CS3157: Advanced Programming

Lecture #11

Nov 21

Shlomo Hershkop

shlomo@cs.columbia.edu

Outline

- Feedback
- More CPP
  - Language basics: identifiers, data types, operators, type conversions, branching and looping, program structure
  - data structures: arrays, structures
  - pointers and references
  - I/O: writing to the screen, reading from the keyboard, iostream library
  - classes: defining, scope, ctors and dtors
- Reading
  - c++core ch 3-6

Feedback

- Emailing TA
  - If you send an email and do not get a reply (mutliple emails) its possible they are not getting it….try to cc myself or go talk to the TA during office hours.
- Problems with the labs
  - C string comparisons
  - Pointers

C++ vs. Java

- advantages of C++ over Java:
  - C++ is very powerful
  - C++ is very fast
  - C++ is much more efficient in terms of memory
  - compiled directly for specific machines (instead of bytecode layer, which could also be seen as a portability advantage of Java over C++...)
- disadvantages of C++ over Java:
  - Java protects you from making mistakes that C/C++ don't, as you've learned now from working with C
  - C++ has many concepts and possibilities so it has a steep learning curve
  - extensive use of operator overloading, function overloading and virtual functions can very quickly make C++ programs very complicated
  - shortcuts offered in C++ can often make it completely unreadable, just like in C

Identifiers

- i.e., valid names for variables, methods, classes, etc
- just like C:
  - names consist of letters, digits and underscores
  - names cannot begin with a digit
  - names cannot be a C++ keyword
- literals are just like in C with a few extras:
  - numbers, e.g.: 5, 5u, 5L, 0x5, true
  - characters, e.g., 'A'
  - strings, e.g., "you" which is stored in 4 bytes as 'y', 'o', 'u', '\0'

data types

- simple native data types: bool, int, double, char, wchar_t
- bool is like boolean in Java
- wchar_t is "wide char" for representing data from character sets with more than 255 characters
- modifiers: short, long, signed, unsigned, e.g., short int
- floating point types: float, double, long double
- enum and typedef just like C

Operators

- same as C, with some additions
- if you recognize it from C, then it's pretty safe to assume it is doing the same thing in C++

Type conversions

- all integer math is done using int datatypes, so all types (bool, char, short, enum) are promoted to int before any arithmetic operations are performed on them
- mixed expressions of integer / floating types promote the lower type to the higher type according to the following hierarchy:

```
int < unsigned < long < unsigned long
< float < double < long double
```

- you can do explicit conversions like in C using (int), e.g.
- you can also do explicit conversions using C++ operators:
  - static_cast – safe and portable; e.g. c = static_cast<char>(i);
  - reinterpret_cast – system dependent, not good to use
  - const_cast – lets you change a const into a modifiable variable
  - dynamic_cast – used at run-time for casting objects from one class to another (within inheritance hierarchy); this is sort of like Java but can get really messy and is really a more advanced topic...

Branching and Looping

- if, if/else just like C and Java
- while and for and do/while just like C and Java
- break and continue just like C and Java
- switch just like C and Java
- goto just like C (but don’t use it!!!)

Program structure

- just like in C
- program is a collection of functions and declarations
- language is block-structured
- declarations are made at the beginning of a block; allocated on entry to the block and freed when exiting the block
- parameters are call-by-value unless otherwise specified

arrays

- similar to C
- dynamic memory allocation handled using new and delete instead of malloc (and family) and free
- examples:

```
int a[5];
char b[3] = { ’a’, ’b’, ’c’ };
double c[4][5];
int *p = new int(5); // space allocated and *p set to 5
int **q = new int[10]; // space allocated and q = &q[0]
int *r = new int; // space allocated but not initialized
```

Structures

- struct keyword like in C (but you don't need typedef) (last class)
- use dot operator or -> to access members (fields) of a struct or struct *
- C++ allows **functions** to be members, whereas C only allows data members (i.e., fields)
- example

```
struct point {
public:
void print() const { cout << "(" << x "," << y << ")"; }
void set( double u, double v ) { x=u; y=v; }
private:
double x, y;
}
```

Pointers and References

- pointers are like C:
  - int *p means “pointer to int”
  - p = &i means p gets the address of object i. references are not like C!! they are basically aliases – alternative names – for the values stored at the indicated memory locations, e.g.:

```
int n;
int &nn = n;
double a[10];
double &last = a[9];
```

- The difference between them:

```
int a = 5; // declare and define a
int *p = &a; // p points to a
int &refa = a; // alias (reference) for a
*p = 7; // *p points to a, so a is assigned 7
refa = *p + 1; // a is assigned value of *p=7 plus 1
```

I/O Screen

```
// hello world in C++
#include <iostream>
using namespace std;
int main() {
cout << "hello world" << endl;
}
```

- comment characters are // or /* ... */, just like Java
- using namespace is sort of like importing a package in Java; it is used in conjunction with the header declaration
- you could also say #include <iostream.h> and leave out the using namespace std; line; this is an older style of C++ but it still works
- cout << is like System.out.print in Java or like printf() in C
- endl outputs a newline; saying cout << "\n"; does the same thing
  - Advantage is its system dependant

I/O keyboard

- read from the keyboard using cin >>, which is like scanf() in C
- example:

```
#include <iostream>
using namespace std;
int main() {
int i;
cout << "enter a number: ";
cin >> i;
cout << "you entered " << i <<"\n";
}
```

C++ iostream

- two bit-shift operators:
  - << meaning “put to” output stream (“left shift”)
  - >> meaning “get from” input stream (“right shift”)
- three standard streams:
  - cout is standard out
  - cin is standard in
  - cerr is standard error
- the iostream library is “type safe”, so you don’t have to use formatting statements:
  variables are input/output based on their datatype

ostream and istream

- ostream
  - cout is an ostream, << is an operator
  - use cout.put(char c) to write a single char
  - use cout.write(const char *p, int n) to write n chars
  - use cout.flush() to flush the stream
- istream
  - cin is an istream, >> is an operator
  - use cin.get(char &c) to read a single char
  - use cin.get(char *s, int n, char c='\n') to read a line (inputs into string s at most n-1 characters, up to the specified delimiter c or an EOF; a terminating 0 is placed at the end of the input string s)
  - also cin.getline(char *s, int n, char c='\n')
  - use cin.read(char *s, int n) to read a string

Formatted output

- in <iomanip> header file, the following are defined:

- scientific – prints using scientific notation
- left – fills characters to right of value
- right – fills characacers to left of value
- internal – fills characters between sign and value
- setfill(int) – sets fill character
- setw(int) – sets field width
- setprecision(int) – sets floating point precision

Example

- cout << setprecision(3) << 2.34563;

Declaring Class

- Almost like struct, the default privacy specification is private whereas with struct, the default privacy specification is public
- example

```
class point {
double x, y; // implicitly private
public:
void print();
void set( double u, double v );
}
```

- classes can be nested (like java)
- static is like in Java, with some weird subtleties

Using

```
point x;
x.set(3,4);
x.print();

point *pptr = &x;

pptr->set(3,2);
pptr->print();
```

Classes: function overloading and overriding

- overloading:
  - when you use the same name for functions with different signatures
  - functions in derived class supercede any functions in base class with the same name
- overriding:
  - when you change the behavior of base-class function in a derived class
  - DON'T OVERRIDE BASE-CLASS FUNCTIONS!!

- because compiler can invoke wrong version by mistake
- but init() is okay to override
- (more explanation in ch 12...)

Access specifiers

- In class declaration can have:
- Public
  - Anyone can access
- Private
  - Only class members and friends can access

Access specifiers

- public
  - public members
  - can be accessed from any function
- private members
  - can only be accessed by class's own members
  - and by "friends" (see ahead)
- Protected
  - Class members, derived, and friends.

- "access violations" when you don't obey the rules...
- can be listed in any order
- can be repeated

Class scope

- ::
- example:

```
::i // refers to external scope
point::x // refers to class scope
std::count // refers to namespace scope
```

- given previous definition of point, we could do:

```
point p;
p.print();
p.point::print(); // redundant but legal
```

Defining functions

```
void point::print(){
cout << "(" << x "," << y << ")";
}

void point::set( double u, double v )
{ x=u; y=v; }
```

Constructors and destructors

- constructors are called ctors in C++; they take the same name as the class in which they are defined, like in Java
- destructors are called dtors in C++; they take the same name as the class in which they are defined, preceded by a tilde (˜); sort of like finalize in Java
- ctors can be overloaded and can take arguments
- dtors can not
- default constructor has no arguments
- constructor with one argument is a conversion constructor that converts its argument datatype to an object of the class being constructed
- constructor initializer is a special type of constructor that is used to initialize the values of data members of a class

```
class point {
double x,y;
public:
point() { x=0;y=0; } // default
point( double u ) {x =u; y=0; }
// conversion
point( double u, double v )
  { x =u; y =v;}
.
.
.
}
```

usage

point p;

Constructors II

- default constructor (ctor")
- has same name as class it constructs
- in array5.cpp, ctor is used instead of init()
- declare as:

```
class IntArray() {
public:
IntArray();
// etc
}
void IntArray::IntArray() {
numElems = 0;
elems = 0;
} // end of default constructor
```

- invoked when object is allocated: IntArray a;
- but remember that built-in types are not automatically initialized

destructors

- default destructor (“dtor”)
- performs same job as cleanup():

```
class IntArray {
public:
IntArray(); // constructor
˜IntArray(); // destructor
// etc
}
void IntArray::˜IntArray() {
if ( elems != 0 ) free( elems );
}
```

- invoked automatically when object is no longer usable (i.e., when it is popped off the stack, like a local function variable)

ctor and dtor

- chaining
  - constructors and destructors are chained automatically
  - derived class ctors invoke base class constructors and
  - execute in reverse order (lowest base class first)
  - derived class dtors invoke base class dtors and execute in order (derived class first)
- arrays
  - default ctors and dtors are called on each element in the array
- implicit ctors and dtors exist (and are invoked) if you don’t write them explicitly

- ctors and dtors can be private, but typically are public

- never invoke default ctors or dtors explicitly!

```
e.g.: ia.IntArray(); // NO!!!
ia.˜IntArray(); // NO!!
```

Abstraction with member functions

- example #1: array1.cpp
- example #2: array2.cpp
  - array1.cpp with interface functions
- example #3: array3.cpp
  - array2.cpp with member functions
- class definition
- public vs private
- declaring member functions inside/outside class definition
- scope operator (::)
- this pointer

array1.cpp

```
struct IntArray {
  int *elems;
  size_t numElems;
};
main() {
  IntArray powersOf2 = { 0, 0 };
  powersOf2.numElems = 8;
  powersOf2.elems = (int *)malloc( powersOf2.numElems *
   sizeof( int ));
  powersOf2.elems[0] = 1;
  for ( int i=1; i<powersOf2.numElems; i++ ) {
    powersOf2.elems[i] = 2 * powersOf2.elems[i-1];
  }
  cout << "here are the elements:\n";
  for ( int i=0; i<powersOf2.numElems; i++ ) {
    cout << "i=" << i << " powerOf2=" <<
   powersOf2.elems[i] << "\n";
  }
  free( powersOf2.elems );
}
```

array2

```
void IA_init( IntArray *object ) {
  object->numElems = 0;
  object->elems = 0;
} // end of IA_init()

void IA_cleanup( IntArray *object ) {
  free( object->elems );
  object->numElems = 0;
} // end of IA_cleanup()

void IA_setSize( IntArray *object, size_t value ) {
  if ( object->elems != 0 ) {
    free( object->elems );
  }
  object->numElems = value;
  object->elems = (int *)malloc( value * sizeof( int ));
} // end of IA_setSize()

size_t IA_getSize( IntArray *object ) {
  return( object->numElems );
} // end of IA_getSize()
```

Class friends

- allows two or more classes to share private members
- e.g., container and iterator classes
- friendship is not transitive

heirarchy

- composition:
  - creating objects with other objects as members
  - example: array4.cpp
- derivation:
  - defining classes by expanding other classes
  - like “extends” in java
  - example:

```
class SortIntArray : public IntArray {
public:
void sort();
private:
int *sortBuf;
}; // end of class SortIntArray
```

- “base class” (IntArray) and “derived class” (SortIntArray)
- derived class can only access public members of base class

- complete example: array5.cpp
  - public vs private derivation:
- public derivation means that users of the derived class can access the public portions of the base class
- private derivation means that all of the base class is inaccessible to anything outside the derived class
- private is the default

Class derivation

- encapsulation
  - derivation maintains encapsulation
  - i.e., it is better to expand IntArray and add sort() than to modify your own version of IntArray
- friendship
  - not the same as derivation!!
  - example:
- is a friend of
- B2 is a friend of B1
- D1 is derived from B1
- D2 is derived from B2
- B2 has special access to private members of B1 as a friend
- But D2 does not inherit this special access
- nor does B2 get special access to D1 (derived from friend B1)

Derivation and pointer conversion

- derived-class instance is treated like a base-class instance
- but you can’t go the other way
- example:

```
main() {
IntArray ia, *pia;
// base-class object and pointer
StatsIntArray sia, *psia;
// derived-class object and pointer
pia = &sia; // okay: base pointer -> derived object
psia = pia; // no: derived pointer = base pointer
psia = (StatsIntArray *)pia; // sort of okay now since:
// 1. there’s a cast
// 2. pia is really pointing to sia,
// but if it were pointing to ia, then
// this wouldn’t work (as below)
psia = (StatsIntArray *)&ia; // no: because ia isn’t a
   StatsIntArray
```

- danger:
  - don’t point a base class pointer to an array of derived objects!
  - they aren’t the same size!

Next time

- Work on hw
- Will post lab tomorrow night online
- Will post examples

- Do reading:
  - chapters: 7-9,11-13