C++ Topics

Jonathan Hoyle
Eastman Kodak
2/8/01

Overview

- Constructors
- Destructors
- References
- Const
- Q & A

Constructors

Constructors

- ...are called only ***after*** object memory is allocated (not called if `new` fails)
- ...are invoked ***after*** its base class constructors are completed
- ...are not inherited
- ...do not have return values, can't "fail"
- ...cannot be called directly
- ...do not have function addresses
- ...cannot be declared `static` or `virtual`

Default Constructor

- Constructors with no parameters (or parameters which all have defaults)
- If no constructor is defined for a class, a public default constructor is implied
- The empty parantheses are not used when invoking the default constructor:

```
TypeName   IDname;          // default constr
TypeName   IDname( );       // extern func
```

Constructors with 1 parameter

- Can be constructed with either ( ) or =:

```
class X
{  X( );                         // default constructor
   X(int i);                     // int constructor
   X &operator=(int i);          // assignment operator
   void operator( )(int i); };   // fcn operator

X   a1 = 0;                      // a1: int constructor
X   a2;                          // a2: default constructor
    a2 = 0;                      //  followed by assignment

X   b1(0);                       // b1: int constructor
X   b2;                          // b2: default constructor
    b2(0);                       //    followed by function
```

Explicit Constructor

- Construction with = can be supressed with the `explicit` keyword:

```
class XString
{
    public:
        XString(char *inString);
        explicit XString(int inSize);
};

XString x1("Hello");   // uses char* constructor
XString x2 = "World"; // uses char* constructor
XString x3(256);       // uses int constructor
XString x4 = 128;      // compiler error
```

Copy Constructors & Assignments

- Copy Constructors have this prototype:

```
TypeName::TypeName(TypeName &inVar);
```

- If no constructor is defined, a (bitwise) public copy constructor is implied
- Check code for overlap situations:

```
// The code below fails for "x = x;"
X &X::operator=(const X &inVar)
{
    memset(mString, 0, 256);
    strcpy(mString, inVar.mString);
}
```

What is a "Static Constructor"?

- Technically, there is no such thing in C++
- Usually term is used to describe a static method which creates an object:

```
class X
{
    public:
        X( );
        ~X( );
        static X *Create( )
        { return new X; }
};
```

What is a "Virtual Constructor"?

- Technically, there is no such thing in C++
- Describes a way to create an object whose type is determined at runtime:

```
class Base                          { ... };
class Derived1: public Base  { ... };
class Derived2: public Base  { ... };

Base *Base::Create(int inType)
{
    if (inType == 1) return new Derived1;
    if (inType == 2) return new Derived2;
    return new Base;
}
```

What is an "Anonymous Constructor"?

- Technically, there's...no wait! It is in C++!
- It's the construction of an object without explicitly assigning it to a variable

```
class X                // Class Declaration for X
{
    public:            // Constructor for X
        X(int x);      //              taking an int
};

int foo(X inVar);      // Prototype of fcn using X

foo(X(1));             // Anonymously constructing
                       //                X from 1
```

Constructing & Memory Allocating

- What if you want the memory allocation to take place independently from construction?
  - Allocation & Construction at the same time:

```
X   *xPtr = new X;
```

  - Allocation without Construction:

```
X   *xPtr = (X *) new char[sizeof(X)];
```

  - Construction without Allocation:

```
new (xPtr) X;
```

Constructing Arrays of Objects

- Trivial when using the default constructor:

```
X   myArray[10];   //uses default constructor
```

- How do you do it without using the default?

```
class X
{
    public:
        X(char *inString, int inSize = 256);
};

// Constructor arguments must be array-listed
X myArr[3] = { “Hi”, X(“C++”), X(“C”, 100) };
```

Constructor Errors

- Since Constructors cannot “fail” and do not have a return value, here are some options:
  - Require a separate *initialization* method to be invoked before the object can be used
  - Include a reference to an error parameter in the constructor
  - Throw an exception

Constructor/Initialization pair

- Essentially a two-part construction
- They're "zombie objects" until initialized

```
class X                         // declaration
{
    public:
        X( );                   // constructor
        bool init( );           // initializer
};

X       x;                      // x constructed
if (x.init( ))                  // x initialized
```

Constructor Error Parameter

- Requires the user to check the error after construction:

```
class X                              // declaration
{
    public:
        X(bool &outVal);             // constructor
};

bool    ifOK = false;                // bool check
X       x(ifOK);                     // construct x
if (ifOK)                            // check error
{  ...
```

Constructor Exception

- You've jumped out of the object's scope
- Object never lived, destructor not called

```
try                    // must create a try block
{
    X   x;             // construct x
}

catch (bool inError)
{
    // Handle error
}
```

Bad Constructor Error Handling

- Why wouldn't this work?

```
bool X::init( )
{  /* Do error checking */  }

X::X( )
{
    bool ifOK = init( );    // Call init code
    if (!ifOK)
    {                       // If init fails,
        delete this;        //   delete object
        this = NULL;        //   set to NULL
    }
}
```

Constructor Gotcha’s

- Do not assume polymorphic behavior from virtual functions inside constructors
- Don’t use `this` too early:

```
// “this” allocated but not fully constructed
X::X( ) { ... foo(this); ... }
```

- Be careful of ambiguity between type conversions and constructors:

```
X::X(const Y &)    { ... } // Does x=y use this?
X::X(const X &)    { ... } // Or the copy constr
Y::operator X&( ) { ... } //  after conversion?
```

Destructors

Destructors

- ...are called ***before*** object memory is deallocated (not called if `delete` on `NULL`)
- ...are completed ***before*** its base class destructors are invoked
- ...are not inherited
- ...have only one prototype, no parameters
- ...may be called directly
- ...cannot be declared `static`
- ...can be `virtual` (and even pure virtual)

Virtual Destructors

- Necessary for polymorphism
- You almost always want to make it virtual

```
class Base                   { ... };
class Derived: public Base  { ... };

Base *bPtr = new Derived; // Create Derived
DoStuff(bPtr);             // Do stuff
delete bPtr;               // Delete Derived?
```

- In above example, Derived's destructor will never get called if it's not `virtual`.

Pure Virtual Destructors

- Destructors can be pure virtual as well:

```
class X
{
    public:
        X( );
        virtual ~X( ) = NULL;
};
```

- The class necessarily becomes abstract
- Subclasses are not (dest’s not inherited)
- Must implement destructor, even if pure

Destructing & Memory Deallocating

- What if you want the memory deallocation to take place independently from destruction?
  - Deallocation & Destruction at the same time:

```
delete xPtr;
```

  - Deallocation without Destruction:

```
delete     (void *) xPtr;  //if using new
delete [ ] (void *) xPtr;  //if using new [ ]
```

  - Destruction without Deallocation:

```
xPtr->~X();
```

References

Pass by Reference

- Allows variables to be modified without having to check pointer validity
- Const reference passing gives better performance than pass by value:

```
void foo(X inObj);              // less optimal
void foo(const X &inObj);       // more optimal
```

- Types must match, no conversion:

```
void incr(long &x) { x++; } // increments long

short x = 12;       // x stays 12 since only
incr(x);            // a temp copy is changed
```

Reference variables

- "References are synonyms, not objects."

```
int *ptr1 = aPtr;          // ptr1 takes up space
int &ref1 = myInt;         // ref1 does not!
```

- Types must be exact:

```
unsigned int uInt = 0;
int &ref2 = uInt;           // Error!

int intArray[10];
int *&ref3 = intArray;      // Error!
```

- Must be assigned at time of declaration
- Can't have arrays of references

Const

const pointers

- Read from right to left (mostly):

```
const  T *p;          // ptr to a const T
const  T *const p;    // const ptr to const T
T  const *p;          // ptr to a const T
T *const  p;          // const ptr to a T
T  const *const p;    // const ptr to const T
```

- Note that `const T *p` == `T const *p`
- enum’s or const’s?
  - enum definitions do not take up memory
  - const allow freer additions

const member functions

- Indicate method will not change object:

```
void X::foo( );         // foo( ) might change x
void X::bar( ) const;   // bar( ) will not
```

- Functions operating on const objects are free to call const methods:

```
void ExamineX(const X &inObject)
{
    inObject.foo( );  // Error! Can't use foo
    inObject.bar( );  // OK! bar is safe to use
}
```

const_cast< >

- Very dangerous, allows you to overwrite const data:

```
void foo(const int &inVal)
{
    int &theVal = const_cast<int &>(inVal);
    theVal++;
}

int x = 5;   // Set our variable to 5
foo(x);      // Should be OK, foo claims const
cout << x;   // Oh no! x is now 6!
```

logical const vs. bitwise const

- The intention behind `const_cast< >` is to allow changes to the “bitwise state” to a class while leaving the “logical state”.
- For example, performing diagnostics, optimization or caching.
- `const_cast< >` changes to an object in read-only memory is undefined behavior
- Better than `const_cast< >`, use the new `mutable` keyword

const_cast< > vs. mutable

- Variables declared `mutable` are free to be modified even if the method is `const`:

```
class X
{
    public:
        double getData( ) const;
        { mCount++; return mData; }

    protected:
        double      mData;
        mutable int mCount;
};
```

Q & A