ICSI 403
Design and Analysis of Algorithms

Lecture 02 – C++ Programming

J Marques de Carvalho

History of C++

- BCPL: Basic Combined Programming Language (Cambridge – 1966)
- Originally intended for writing compilers for other languages
- B: A stripped-down, syntactically changed version of BCPL (Bell Labs [AT&T] - 1969)
- C: (Dennis Ritchie – Bell Labs 1969 – 1973)
  - First high-level language used to write a UNIX kernel (had been written in assembly language prior to that)
  - C and UNIX were CLOSELY tied

History of C++ (2)

- Bell Labs made UNIX widely available to colleges and universities for free (for a long time; in 1983, they raised the license fees a lot)
- This made C very popular, as the number of college students working with UNIX skyrocketed.
- Throughout the 1980’s, C was the language all “serious” code was written in.
- Bjarne Stroustrup (1979-1984, Bell Labs) worked on “C with Classes”, which became C++ in 1985
  - See Stroustrup’s text: ISBN 0201700735

History of C++ (3)

- In the late 1980’s, C++ became the de facto “preeminent language” for development.
  - Most universities used C++ as their introduction to programming language
- Java 1.0 (Sun Microsystems,1991-1995): 1995
- As we go forward, recall that you might have *learned* Java first, but C and C++ preceded Java by at least a couple of decades.
- Some things in C/C++ will *seem* like a colossal step backwards.
- Bear in mind: native C/C++ code is blindingly fast

History of C++ (4)

- Java borrowed heavily from C/C++ in terms of syntax, but it broke away (in a big way) with respect to objects
- C++ was “C with objects bolted on”
- Java was designed to be purely O-O from the beginning
- We will begin by covering some concepts that are (pretty much) the same in both languages
- Later, we will look into the bigger differences

Let’s See Some Code

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

int main()
{
  int numberOfLanguages;
  cout << "Hello Reader\nWelcome to C++\n";
  cout << "How many programming languages do you know? ";
  cin >> numberOfLanguages;
  if (numberOfLanguages < 1)
     cout >> "You may need a more elementary text\n";
  else
     cout >> "Enjoy the book";
  return 0;
}
```

Let’s See Some Code

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

int main()
{
  int numberOfLanguages;
  cout << "Hello Reader\nWelcome to C++\n";
  cout << "How many programming languages do you know? ";
  cin >> numberOfLanguages;
  if (numberOfLanguages < 1)
     cout >> "You may need a more elementary text\n";
  else
     cout >> "Enjoy the book";
  return 0;
}
```

`#include` is the C/C++ analog of Java’s `import`. It pulls in library support from outside our code. Any line that starts with a # is a *preprocessor directive*, and will be handled before your code is actually compiled. It pulls in *source code*.

Let’s See Some Code

Anything we `#include` in angle brackets (<>) is a system library. Things we `#include` in double quotes ("") are in the same directory as our source code

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

int main()
{
  int numberOfLanguages;
  cout << "Hello Reader\nWelcome to C++\n";
  cout << "How many programming languages do you know? ";
  cin >> numberOfLanguages;
  if (numberOfLanguages < 1)
     cout >> "You may need a more elementary text\n";
  else
     cout >> "Enjoy the book";
  return 0;
}
```

Let’s See Some Code

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

int main()
{
  int numberOfLanguages;
  cout << "Hello Reader\nWelcome to C++\n";
  cout << "How many programming languages do you know? ";
  cin >> numberOfLanguages;
  if (numberOfLanguages < 1)
     cout >> "You may need a more elementary text\n";
  else
     cout >> "Enjoy the book";
  return 0;
}
```

We’ll talk about namespaces later; for now, just know that this line has to be at the top of your code.

Let’s See Some Code

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

int main()
{
  int numberOfLanguages;
  cout << "Hello Reader\nWelcome to C++\n";
  cout << "How many programming languages do you know? ";
  cin >> numberOfLanguages;
  if (numberOfLanguages < 1)
     cout >> "You may need a more elementary text\n";
  else
     cout >> "Enjoy the book";
  return 0;
}
```

In C++, we don’t *have* to have any classes, but we *do* have to have a function called `main` somewhere. “Function” is more prevalent in C++ documentation than “method”, even when used in the context of objects

Let’s See Some Code

Variable declarations look a lot like they do in Java. We have integers of multiple sizes, as well as floating point of varying sizes. Strings are *not* a primitive type, just as in Java.

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

int main()
{
  int numberOfLanguages;
  cout << "Hello Reader\nWelcome to C++\n";
  cout << "How many programming languages do you know? ";
  cin >> numberOfLanguages;
  if (numberOfLanguages < 1)
     cout >> "You may need a more elementary text\n";
  else
     cout >> "Enjoy the book";
  return 0;
}
```

Let’s See Some Code

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

int main()
{
   int numberOfLanguages;
   cout << "Hello Reader\nWelcome to C++\n";
   cout << "How many programming languages do you know? ";
   cin >> numberOfLanguages;
   if (numberOfLanguages < 1)
      cout >> "You may need a more elementary text\n";
   else
      cout >> "Enjoy the book";
   return 0;
}
```

`cout` is the “Console OUTput” object. We direct output to it with the << operator. Remember: the << points in the direction the data flows – the string flows TO the console in this line.

Let’s See Some Code

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

int main()
{
  int numberOfLanguages;
  cout << "Hello Reader\nWelcome to C++\n";
  cout << "How many programming languages do you know? ";
  cin >> numberOfLanguages;
  if (numberOfLanguages < 1)
     cout >> "You may need a more elementary text\n";
  else
     cout >> "Enjoy the book";
  return 0;
}
```

Similarly, `cin` is the “Console INput” object. We direct input from it with the >> operator. Remember: the >> points in the direction the data flows – the data flows FROM the console TO the variable in this line.

Let’s See Some Code

`if-then-else` works as you would expect:

```
if (condition)
   true-stmt
[else
   false-stmt]
```

brackets [ ] to show that the “else” clause is optional

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

int main()
{
  int numberOfLanguages;
  cout << "Hello Reader\nWelcome to C++\n";
  cout << "How many programming languages do you know? ";
  cin >> numberOfLanguages;
  if (numberOfLanguages < 1)
     cout >> "You may need a more elementary text\n";
  else
     cout >> "Enjoy the book";
  return 0;
}
```

Let’s See Some Code

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

int main()
{
  int numberOfLanguages;
  cout << "Hello Reader\nWelcome to C++\n";
  cout << "How many programming languages do you know? ";
  cin >> numberOfLanguages;
  if (numberOfLanguages < 1)
     cout >> "You may need a more elementary text\n";
  else
     cout >> "Enjoy the book";
  return 0;
}
```

Curly braces are used to open and close blocks of code (the main function). Applies to loops, then- and else-clauses, etc.

Let’s See Some Code

`main` isn’t of type `void`. It *always* returns an `int` (that’s the operating system’s way of knowing if the program ended OK, or if there was a problem. We can check the return code in windows to see what happened.

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

int main()
{
   int numberOfLanguages;
   cout << "Hello Reader\nWelcome to C++\n";
   cout << "How many programming languages do you know? ";
   cin >> numberOfLanguages;
   if (numberOfLanguages < 1)
      cout >> "You may need a more elementary text\n";
   else
      cout >> "Enjoy the book";
   return 0;
}
```

Identifiers

- Identifiers
  - “Identifier” = “name of something”
    - Variables, classes, structures, etc.
  - Identifiers start with a letter or underscore, can contain letters, digits, or underscores.
  - System-defined identifiers usually start with an underscore, so you’re discouraged from starting identifiers with an underscore
  - C++ is case-sensitive, so be careful with your names
  - C++ supports (but doesn’t require) the convention of starting with a lower-case word, and capitalizing any following words in an identifier (`rateOfReturn`)

Identifiers, cont’d

- Reserved words
- These can’t be used as identifier names
- Identifiers containing a double underscore are also reserved.

Identifiers, cont’d

- Variables: must be declared before use
  - Integer types:

`bool`	1 bit, stored in 1 byte
`char`	1 byte
`short` (or `short int`)	2 bytes
`int`	4 bytes
`long` (or `long int`)	4 bytes
`long long`	8 bytes

  - C++ also has *unsigned* integers (all non-negative)
    `unsigned int`, `unsigned short`, etc.
  - `float`, `double`, and `long double` for floating point

Identifiers, cont’d

- Variables: *must be declared* before use
- Variables: *should be initialized* before use (but the compiler doesn’t force it)
- Variables can be initialized as they’re declared, either of two ways:

```
 int x=3, y=4, z=10;

int x, y, z;
x = 3;
y = 4;
z = 10;
```

Assignments

- Assignment statements
  - Just like in Java:
  - `variable = expression;`
  - The value of an assignment is the value that was assigned. Therefore

    ```
    X=4;
    ```

  - Has a value of 4, so
  - `Y = (X = 4);` assigns 4 to both `X` and `Y`

Assignments, cont’d

- Assignment statements
  - Just as in Java, C++ supports the usual arithmetic operators, as well as the combination arithmetic/assignment operators:
  - + (Add) += (add and assign)
  - - (Subtract) -= (subtract and assign)
  - * (Multiply) *= (multiply and assign)
  - / (Divide) /= (divide and assign)
  - % (Modulus) %= (assign modulus)
  - --n and n-- pre- and post-decrement
  - ++n and n++ pre- and post-increment

Assignments, cont'd

- Assignment statements
  - As in Java, you can assign values to "bigger" variable types with no problem:
    - `double = float` (or any of the integers)
    - `long long = long` (or `int` or `short` or `char`)
    - `long = int` (or `short` or `char`)
    - `int = short` (or `char`)
    - `short = char`
  - Going the other way (to a "smaller" variable) requires an explicit cast (type coercion). Example:
    - `int = (int) long;`

Assignments, cont'd

- Assignment statements
  - Boolean variables can accept integer values, but anything that's non-zero becomes 1 (0 stays 0)

- Quotes and strings:
  - Single quotes ('a') are used around single *characters*
  - Double quotes ("hello") are used around *strings*. Even if it contains a single character, if it's in double quotes, it's a string
  - Escape sequences (`\n`, `\"`, `\'`, `\\`, etc.) are used to embed special characters in a string.

Assignments, cont’d

Escape sequences:

Escape sequence	Description	Representation
Simple escape sequences		
\'	single quote	byte 0x27 in ASCII encoding
\"	double quote	byte 0x22 in ASCII encoding
\?	question mark	byte 0x3f in ASCII encoding
\	backslash	byte 0x5c in ASCII encoding
\a	audible bell	byte 0x07 in ASCII encoding
\b	backspace	byte 0x08 in ASCII encoding
\f	form feed - new page	byte 0x0c in ASCII encoding
\n	line feed - new line	byte 0x0a in ASCII encoding
\r	carriage return	byte 0x0d in ASCII encoding
\t	horizontal tab	byte 0x09 in ASCII encoding
\v	vertical tab	byte 0x0b in ASCII encoding

Assignments, cont’d

- Named Constants:
  - Work just like in Java, except:
  - Use `const` instead of `final` on the declaration, and
  - You *must* assign the value on the declaration statement; you cannot assign the value in executable code.
    - const char newline = ‘\n’;
    - const double pi = 3.14159;

Console I/O

- `cout` can accept multiple items in a single statement:

```
cout << "line 1\n";
cout << "line 2\n";
cout << "the result is ";
cout << x;
cout << "\n";
```

- Is equivalent to:

```
cout << "line 1\nline 2\n" <<
    "the result is " << x << "\n";
```

Console I/O (cont’d)

- There are two ways of sending a newline to the console:
  - `cout << “\n”;`
  - `cout << endl;`
  - The general rule of thumb is that if you can append `\n` to a string you’re going to output, do it that way; if you want to output *just* a newline, use `endl`.

- Formatting output with a set number (n) of digits:
  - `cout.setf(ios::fixed);`
  - `cout.precision(n);`

Console I/O (cont’d)

- Formatting output with a set number (n) of digits:
  - `cout.setf(ios::fixed);`
    - Makes cout print floats with a fixed number of decimals.
  - `cout.precision(3);`
    - Sets the number of decimals to be three.
  - double f = 2.5;
  - cout << f;   // will print 2.500

Console I/O (cont’d)

- You can accept multiple values from `cin` in a single statement:

```
cout << “Enter x, y, radius: “;
cin >> x >> y >> r;
```

The system will buffer until it sees a newline and will then start parsing the input. Values can be separated by spaces and/or newlines:

```
9  3 10          9              9   3               9
                 3  10          10                  3
                                                    10
```

Program Style

- C++ supports line comments (//) and block comments (/*…*/)
- There’s nothing like Javadoc comments (/**…*/) in C++

- *Note*: Students tend to under-comment their code. When in doubt, use more comments. Your comments should explain the code to someone who doesn’t already know what it does, and how it does it. That maybe *you* in a year when you look at it again!

Flow of Control

- Boolean operators and expressions:
- Comparison operators:
  - Just like Java:
  - `>`, `<`, `==`, `>=`, `<=`, `!=`
- Logical operators:
  - Just like Java:
  - `&&` (and), `||` (or) and `!` (not)
- Operator Precedence
  - Just like Java – when in doubt, use parentheses!
  - (`>`, `>=`, `<`, `<=`), then (`==`, `!=`), `&&`, `||`
- `true` and `false` are logical constants

Flow of Control (cont'd)

- Just like Java, C++ will short-circuit logical expressions.
  - As soon as it sees true for things OR-ed together, it knows the whole expression must be true, so it quits evaluating the expression
  - As soon as it sees false for things AND-ed together, it knows the whole expression will be false, so it quits evaluating the expression
  - Useful to avoid error conditions:

```
if ((x != 0) && (y / x > 3))  vs
if (y / x > 3) && (x != 0))
```

Flow of Control (cont'd)

- Be aware that any integer can be used in a logical expression. If the integer contains *any* non-zero value, it is taken as `true` (only zero is taken as `false`).
- Inadvertently using an integer variable when you meant to use a Boolean can lead to seemingly strange results.
- Also beware of using `=` when you mean `==` in a condition. Syntactically, it's legal, so the compiler won't complain, but it's probably not what you want!

Flow of Control (cont'd)

- Branching Mechanisms:
- `if-else` works just like Java:

```
if (condition)
    then-statement;
else
    else-statement;
```

- C++ is white-space blind, so the above 4 lines are equivalent to

```
if (condition) then-statement;
          else else-statement;
```

- When the then- and else-statements are both short, this can be easier to read.

Flow of Control (cont’d)

- If we need to execute multiple statements in the “then” and/or “else” clause, we can make a block (i.e., a compound statement) out of the multiple statements with braces:

```
if (condition)
{
   then-statement-1;
   …
   then-statement-n;
}
else
{
  else-statement-1;
  …
  else-statement-m;
 }
```

Flow of Control (cont'd)

- The else clause of an if-then-else is optional
- If statements can be nested
- We normally indent to show nesting
- The "multiway if" can be used to select:

```
if (condition) {statement(s)}
else if (condition2) {statement(s)}
else if (condition3) {statement(s)}
…
else {statement(s)}
```

Flow of Control (cont'd)

- Java's `switch` statement (multiple selection) was taken directly from C++
- Don't forget to use `break` at the end of each `case`, unless you want execution to fall through to the next `case`:

```
switch (vlaue) {
    case(value1): {statement(s)}
    case(value2): {statement(s)}
    …
    default: {statement(s)}
}
```

Flow of Control (cont’d)

- enum types: a special kind of named constant.

  -
    ```
    enum Color { red, green, blue };
     Color r = red;
     switch( r )
     {
      case red : std::cout << “red\n”;
        break;
      case blue : std::cout << “blue\n”;
        break;
      case green : std::cout << “green\n”;
        break;
     }
    ```

Flow of Control (cont’d)

- The conditional operator C++’s (and Java’s) only ternary operator:

  ```
  condition ? value1 : value2
  ```

  If `condition` is `true`, then the value of the whole expression is `value1`, otherwise the value of the whole expression is `value2`.

  ```
  x = (y==1) ? z : q;
  ```

  ```
  if (y==1) x=z;
            else x=q;
  ```

Flow of Control (cont’d)

- Loops:
- Java and C++ both support the same loop structures:
  - `for`
  - `while`
  - `do…while`

- Loops in C++ can be nested, just as in Java

Flow of Control (cont'd)

```
for (initialize;
       continue-check;
       update)
{statement(s)}
```

Make SURE there's no semicolon here!

`initialize` and `update` can consist of multiple expressions, separated by commas:

```
for (x=1,y=0; x<10; x++,y--)
        {statement(s)}
```

Flow of Control (cont’d)

```
while (condition) {statement(s)}
```

- If you put while (condition) on a line by itself, make sure you don’t follow it with a semicolon!

Flow of Control (cont’d)

```
do {statement(s)} while (condition);
```

- Remember that `do…while` checks the condition after the body of the loop after the loop runs, meaning the body of the loop is guaranteed to run at least once
- A `while` loop checks the condition before the body of the loop, so the body of the loop isn’t guaranteed to execute at all.

Flow of Control (cont’d)

- `break` and `continue` can be used to interrupt the flow of a loop.
- `break` will exit the loop. If loops are nested when `break` is encountered, then it will exit only the innermost loop.
- `continue` will skip the remaining statements in the body of a loop and go straight to the bottom of the loop.
- `break` cuts short the entire loop.
- `continue` cuts short the current iteration.

Functions

- Predefined functions:
  - C++ includes a rich library of functions, just as Java includes a rich object library.
  - Functions can return a value; in which case the function is declared as having a type that matches the type of value it returns.
  - Functions can return *no* value; in which case the function is declared as being of type `void`.
  - If you want use a library function, you will have to `#include` the library that contains the function in order to gain access to the function.
    - `#include <cmath>` gets us the `sqrt` function.

Functions

- Sometimes it's hard to know which library a given function resides in:
- `abs` (absolute value of an `int`) is in `cstdlib` (the C Standard Library).
- `labs` (absolute value of a `long int`) is also in `cstdlib`.
- `fabs` (absolute value of a `float`) is in `cmath`
- Why? I don't know.

Functions

- The `exit()` function (in cstdlib) halts execution immediately and returns control to the operating system.
- The argument we give `exit()` is the return value of the program's execution.
- Typically, we return 0 only for successful completion, and use `exit` with some non-zero value to tell the O/S we had a problem.

Functions

- Functions that deal with random numbers.
- `srand()` is used to seed the random number generator. Typically, we give it the time of day, because that's an ever-changing value.
- `rand()` generates a random integer between 0 and `RAND_MAX` (a system-defined value of 32767).

Programmer-defined Functions:

- Unlike with Java, we have to declare our functions with a *function prototype* or *function declaration* (the two terms are synonymous).
- Declarations are placed before any executable code.
- A function prototype is the functions' header, with no code, but with a semicolon after the closing parenthesis.
- It declares our intention to use the function, just as a variable declaration does (except it doesn't reserve any memory).

Prototype Functions:

```
double someFunction( double, int );

int main()
{
    double a = 3.5;
    int b = 2;
    double c;

    c = someFunction( a, b );
}


double someFunction( double x, int y )
{
    return x * y;
}
```

Using prototypes allows code to be organized better and prevents errors being introduced if code is reorganized.

Functions

- You can have a function with no parameters; you simply have to include an empty parameter list `()` in the declaration, the function call, and the function definition.
- Void functions can still have return statements (to leave the function); the return statements just can’t have a value:

```
return;        // OK in a void function
return(0);   // not allowed in a void function
```

Functions

- It’s always a good idea to document (with comments)
- the *preconditions* (what assumptions the function is operating under when it starts), and
- the *postconditions* (in what state the function will leave things) associated with a function we write.

Functions

- `main()` as a function:
  - `main` *is* a function
  - If we omit its return value, some compilers will simply return 0 for us; others will generate either a warning or an error.
  - To be on the safe side, let the last thing in your `main` function be `return (0);`

- C++ allows recursion, which works just like it does in Java.

Functions

- Scope Rules:
- Scoping in C++ works just like in Java:
  - Variables are only “in scope” (visible) in the block within which they are declared.
  - We can create a local loop variable: `for (int i=`…
  - We can create a global variable by declaring it *outside* of all functions.
  - Constants can be global, too, if they’re declared before all other executable code (good for things like PI, SQRT2, etc.)

Functions

- Congratulations!!
- So far, C++ and Java have had more similarities than they have differences.
- Next time, we will get into more of the differences between the two: pointers and how they pass parameters to functions (call by value and call by reference).