UMassAmherst

Operating Systems
CMPSCI 377, Lec 2
Intro to C/C++

Prashant Shenoy
University of Massachusetts Amherst

Why C?

- Low-level
  - Direct access to memory
  - WYSIWYG (more or less)
  - Effectively no runtime system
    - No garbage collector
    - No other threads
    - No "read" or "write barriers"
- Efficient
  - Space & time
  - C: effectively portable assembly code

OK, Why C++?

- C++: extends C
  - Upwardly-compatible
- Adds significant software engineering benefits
  - Classes
  - Encapsulation (private)
  - Templates (“generics”)
  - Other modularity advantages
  - Inlining instead of macros

Outline, part I

- Basics – compiling & running
- Intrinsic types, conditionals, etc.
- Pointers + Reference variables
  - Assignment
  - Objects
  - &, *, ->
- Stack vs. heap

Outline, part II

- Functions
  - Parameter passing
- Structs & classes
- Overloading & inheritance
- Stack vs. heap
- I/O, command-line
- STL

Basics

- Main & compilation

Java Hello World (filename: HelloWorldApp.java):

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

C++ Hello World (filename: HelloWorld.cpp):

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

int main (){
    cout << "Hello World!" << endl;
    return 0;
}
```

Java:	C++:
`javac HelloWorldApp.java`	`g++ -o HelloWorld HelloWorld.cpp`
`java HelloWorldApp`	`./HelloWorld`

Intrinsic Types

- Essentially identical

Java:	C++:
`byte myByte;`	`char myByte;`
`short myShort;`	`short myShort;`
`int myInteger;`	`int myInteger;`
`long myLong;`	`long myLong;`
`float myFloat;`	`float myFloat;`
`double myDouble;`	`double myDouble;`
`char myChar;`	`char myChar;`
`boolean myBoolean;`	`bool myBoolean;`

Conditionals

- Mostly the same
  - C/C++: nonzero int same as true

Java:	C++:
`boolean temp = true;`	`bool temp = true;`
`boolean temp2 = false;`	`int i = 1;`
`if (temp)`	`if (temp)`
`  System.out.println("Hello World!");`	`  cout << "Hello World!" << endl;`
`if (temp == true)`	`if (temp == true)`
`  System.out.println("Hello World!");`	`  cout << "Hello World!" << endl;`
`if (temp = true) // Assigns temp to be true`	`if (i)`
`  System.out.println("Hello World!");`	`  cout << "Hello World!" << endl;`

File I/O

- Simple stream-based I/O
  - cout << “foo” print foo
  - cin >> x read x from the console

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

int main(){

  int a;
  ifstream input;

  input.open("myfile");
  while (input >> a);
  input.close();
  input.clear();

  return 0;
}
```

Command-line Arguments

- Again, similar to Java

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

int main(int argc, char * argv[]){

  int a;
  ifstream input;

  if (argc != 3){ // One for the program name and two parameters
     cout << "Invalid usage" << endl;
     return -1;
  }

  int a = atoi (argv[1]);  // First parameter is a number
                           // atoi converts from characters to int

  input.open(argv[2]);  /// Second parameter is the name of a file
  input.close();

  return 0;
}
```

Key Differences

- Differences between C/C++ and Java
  - Assignment
  - Pointers
  - Parameter passing
  - Heap & Stack
  - Arrays

Assignment

- Java assignment: makes reference
- C++ assignment: makes copy

```
SomeClass x, y;
SomeClass *a;

x=y; // This copies object y to x.  Modifying x does NOT modify y.

a=&x;  // This copies a reference to x.
       // Modifying the object a points to modifies x.
```

Pointers & Friends

- "Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backwards from which we may never recover."
  - C.A.R. Hoare

Pointers & Friends

- Concept not in Java: address manipulation

```
int *ptr, *ptr2;
int x = 5;
int y = 4;

ptr = &x;
ptr2 = &y;
```

```
int *ptr;
int x = 5;

ptr = &x;

cout << *ptr << endl;  //prints 5
```

Functions & Parameter Passing

- C/C++ – all parameters copied by default

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

void foo(int i){
  cout << i << endl; // Prints 1
}

int main (){
  foo (1);

  return 0;
}
```

Parameter Passing

- To change input, pass pointer
  - or call by reference

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

void foo(int *i){
  *i = 6;
}

void bar(int i){
  i = 10;
}

int main (){
  int i = 0;

  foo (&i);
  cout << i << endl;  // prints 6

  bar (i);
  cout << i << endl;  // prints 6

  bar (&i);  // WOULD NOT COMPILE

  return 0;
}
```

Pass by Reference

- Syntactic sugar:
  foo (int &i) = pass by reference
  - Secretly does pointer stuff for you

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

void foo(int &i){
  i = 6;
}

int main (){
  int i = 0;

  foo (i);
  cout << i << endl;  // prints 6.

  return 0;
}
```

Stack & Heap

- In C/C++ as in Java, objects can live on:
  - Stack = region of memory for temporaries
    - Stack pointer pushed on function entry
    - Popped on function exit
  - Heap = distinct region of memory for persistent objects
    - C/C++ – explicitly managed
- Pointers introduce problems!

The Stack

- Stack data: new every time

```
int foo (int a){
  int b = 10;

  return 0;
}
```

Big Stack Mistake

- Never return pointers to the stack!

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

// BAD BAD BAD

int* foo (){
  int b = 10;

  return &b;
}

int main(){
  int *a;

  a = foo();

  cout << *a << endl;  // Print out 10?

  return 0;
}
```

The Heap

- Allocate persistent data on heap with new

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

// GOOD GOOD GOOD

int* foo (){
  int *b;

  b = new (int);

  *b = 10;

  return b;
}

int main(){

  int *a;

  a = foo();

  cout << *a << endl;  // Print out 10!

  return 0;
}
```

Explicit Memory Management

- Java heap – garbage collected
- C/C++ – explicit memory management
  - You must delete items (or memory leak)

```
bar *qux;

for (int i=0; i < 1000000; i++){
    qux = new (bar);
    delete (qux);
}
```

  - Delete them too soon (still in use) – crash
    - "Dangling pointer" error
  - Delete something twice – crash
    - "Double-free" error

Classes & Objects

- No “top” object (as in Java Object)
  - Also: C++ has no interfaces but has multiple inheritance – stay far away

Struct Member Access

- struct = class with everything public
  - Use these sparingly

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

struct foo{
  int a;
};

int main (){
  foo b, *c; // b is a struct, c points to a struct

  b.a = 6;
  c = &b;
  c->a = 7; // Remember c points to the struct b!

  cout << b.a << endl;  // prints 7.

  return 0;
}
```

Class Declaration

- Pretty similar

```
class IntCell
{
  public:

  IntCell( int initialValue = 0)
     { storedValue = initialValue; }

  int getValue( )
     { return storedValue; }

  int setValue( int val )
     { storedValue = val; }

  private:

  int storedValue;
};
```

Arrays

- Numerous differences
  - Arrays do not have to be allocated with new
  - Array bounds not checked
  - Item = pointer to start of array
  - Arrays just syntactic sugar for pointer arithmetic! (scary! avoid!)
    - v = 12; *(Item + v) = 1;
    - Same as Item[12] = 1;
  - Note: sizeof(x) = number of bytes to hold x
- Multi-dimensional arrays (matrices)
  - just arrays of pointers to arrays

Other Features

- Operator overloading
  - New meanings to existing operators
    - int operator+(MyType& a, MyType& b);
  - Controversial, but useful for things like complex math, matrix operations
    - int& operator()(int x, int y);
- Templates
  - A.k.a. generics in Java
  - template <class X> void foo (X arg);

Standard Template Library(STL)

- Implements useful data structures

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

queue<int> myQueue;
int main(int argc, char * argv[]){

  myQueue.push(10);
  myQueue.push(11);

  cout << myQueue.front() << endl; // 10

  myQueue.pop();

  cout << myQueue.front() << endl; // 11
  myQueue.pop();

  cout << myQueue.size() << endl; // Zero
}
```

End of Lecture

Classes & Objects

- No “top” object (as in Java Object)
  - Also: C++ has no interfaces but has multiple inheritance – stay far away
- Key difference for you – not all methods dynamically-dispatched
  - Methods associated with declared type rather than dynamic type unless labeled virtual