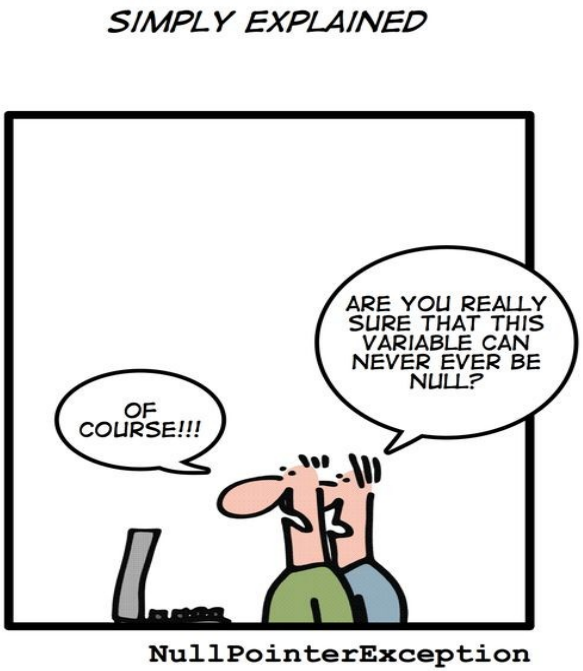
Functors, std::functions



Krishna Kumar

Functors

- A functor is pretty much just a class or a struct which defines the operator(). That lets you create objects which "look like" a function.
- One is that unlike regular functions, they can contain state.

```
class Multiplier { //functor classs
double y;
   public:
    Multiplier(double y): y{y}{};
    double operator()(double x) { return x * y;}
}
Multiplier doubler{2};  // create an instance of the functor class
double x = doubler(5); // call it
```

Functors (cont...)

Multiplier tripler{3}; // create an instance of the functor class

double x = tripler(5); // call it, multiplies given value by 3

std::vector<int> in{1, 2, 3, 4, 5};

// Pass a functor to std::transform, which calls the functor on every element

// in the input sequence, and stores the result to the output sequence

std::transform(in.begin(), in.end(), in.begin(), Multiplier(5));

Pros and Cons

- **instead of plain function**:
  - Pros:
    - Functor may have state
    - Functor fits into OOP
  - Cons:
    - There is more typing, a bit longer compilation time etc.
- **Instead of function pointer:**
  - Pros:
    - Functor often may be inlined
  - Cons:
    - Functor can not be swapped with other functor type during runtime (at least unless it extends some base class, which therefore gives some overhead)
- **Instead of polymorphism:**
  - Pros:
    - Functor (non-virtual) doesn't require vtable and runtime dispatching, thus it is more efficient in most cases
  - Cons:
    - Functor can not be swapped with other functor type during runtime

Better design with Functors

- **get and set accessor functions**

```
class Point {
        double Xcoord_, Ycoord_;
  –   public:
        Point();
        Point( double Xcoord, double Ycoord );
        double getXcoord() const;
        void setXcoord( double newValue );
        double getYcoord() const;
        void setYcoord( double newValue );
};
```

Avoiding accessors?!

- We can eliminate a lot of code by doing:

```
class Point {
    public:
        double Xcoord_, Ycoord_;
        Point();
        Point( double Xcoord, double Ycoord );
};
```

  - **DON't do this. It BREAKS encapsulation!!!**

Simple idiomatic accessor

```
class Point {
        double Xcoord_, Ycoord_;
    public:
        Point();
        Point( double Xcoord, double Ycoord );
        double Xcoord() const;
        void Xcoord( double newValue );
        double Ycoord() const;
        void Ycoord( double newValue );
};
```

Simple idiomatic accessor (cont...)

- **Instead of doing:**
  - Point point;
  - point.setXcoord(point.getXcoord() + 10.);
  - return point.getXcoord();
- **We can do:**
  - Point point;
  - point.Xcoord(point.Xcoord() + 10.);
  - return point.Xcoord();

Objects to the rescue

```
class Coordinate {
        double coord_;
-       public:
        Coordinate();
        Coordinate(double coord) :  coord_ {coord} {};
        double operator()() const {
            return coord_;
         }
        void operator()( double coord ) {
            coord_ = coord; }
};
```

Objects to the rescue

```
class Point {
    public:
        Coordinate Xcoord, Ycoord;
        Point();
        Point( double Xcoord, double Ycoord );
};
```

- **We can do:**
  - Point point;
  - point.Xcoord(point.Xcoord() + 10.);

Advantages

- The class is shorter and much more clearly expresses our intent without any excessive verbosity.
- Instead of writing two sets of accessors which are nearly identical we've written one in a helper class.
- We have two classes that do exactly half the job that one was doing. This means that each class is smaller and easier to understand.
- We're still free to change the underlying implementation if we want to because we haven't changed the syntax used to access the xcoord and ycoord.

Templatised

```
template <class t_coord>
class Coordinate {
        t_coord coord_;
    public:
        Coordinate();
        Coordinate(t_coord coord) :  coord_ {coord} {};
        t_coord operator()() const {
            return coord_;
         }
        void operator()( t_coord coord ) {
            coord_ = coord; }
};
```

Generic templatised accessor

- template< typename T >
- class Accessors {
  - private:

```
T t_;
```

  - public:

```
Accessors() {}
explicit Accessors( const T& t ) : t_( t ) {}
const T &operator() () const { return t_; }
void operator() ( const T& t ) {t_ = t; }
```

};

Nested Functions

- Some languages (but not C++) allow nested functions, which are similar in concept to nested classes. A nested function is defined inside another function (the "enclosing function"), such that:
  - the nested function has access to the enclosing function's variables; and
  - the nested function is local to the enclosing function, that is, it can't be called from elsewhere unless the enclosing function gives you a pointer to the nested function.
- Just as nested classes can be useful because they help control the visibility of a class, nested functions can be useful because they help control the visibility of a function.
- C++ does not have nested functions. But can we get the same effect?

Nested functions (cont...)

```
int f( int i )     {
  int j = i*2;
  int g( int k )       {
    return j+k;
  }
  j += 4;
  return g( 3 );
}
```

This won't compile!
C++ doesn't support nested functions.

- How to make this possible?
  With a little code reorganization and a minor limitation.
  The basic idea is to turn a function into a functor.

Naive "local functor" (doesn't work)

```
int f ( int i )   {
   int j = i*2;
   class g_     {
   public:
     int operator()( int k )  { return j+k;   // error: j isn't accessible
     }
   } g;
   j += 4;
   return g( 3 );
}
```

- The local class object doesn't have access to the enclosing function's variables.
- why don't we just give the local class pointers or references to all of the function's variables?

Naive "local functor + references to variables"

```
int f ( int i ) {
  int j = i*2;
  class g_  {
   public:
            g_( int& j ) : j_( j ) { }
           // access j via a reference
            int operator()( int k ) {     return j_+k;      }
  private:
    int& j_;
  } g( j );
  j += 4;
  return g( 3 );
}
```

Issues with the implementation

- For example, consider that just to add a new variable requires four changes:
  - add the variable;
  - add a corresponding private reference to g_;
  - add a corresponding constructor parameter to g_; and
  - add a corresponding initialization to g_::g_().
- That's not very maintainable.
- It also isn't easily extended to multiple local functions.
- Can we do better?
  We can do better by moving the variables themselves into the local class

Somewhat better solution

```
int f( int i )  {
    class g_     {
        public:
          int j;
          int operator()( int k ) { return j+k;}
    } g;
    g.j = i*2;
    g.j += 4;
    return g( 3 );
}
```

Nearly there...

```
int f( int i ) {
    class Local_  {
        public:
            int j;
            int g( int k ) { return j+k;  }
            void x() { /* ... */ }
    } local;
    local.j = i*2;
    local.j += 4;
    local.x();
    return local.g( 3 );
}
```

Issues

- This still has the problem that, if you need to initialize j using something other than default initialization, you have to add a clumsy constructor to the local class just to pass through the initialization value.
- The original question tried to initialize j to the value of i*2; here, we've had to create j and then assign the value, which could be difficult for more complex types.

A more complete solution

- class f {
  - int retval; // f's "return value"
  - int j;
  - int g(int k) { return j + k; };
  - void x() { /* ... */ }
  - public:
    - f(int i) : j(i*2) { // original function, now a constructor
    - j += 4;
    - x();
    - retval = g(3);
  - }
- operator int() const { // returning the result
  - return retval;}
- };

std::function

- Class template std::function is a general-purpose polymorphic function wrapper.
- Class that can wrap any kind of callable element (such as functions and function objects) into a copyable object, and whose type depends solely on its call signature (and not on the callable element type itself).
- An object of a function class instantiation can wrap any of the following kinds of callable objects: a function, a function pointer, a pointer to member, or any kind of function object (i.e., an object whose class defines operator(), including closures).

References

- http://www.kirit.com/C%2B%2B%20killed%20the%20get%20%26%20set%20accessors
- http://www.kirit.com/C%2B%2B%20killed%20the%20get%20%26%20set%20accessors/A%20simple%20meta-accessor