Chapter 1
Overview

Sherri Shulman

Math-linux.com

Oct 1, 2013

Very brief Intro C++

1. Resources in many places on learning C++
2. Check syllabus page for two
3. Chapter 1 in textbook
4. C++ is a superset of C
5. OS work in C!
6. Data structures work primarily in C++

1. C++ is an object oriented language built on C
2. Some of it will look very familiar (like Java)
3. But C++ is a much "larger" language and includes more programmer control
4. C++ is definitely closer to the machine in some senses
5. C++ includes pointers ... always a challenge

1. Basic class syntax
2. We'll follow the example from the book
3. Best way to learn a language is to read lots of (good) code
4. (and read the reference manual, but that's pretty dry)

1. A C++ class consists of members: either data or functions
2. The functions are called *member functions*
3. Like Java, some member functions are constructors.
4. Like Java, constructors (and other functions) can be overloaded.
5. Like Java, there are visibility keywords: private and public to start.
6. Lets look at the code from figurer 1.5 (the first IntCell)

```
/**
 * A class for simulating an integer memory cell.
 */
class IntCell
{
  public:
    /**
     * Construct the IntCell.
     * Initial value is 0.
     */
    IntCell( )
      { storedValue = 0; }

    /**
     * Construct the IntCell.
     * Initial value is initialValue.
     */
    IntCell( int initialValue )
      { storedValue = initialValue; }

    /**
     * Return the stored value.
     */
    int read( )
      { return storedValue; }

    /**
     * Change the stored value to x.
     */
    void write( int x )
      { storedValue = x; }

  private:
    int storedValue;
};
```

1. Conventionally in C++ the data fields come at the end (don't ask me why)
2. Now we're going to tweak this code.
3. Figure 1.6 is pretty much the same code with some slight changes
4. First it has altered the constructors ... now there's just one
5. It uses a default parameter
6. We've also made it explicit (keyword) – to prevent automatic coercions
7. We used an initializer list.
8. we've modified the type of read to assert that it doesn't change state

```
/**
 * A class for simulating an integer memory cell.
 */
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 )
      : storedValue( initialValue ) { }
    int read( ) const
      { return storedValue; }
    void write( int x )
      { storedValue = x; }

  private:
    int storedValue;
};
```

1. Using a default parameter means that you have some control over default information without writing a separate constructor.
2. More important ... the initializer list is an efficient way to initialize the data fields. In particular, if a data field itself is a *const* then it must be initialized using an initializer list, since you can't assign to a const.
3. Also, if a data field is a class (and not a primitive type) and it does not have a 0-parameter constructor, then it must be initialized with the initializer list.

1. Explicit keyword.
2. C++ has a list of rules it uses to do type conversions.
3. In particular, suppose that I mistakenly write:
4. IntCell i2 = 6;
5. Did I mean that? Do I want the g++ compiler to "catch" this as a type error?
6. without the *explicit* keywork, c++ looks for a constructor that takes an integer and produces an IntCell. It then assigns that newly created IntCell to i2.
7. Using explicit requires that you eplicitly call the constructor. (a note on what is explicit shortly)

```
#include <cstdlib>
#include <iostream>
#include "IntCell.h"

using namespace std;


int main(int argc, char *argv[]) {
IntCell i1(5) ;
/*
 IntCell i2 = 6; this is ok without the explicit
 with the explicit keyword: get a type error
 TestIntCell.cpp: In function âint main(int, char**)â:
 TestIntCell.cpp:10:15: error: conversion from âintâ to
 non-scalar type âIntCellâ requested
*/
IntCell i2(6);


//cout << "Intcell: " << i1.read() << endl;

cout << "Intcell using overloaded: " << i1 << endl;
cout << "Intcell using overloaded: " << i2 << endl;

}
```

1. C++ uses header files ... this is part of a preprocessor package
2. Any line that starts with a hash is interpreted by the preprocessor.
3. We'll see a few examples shortly.
4. In particular include files (used for a variety of purposes) are introduced with *include*. The preprocessor then reads the indicated file into the file that referenced it at that point.
5. You might immediate say ... what prevents it from being loaded more than once?
6. Nothing ... so you have to protect yourself. Let's look at fig 1.7:

```
#ifndef IntCell_H
#define IntCell_H

/**
 * A class for simulating an integer memory cell.
 */
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 );
    int read( ) const;
    void write( int x );

  private:
    int storedValue;
};

#endif
```

1. OK ... so Fig 1.7 is a header file that essentially includes the interface to the IntCell class.
2. It also includes the data field(s)
3. But it does not include any function definitions (implementations)
4. So in a way it's sort of like an interface in Java
5. But now, when we want to introduce the implementations, we no longer specify the class, since the class has already been introduced in this header file (sometimes these types are called prototypes)
6. Fig 1.8 shows the implementation file.

```
#include "IntCell.h"

/**
 * Construct the IntCell with initialValue
 */
IntCell::IntCell( int initialValue ) : storedValue( initialValue )
{
}

/**
 * Return the stored value.
 */
int IntCell::read( ) const
{
    return storedValue;
}

/**
 * Store x.
 */
void IntCell::write( int x )
{
    storedValue = x;
}
```

1. How to compile and use
2. So you may be using an IDE which will then do most of your work for you.
3. But you should know the basics of compiling and loading on Unix/Linux
4. Look at Figure 1.9:
5. To compile and load this: g++ IntCell.cpp TestIntCell.cpp
6. (assuming these are the names.)
7. you can also do just the compile using the -c option on g++ which will leave a file IntCell.o or TestIntCell.o in your directory.

```
#include <iostream>
#include "IntCell.h"
using namespace std;

int main( )
{
    IntCell m;   // Or, IntCell m( 0 ); but not IntCell m( );

    m.write( 5 );
    cout << "Cell contents: " << m.read( ) << endl;

    return 0;
}
```

1. C++ vectors vs arrays.
2. Both Java and C++ have the same problem with arrays: they come from C.
3. Because they were inherited from C, they behave differently than other kinds of objects. Java has made various changes to "fix" arrays, with varying degrees of success.
4. For C++, somethings that work for objects don't work for arrays.
5. So, for instance, in C++ objects it is conventional to overload the assignment operator so that you can say: o1 = o2 and something sensible happens (usually a copy, but to some extent this is up to the implementor).
6. You can't use assignment to copy arrays.
7. C++ arrays don't have any index checking.

1. Strings are just arrays of characters (which they are in C and also in Java).
2. However in Java, the String class has added more behavior.
3. Similarly, in the STL (standard template library) strin and vectors behave more as you'd expect.
4. In particular the string class (as opposed to the primitive string) understands comparison operators such as ==, ¡=, and so no.
5. Both vectors and strings can be copied with the assignment operator.

1. In order to use a data structure provided in the STL you have to include them
2. Fig 1.10 shows an example with vector:

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> squares( 100 );

    for( int i = 0; i < squares.size( ); i++ )
        squares[ i ] = i * i;

    for( int i = 0; i < squares.size( ); i++ )
        cout << i << " " << squares[ i ] << endl;

    return 0;
}
```

1. Now onto pointers
2. You may think you understand pointers, but in reality, no one really understands pointers until you've used them extensively.
3. Pointers are like Java references in some ways but they are more powerful.
4. Pointers are values and they can be manipulated independently of what they point to (unlike references).
5. But some of the uses of pointers are exactly like references!
6. As in java, we often want to store pointers (references) to objects and not the object itself.
7. Unlike java, there is no automatic garbage collection so that you have to keep track of your pointers so you can release memory when required (this is the source of many memory leaks!)
8. When you allocate memory dynamically you get a pointer to the memory that has been allocated.

```
int main( )
{
    IntCell *m;

    m = new IntCell( 0 );
    m->write( 5 );
    cout << "Cell contents: " << m->read( ) << endl;

    delete m;
    return 0;
}
```

1. This example is just to show you how it works
2. No benefit in this case to use the pointer rather than the object itself.
3. But it does demonstrate the flexibility you have that Java does not
4. Meaning that sometimes you may want the object itself and sometimes the pointer and you have control over this.
5. Like java the *new* keyword dynamically allocates an object
6. Different styles of allocation (m = new IntCell() vs m = new IntCell)
7. note that (IntCell m) allocates m on the stack ... ie m is an IntCell, not a reference or pointer.
8. when the original program declared (IntCell m) it is automatically garbage collected when you exit.
9. So they are called automatic variables (automatically allocated and deallocated.)

1. Operators and syntax associated with pointers
2. The declaration (IntCell * m) indicates that m is a pointer to an IntCell
3. (IntCell & m) indicates that m is a reference to an IntCell
4. If m is an IntCell, (& m) is the addressof operator (ie it returns to you the address of m).

1. So garbage collection and memory leaks
2. You have to keep track of every use of new so that you can release that memory using the delete operation.
3. In C++ we have both constructors (to allocate memory) and destructors (to deallocate memory.
4. Delete is demonstrated in Fig 1.11 above
5. Don't use new unless you need it.

1. Since pointers are values they behave like values
2. If I compare two pointers, they are equal if they have the same value (ie the are the same address)
3. A pointer can be less than another: in an array, the address of an item at position 1 has a smaller value than the address of an item at position 2.
4. This relationship is sometimes used for a compact way to move through an array and test for the end of the array though it leads to obscure looking code
5. If p1 and p2 are both pointers, then the assignment p1 = p2 with make p1 have the same value as p2 (so they both point to the same object)
6. If you have a pointer to an object that has a data field that you can access you use the -¿ operator (see fig 1.11 again)
7. If you have the object itself you use the . operator (like Java)

1. Parameter passing
2. As in many other things, C++ allows significant programmer control
3. In Java all constructed types are passed by reference, while all primitive types are passed by value.
4. C is pass by value (except for arrays which are passed as a pointer, which is a value, but maybe not the value you expect)
5. C++ allows three ways to pass parameters: call by constant reference, call by value, and call by reference. Note that if you pass in a pointer, technically it is call by value, since a pointer is a value.

1. The eg from the book:
2. double avg (const vector¡int¿ & arr, int n, bool & errorFlag);
3. Here the vector is a reference but it is const, so it can't be altered.
4. the second parameter is a call-by-value: the int is copied to a local variable.
5. The third parameter is call by reference, and a value can be "returned" to the calling function.
6. Note that I could also say: bool *errorFlag which would pass the pointer.
7. Pointers and references behave differently.
8. If p is a pointer to an IntCell, I have to say p-¿write(5).
9. If p is a reference to an IntCell, I just say p.write(5)
10. So even though they are both addresses ... they still are different.

1. How to choose?
2. Call by value is good for small objects that you don't want to change
3. Call by reference is good for objects that you want to alter persistently in the program
4. Call by const ref is good for large objects that you don't want to change
5. Call by pointer mostly is like call by reference but more dangerous

1. Just as you can pass parameters differently, you can also return values differently
2. return by value
3. return a pointer (which is a value)
4. return by const reference
5. return by reference ... limit the use of this
6. in general be careful about returning a reference since you need to know that the reference will correctly refer to an object that does not disappear (eg an automatic var)
7. Check fig 1.12

```
const string & findMax( const vector<string> & arr )
{
    int maxIndex = 0;

    for( int i = 1; i < arr.size( ); i++ )
        if( arr[ maxIndex ] < arr[ i ] )
            maxIndex = i;

    return arr[ maxIndex ];
}

const string & findMaxWrong( const vector<string> & arr )
{
    string maxValue = arr[ 0 ];

    for( int i = 1; i < arr.size( ); i++ )
        if( maxValue < arr[ i ] )
            maxValue = arr[ i ];

    return maxValue;
}
```

1. So what's the difference between references and pointers?
2. Pointers are values and are independent of what they point to.
3. References are stuck to the objects they point to.
4. You can't do reference arithmetic, but you can do pointer arithmetic.
5. You may have a null reference, but you can't have an address that isn't stuck to some object. You can have a point that points anywhere.
6. So unless there's a real need, references are better than pointers and less danger prone.
7. However you will find that pointers are often used in OS programming.

1. You may use references in a variety of places.
2. References can be used for data fields, temp variables, etc.
3. Next class we will start with the big three: destructor, copy constructor, and operator=
4. These three are used implicitly by C++ and C++ does provide default values for them.
5. However, you may want to define these three together for new classes you create so that you control how a value is copied (for parameter passing purposes, for instance), how a value is assigned (shallow copy? deep copy? by reference?) and how an object is freed (releasing memory)