Chapter 1
Overview

Sherri Shulman

Math-linux.com

Oct 1, 2013

Very brief Intro C++ – part 2

1. So there are three special functions (actually 4 if you include constructors).
2. Destructor
3. Copy Constructor
4. operator=
5. There are default definitions that C++ provides, and sometimes those are adequate; sometimes not.

1. The destructor dtermines how memory is released when an object goes out of scope
2. it is implicitly called when you do delete.
3. The default just calls delete on each data member.
4. Consider that in some data structures with links (pointers) to the next item in some traversal ... the default destructor will not know how to follow those links.

1. Suppose that I have IntCell variables B and C.
2. If I say IntCel B = C; I intend that B will be a copy of C. How is that copy made? With the copy constructor.
3. This also implies if you initialize B by saying IntCell B(C);
4. But not B = C; which is using the operator=
5. If you pass an IntCell by value, than a copy needs to be placed on the stack.
6. If you return a value, again it needs to be copied into the return location.
7. (in a small item like IntCell, call by value is fine, but in a large collection call by value is rarely done.)
8. Like the destructor, we want to call the copy constructor on each of the data members (essentially giving us a deep copy).

1. operator= is the assignment operator.
2. it typically does a copy. I.e. B = C will copy C into B.
3. again it applies operator= to each data member.

1. As mentioned above, defaults don't work if we have pointers (links) since a pointer is a value, and what would that mean?
2. If we copied the pointer, then we are pointing to the same thing and it isn't really a deep copy (that it typically called a shallow copy).
3. – Draw a picture on the board –
4. Sometimes we may want a shallow copy.

```
IntCell::~IntCell( )
{
    // Does nothing, since IntCell contains only an int data
    // member. If IntCell contained any class objects, their
    // destructors would be called.
}

IntCell::IntCell( const IntCell & rhs ) : storedValue( rhs.storedValue )
{
}

const IntCell & IntCell::operator=( const IntCell & rhs )
{
    if( this != &rhs )   // Standard alias test
        storedValue = rhs.storedValue;
    return *this;
}
```

1. The protoypes (types or signatures) of the big 3 ...
2. notice that constructors and destructors don't have return types. In a sense they are not true functions but are part of the connection between the system (memory allocation and deallocation) and the program.
3. The type of the copy assignment operator is ¡b¿ const IntCell & operator=(const IntCell & rhs);
4. The operator= has an implicit argument (the receiver). Eg B=C, B is the implicit argument. C is the explicit argument. The return type is a const IntCell &. You may think it odd that this is a const when in fact it changes. operator= actually associates to the right. So in a = b = c, it really is a = (b = c). So c is the arg, it is const and is not changed. b actually changes (the type notwithstanding) but the result of b = c needs to be a const ref, so that's what operator= returns.
5. Next an exmaple of a pointer, where the default would not work:

```
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 )
      { storedValue = new int( initialValue ); }

    int read( ) const
      { return *storedValue; }
    void write( int x )
      { *storedValue = x; }
  private:
    int *storedValue;
};
```

1. So pointers present a problem for the default. A pointer is a value and so what is copied is the value of the pointer.
2. Let's look at the pointer version of IntCell (Fig1.14) and how it presents a problem (Fig 1.15):

```
int f( )
{
    IntCell a( 2 );
    IntCell b = a;
    IntCell c;

    c = b;
    a.write( 4 );
    cout << a.read( ) << endl << b.read( ) << endl << c.read( ) << endl;
    return 0;
}
```

1. In this we set a to 2.
2. Then we assign a to b.
3. Then we assign b to c.
4. Since this was a shallow copy, really they only copied the same pointer value, so a,b,c all refer to the same stored value.
5. The solution is to implement the big 3. See Fig 1.16

```
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 );

    IntCell( const IntCell & rhs );
    ~IntCell( );
    const IntCell & operator=( const IntCell & rhs );

    int read( ) const;
    void write( int x );
  private:
    int *storedValue;
};

IntCell::IntCell( int initialValue )
{
    storedValue = new int( initialValue );
}

IntCell::IntCell( const IntCell & rhs )
{
    storedValue = new int( *rhs.storedValue );
}

IntCell::~IntCell( )
{
    delete storedValue;
}

const IntCell & IntCell::operator=( const IntCell & rhs )
{
    if( this != &rhs )
        *storedValue = *rhs.storedValue;
    return *this;
```

1. C arrays and strings
2. If we say int arr1[10] then arr1 is a constant pointer to a contiguous memory space large enough to hold 10 ints.
3. This is the same as we spoke of in the paper for the regexpr.
4. So arr1 acts like a pointer in many ways, but we can't assign to it, since it is a constant pointer to this preallocated space.
5. We can also make an array using new: int *arr2 = new int[10].
6. But then remember to delete!

1