CPSC 427: Object-Oriented Programming

Michael J. Fischer

Preview Lecture 21
November 14, 2018

Singleton Design Pattern (revisited)

More on Functions

Casts and Conversions

Singleton Design Pattern (revisited)

Another version of `Serial`

In demo `20b-SmartPointer`, we used the singleton design pattern to create class `Serial` to serve as a UID generator.

To review, a public static function `uidGen()` returns a pointer to a newly created instance of `Serial` the first time it is called, and it saves that pointer in a private static variable `Sobj`.

Subsequent calls to `uidGen()` simply return the saved pointer.

Because the constructor is private, no other instantiations are possible.

The instance defines a public `operator()`, making it a functor which can be called to return the next UID.

Drawbacks to this implementation

The primary drawback to the implementation of `Serial` in `20b-SmartPointer` is that the client must do two steps to get the next UID:

1. Call `Serial::uidGen()` to obtain the instance pointer `ip`.
2. Call `ip()` to get the next UID.

In the `SPtr` example, we confusingly called the instance pointer `uidGen` so we could write `uidGen()` to get the next UID.

By choosing to store the pointer `uidGen` as a data member of `SPtr`, we incur the storage cost on every instance of `SPtr`.

A streamlined UID generator

In demo `21a-SmartPointer`, we improve the implementation of `Serial` so that there only a single public static function`nextID()` that the client must call to get the next UID, e.g.,

```
const int my_id = Serial::newID();
```

To do this, the code is turned around. `uidGen()` becomes private, and the new public static function `newID()` replaces the old `operator()` extension.

Now `newID()` calls `uidGen()` each time it is called to get the instance pointer, which it then uses to access the private data member `nextUID`.

Serial.hpp, version 2

```
// Singleton class for generating unique ID's

class Serial {
private:
   static Serial* Sobj; // pointer to singleton Serial object
   int nextUID=0;       // data member for next UID to be assigned
   static Serial* uidGen() {  // instaniates Serial on first call
      if (Sobj == nullptr) Sobj = new Serial;
      return Sobj;
   }
   Serial() =default;   // private constructor prevents external instantiation
public:
   static int newID() { return uidGen()->nextUID++; }
};
```

A UID generator

What we want is a class `Serial` with a private data member `nextUID` and a public function `uidGen()` that returns and updates the next UID.

In order to call the function, we need a class instance `uidGen` of `Serial` that initializes `nextUID` and supports the public function `uidGen()`. Now, to generate a new serial number, simply call `uidGen.uidGen()`.

However, this solution has two problems:

1. How can one make the object `uidGen` available wherever needed?
2. Where should `Serial` be instantiated?

Singleton class

A singleton class solves both problems.

1. It has a static function that returns a pointer to the single instantiation whenever it is called.
2. Initially there is no instantiation, so it creates and remembers an instantation the first time it is called. It uses a private static variable for this purpose.

Functors

A **functor** is an object that acts like a function.

Let `obj` be a functor. Then one can write `obj()`, pretending that `obj` is a function.

All that is needed to make this work is to define `operator()` within the class.

For our UID generator, we define the behavior of `obj` to be the same as for `uidGen()` discussed above.

Serial.hpp

```
// Singleton class for generating unique ID's

class Serial {
private:
   static Serial* Sobj;
   int nextUID=0;
   Serial() =default;
public:
   static Serial& uidGen() {
      if (Sobj == nullptr) Sobj = new Serial;
      return *Sobj;
   }
   const int operator()() { return nextUID++; }
};
```

Serial.cpp

```
// Initialize Serializer static variable
Serial* Serial::Sobj = nullptr;
```

More on Functions

Functional composition

Functional composition refers to using the result returned by one function as the argument for another.

Example: `g(f(x))`.

The type of `f(x)` (which is the result type declared in the definition of `f()`) must be **compatible** with the corresponding parameter type for *some* method of `g()`.

Types are compatible if they are the same, or if the result type can be converted to the corresponding parameter type.

Type compatibility

Here's what the compiler does when it sees the call `g(f(x))`.

1. It finds the type of `f(x)`. Call it `T`.
2. It looks for a method for `g` with **signature** `(T)`.
3. If it finds one, that method is selected.
4. If not, it searches the methods for `g` with signatures that are compatible with `(T)`, meaning that it is possible to convert `T` to the type required by the signature.
5. If it finds exactly one such method, then that is used.
6. If it fails to find one, it reports "no match", and it lists the candidates it tried.
7. If it finds more than one possible method, it reports "ambiguous".

Calling constructors implicitly

Normally, constructors are called implicitly when an object is created, whether by `new` (in the case of dynamic storage) or by having a declaration executed (in the case of automatic storage).

When several constructor methods are present, which is chosen depends on the arguments supplied, either explicity or through ctors, but the call itself is implicit.

Examples

- `MyClass b` creates a stack object and invokes the default constructor `MyClass()`.
- `MyClass b(4);` creates a stack object and invokes constructor `MyClass(4)`.
- `new MyClass(6)` creates a dynamic object and invokes constructor `MyClass(6)`.

Calling constructors explicitly

Constructors can also be called explicitly, just like ordinary global functions.

The meaning is to create a new temporary stack object, just as a new temporary is created to hold the result of `y+z` in the expression `x*(y+z)`.

As with all object construction, the constructor is called when the object is created, and the destructor is called when it is deleted.

Because the created object is temporary, it must be used immediately, after which it will be discarded.

This is how `throw Fatal("Error message")` works. `Fatal()` creates an exception object of type `Fatal` for use by `throw`.

Conversion using constructor

Now suppose `f()` returns an object of type `A&` and `g()` expects an argument of type `B`. What happens with `g(f())`?

Example 1:

```
class A;  // forward declaration

class B {
public:
    B(){}
    B(A& aa) { cout << "B constructor called" << endl; }
};
```

Compiler will use `B`'s constructor to build a `B&` from an `A&`.

Output is “`B constructor called`”.

Conversion using a cast

Example 2:

```
class B;  // forward declaration

class A {
public:
    operator B() {
        cout << "operator B cast called" << endl;
        return *new B;
    }
};
```

Compiler will use `A::operator B()` to cast the `A&` returned by `f()` to the `B` expected by `g()`.

Output is "`operator B cast called`".

What if both options exist?

```
class A;  // forward declaration
class B { public:
    B(){}
    B(A& aa) { cout << "B constructor called" << endl; }
};
class A { public:
    operator B() {
        cout << "operator B cast called" << endl;
        return *new B;
    }
};
A& f() { return *new A; }
B& g(B aa) { return *new B; }
```

Compiler will complain "`error: conversion from 'A' to 'B' is ambiguous`".

Casts and Conversions

Casts in C

A C cast changes an expression of one type into another.

Examples:

```
int x;
unsigned u;
double d;
int* p;

(double)x;     // type double; preserves semantics
(int)u;        // type unsigned; possible loss of information
(unsigned)d;   // type unsigned; big loss of information
(long int)p;   // type long int; violates semantics
(double*)p;    // preserves pointerness but violates semantics
```

Different kinds of casts

C uses the same syntax for different kinds of casts.

Value casts convert from one representation to another, partially preserving semantics. Often called *conversions*.

- `(double)x` converts integer `x` to equivalent `double` floating point representation.
- `(short int)x` converts integer `x` to equivalent `short int`, *if the integer falls within the range of a* `short int`.

Pointer casts leave representation alone but change interpretation of pointer.

- `(double*)p` treats bits at destination of `p` as the representation of a double.

C++ casts

C++ has four kinds of casts.

1. *Static cast* includes value casts of C. Tries to preserve semantics, but not always safe. Applied at compile time.
2. *Dynamic cast*. Applies only to pointers and references to objects. Preserves semantics. Applied at run time. [See demo `21b-Dynamic_cast`].
3. *Reinterpret cast* is like the C pointer cast. Ignores semantics. Applied at compile time.
4. *Const cast*. Allows `const` restriction to be overridden. Applied at compile time.

Explicit cast syntax

C++ supports three syntax patterns for explicit casts.

1. C-style: `(double)x`.
2. Functional notation: `double(x); myObject(10);`.
   (Note the similarity to a constructor call.)
   Only works for single-word type names.
3. Cast notation:
   ```
   int x; myBase* b; const int c;
   ```
   - `static_cast<double>(x);`
   - `dynamic_cast<myDerived*>(b);`
   - `reinterpret_cast<int*>(p);`
   - `const_cast<int>(c);`

Implicit casts

General rule for implicit casts: If a type `A` expression appears in a context where a type `B` expression is needed, use a semantically safe cast to convert from `A` to `B`.

Examples:

- Assignment: `int x; double d; x=d; d=x;`
- Pointer assignment:
  ```
  class A { ... };
  class B : public A { ... };
  A* ap; B* bp; ap = bp;
  ```
- Initialization:
  `A a=x;` converts `x` to an `A`, then copies.
- Construction:
  `A a(x);` calls `A` constructor, possibly casting `x`.

Ambiguity

Can be more than one way to cast from `B` to `A`.

```
class B;
class A { public:
  A(){}
  A(B& b) { cout<< "constructed A from B\n"; }
};
class B { public:
  A a;
  operator A() { cout<<"casting B to A\n"; return a; }
};
int main() {
  A a; B b;
  a=b;        // Triggers error comments
}
```

Comment from `g++`: conversion from 'B' to 'A' is ambiguous

Comment from `clang++`: error: reference initialization of type 'A &&' with initializer of type 'B' is ambiguous

explicit keyword

Not always desirable for constructor to be called implicitly.

Use `explicit` keyword to inhibit implicit calls.

Previous example compiles fine with use of `explicit`:

```
class B;
class A {
public
  A(){}
  explicit A(B& b) { cout<< "constructed A from B\n"; }
};
...
```

Question: Why was an explicit definition of the default constructor not needed?