

CS3157: Advanced Programming

Lecture #12

Apr 3

Shlomo Hershkop
shlomo@cs.columbia.edu

1

Outline

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

2

Main difference between c and cpp

- C's power is driven by functions. You define a set of function which operate in a specific sequence to implement some algorithm
 - Top down
- CPP is an object oriented language
 - Bottom up

3

Compatible

- Cpp is backwards compatible with c
- Cpp is bottom up approach
- Cpp compilers will compile c code


4

Advantages

- There are a bunch of (claimed) advantages to using CPP over c

5

Advantages

- Can create new programs faster because we can reuse code
- Easier to create new data types
- Easier memory management
- Programs should be less bug-prone, as it uses a stricter syntax and type checking.
- `Data hiding', the usage of data by one program part while other program parts cannot access the data
- Will whiten your teeth 

6

Defining c++ functions

- a function's "signature" is its name plus number and type of arguments
- you can have multiple functions with same name, as long as the signatures are different

- example:

```
void foo( int a, char b );
void foo( int a, int b );
void foo( int a );
void foo( double f );
main() {
foo( 1, 'x' );
foo( 1, 2 );
foo( 3 );
foo( 5.79 );
}
```

- OVERLOADING – when function name is used by more than one function

7

C++ Function II

- Foo() or Foo(void) for void arguments
 - Different than c
- Foo(...) for unchecked parameters
 - Look up va_list and va_start
 - A cleaner approach is to pass in an array
- New Trick:
- Foo(int a, int b, int c=10)
 - Foo(4,5,2)
 - Foo(4,5)

8

Function III

- Inline functions
- Function overloading:
 - void foo(int a, char c)
 - void foo(char c)

 - Not allowed
 - void foo(int a)
 - int foo(int a)

9

Other additions

- C++ includes many compiler side additions to help the programmer (yes that is you) to write better code
- Other technical changes (will be pointing them out as we pass them)

10

Void pointers

- C allows you to assign and convert void pointers without casting

- C++ needs a cast

```
void * V;
```

```
..
```

```
Foo *f = (Foo)V;
```

11

NULL

- null pointer (0)

- in c, it's a language macro:

```
#define NULL (void *)0
```

- in c++, it's user defined because otherwise an explicit cast is needed!

```
#define NULL 0
```

- Some books recommends using 0 instead of NULL

12

enums

- Are treated a little differently in c++
- enum day {Sunday, Monday , .. }
- day X = 1; //only works in c
- day X = Sunday;

13

main()

- In C main is the first thing to run
- C++ allows things to run before main, through global variables
 - What is the implications ?
- Variable which are declared outside of main, have global scope (will cover limits).
- Can have function calls here

14

File conventions

- No one convention
 - .C
 - .cc
 - .cp
 - .cpp ← I prefer this
 - .cXX
 - .C++

15

Keywords c++

- asm
- catch
- class
- friend
- delete
- inline
- new
- operator
- private
- protected
- public
- this
- throw
- template
- try
- virtual

16

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

17

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'

18

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

19

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++

20

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
```

21

Conversions II

- you can do explicit conversions like in C using cast
 - (int)something
- 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...

22

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!!!)

23

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

24

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
```

25

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;
}
```

26

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
```

27

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

28

iostream.h

- it's preferred not to use C's stdio (though you can), because it's not "type safe" (i.e., compiler can't tell if you're passing data of the wrong type, as you know from getting run-time errors...)
- stdio functions are not extensible
- note << is left-shift operator, which iostream "overloads"
- you can string multiple <<'s together, e.g.:
- `cout << "hello" << " world" << "\n";`
- `cout` is like `stdout`
- `cerr` is like `stderr`

29

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";
}
```

30

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

31

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

32

Formatted output

- in `<iomanip>` header file, the following are defined:
- `scientific` – prints using scientific notation
- `left` – fills characters to right of value
- `right` – fills characters 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

33

Example

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

34

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

35

Using

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

```
point *pptr = &x;
```

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

36

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...)

37

Access specifiers

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

38

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

39

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

40

Defining functions

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

41

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

42

```
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;}  
  .  
  .  
  .  
}
```

43

usage

```
point p;
```

44