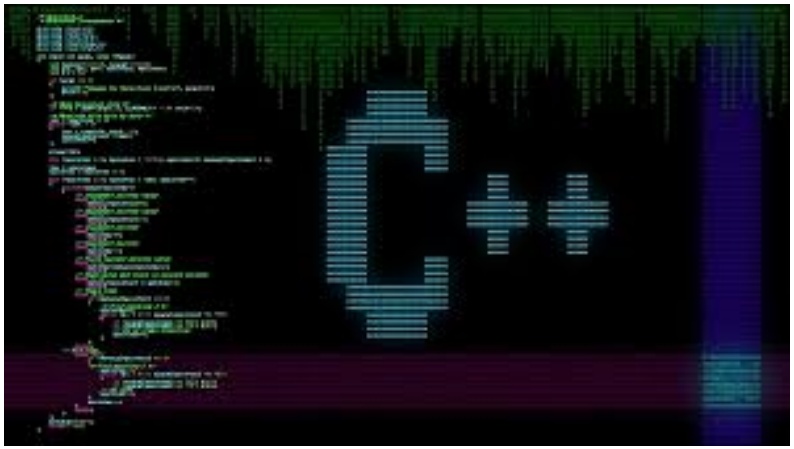
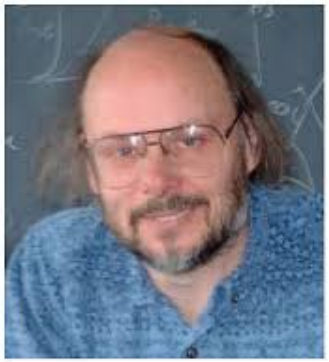
C++

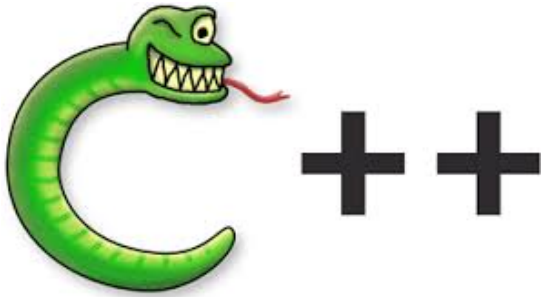
CS 1
Introduction to Computer Programming

Lecture 20: November 20, 2015

C++, part 1







Last time

- Class inheritance
- Raising exceptions using `raise`
- Creating your own exception classes

Today

- A completely new programming language: C++
  - This is preparation for CS 2, which uses C++ exclusively
- Compiling C++ programs
- Comments
- Functions and the `main()` function
- Data types
- Variables
- Operators
- Expressions and statements

Today

- Input/output
- Conditionals: `if`, `else`, `else if`
- `while` and `for` loops

History

- C++ is a descendent of the C language
  - designed at AT&T Bell Labs, 1970s
  - Initially intended as a "systems language" to write operating systems in
  - Now used for much more than this!
  - "Low-level"; allows direct machine access
  - Take CS 11 C track if you're interested

History

- C++ designed by Bjarne Stroustrup at Bell Labs starting in late 70s/early 80s
  - adds object-oriented features to C
  - originally called *"C with Classes"*
  - also adds other features: exceptions, templates, etc.
  - development continues today: C++11, C++14

About C++

- C++ is a desirable language to use when
  - You are writing efficiency-critical software (must run as fast as possible and/or using as little memory as possible)
  - You need extremely precise control over the layout of data in memory and the process of memory allocation/ deallocation
  - You still want to have access to higher-level features like objects, classes, exceptions, *etc.*

Typical C++ applications

- C++ application domains include:
  - Operating systems
  - Video games
  - Computer graphics
  - Web browsers
  - High-frequency trading platforms
  - Embedded systems

Quote

- From Bjarne Stroustrup, creator of C++:

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."

Summing up

- C++ is a large, complex, extremely powerful language with a lot of dangerous corner cases
- Much, *much* more complex than Python!
- C++ programmers really need to understand exactly what they are doing, or bad things can result
  - crashes
  - memory leaks
  - undefined behaviors
  - general confusion

Our approach

- We can't explain all of C++ in 2-3 lectures
- There are many 1000+ page books on C++!
- Primary reference book:
  - "The C++ Programming Language, 4th ed." by Stroustrup
  - 1368 pages!
- There are also the CS 11 C++/Advanced C++ tracks (Donnie Pinkston)
- So… what are *we* doing?

Our approach

- We will cover the bare essentials of C++
- We will then focus on areas known to be problems for CS 2 students:
  - pointers
  - pointer arithmetic
  - memory management with **`new`** and **`delete`**
- We will also try to cover the C++ object-oriented system (in light detail)
- We will contrast C++ with what we know about Python

A simple program

- Here is our first C++ program:

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

int main() {
    cout << "Hello, world!\n";
    return 0;
}
```

A simple program

- This program, when run, will write the string "Hello, world!" to the terminal
- We will explain each line in this program soon
- First, we need to know how to make it into a running program
- We do this by *compiling* the program into an *executable program*
- This executable program consists of instructions in the computer's machine language

A simple program

- Contrast with Python: *there is no interpreter!*
- The only way to make a program run is to *compile* it first to machine language
- We do that as follows (assuming a Linux system):
  - Type the previous program into a text editor
  - Save it as **`hello.cpp`**
  - Compile it into an executable called **`hello`** like this:

```
% g++ hello.cpp -o hello
```

- (The `%` is just the terminal prompt)

A simple program

- g++ is a program called the "GNU C++ compiler"
- The "g" stands for GNU
  - GNU stands for "GNU's Not Unix"
  - GNU software is high-quality free software
  - See http://www.fsf.org for more about GNU

A simple program

- If everything went well, we now have a new program in our directory called **`hello`**
- We can execute it as:

```
% ./hello
```

- and it should print out:

```
Hello, world!
```

- If so, awesome!
- If not, ☹
  - but there should at least be error messages…

Compiling C++ programs

- *Compiling* is the process of taking a C++ program in text form (source code) and turning it into a program in machine language (executable code)
- Our C++ compiler, `g++`, is quite complex and has a large number of command-line options
- Normally, whatever CS2 assignment you will be doing will tell you how to compile the code in detail, so we won't spend too much time on this

Compiling C++ programs

- C++ programs pass through a number of transformations before they become executable programs
- The code you write is *source code*, put in files ending in `.cpp` (for C Plus Plus)
- There is another kind of source code file called *header files* whose names either have no extension or end in `.h`
  - Usually these are part of the C++ standard libraries or are supplied for you

1: Preprocessor

- First, C++ programs go to a *preprocessor* which expands out certain *preprocessor directives*
- In our program, the line

```
#include <iostream>
```

- is a preprocessor directive (`#include`)
- It causes the preprocessor to look for the header file `iostream` in standard locations and include it in the current file
- This will allow us to do input/output (I/O) in our program

2: Compiler

- Then, the preprocessed C++ program goes to the *compiler* proper, which converts the source code into *assembly language* (files end in `.s`)
- Assembly language is a machine-readable form of machine language
- It is extremely low-level
  - Not for the faint-hearted!
- Take CS 24 if you want to get *very* familiar with assembly language ☺

3: Assembler

- Then, the assembly language version of the program goes to an *assembler*, which converts it into *object code* (files end in `.o`)
- Object code is close to machine language, but it is not sufficient to run a program all by itself
  - Some critical system code isn't included in the object code

4: Linker

- Finally, the *linker* takes the object code…
- …adds object code from the system to fill in whatever gaps there may be…
- …and outputs a complete executable program in machine language!
- Files have no particular extension on Unix/Linux/Mac systems
  - On Windows, executables usually end in `.exe`

Summing up

- Compiling C++ programs is complex!
  - *Way* more complex than Python!
- However, `g++` hides almost all the complexity from you
- `g++` will invoke the preprocessor, compiler, assembler, and linker as needed to make your program
- That's all we'll say about compiling C++ programs!

Comments in C++

- C++ has two ways to write comments:
- Comments starting with `//` go to the end of the line

```
// Yes, this is a comment.
```

- Comments starting with `/*` go until a `*/` is encountered, even if it's on a different line
- We will usually use `//` comments (as most C++ programmers do)
- You can't use Python-style `#` comments!

The `main()` function

- Recall our first C++ program:

```
#include <iostream>  // preprocessor cmd
using namespace std;

int main() {     // the main() function
    cout << "Hello, world!\n";
    return 0;
}
```

The `main()` function

- C++ programs are built out of *functions*
  - and also *classes* (discussed later)
- The first function to be executed is the **`main()`** function
- This function must exist in every C++ program
- Once **`main()`** finishes executing, the program is done!

The `main()` function

- Functions in C++ must specify
  - the function name (`main()` here)
  - the return type (`int` here)
  - the function arguments and their types (none here)
  - the *body* of the function (between curly braces)
- The body consists of one or more *statements*
- Simple statements end in semicolons
  - indentation doesn't matter!
- More complex statements (conditionals, loops) have their own special syntax

The `main()` function

- The `main()` function returns an `int`
- This `int` is given to the operating system
- By convention, a return value of `0` from `main()` means "everything completed successfully"
- Any nonzero value means that some kind of error happened
- This explains the line: `return 0;`
- Leaving this line out would be an error
  - no return value in a function returning `int`

Types

- *Major* contrast with Python:
- Have to specify the type of all data, and the compiler checks this before a program ever runs
- If the type checking fails, the compiler will not compile the program!
- We say that C++ is a *"statically-typed"* language
  - Python is *"dynamically-typed"*, which means it doesn't check types until it runs a program
- This can catch a lot of bugs before the code even runs!

Types

- Downside of C++ types: code is somewhat more verbose than Python
- Have to declare:
  - types of variables before using them
  - types of function arguments
  - return type of function
- Leaving out types where needed is a syntax error!

Types

- C++ basic types include:
  - `int` (fixed-length integers)
  - `float` (single-precision floating point numbers)
  - `double` (double-precision floating point numbers)
  - `char` (characters)
  - `bool` (booleans: `true` or `false`)
- More complex types include:
  - arrays (like Python lists, more or less)
  - strings
  - pointers, structs, classes

Variables

- If you want to use a variable in C++, you have to declare it first (including its type!)

```
int i;       // declare i to be an int
int j = 0;   // declare j to be an int == 0
double d = 3.14;
char my_char = 'x'; // single quotes:char
bool my_bool = false;
string s = "foo";  // double quotes: string
```

Variables

- Variables represent a *location in memory* where data can be stored
  - unlike Python! Python variables are just names for data
- Variables can be *initialized* by giving an *initial value* to put in that location in memory

```
int i = 10;  // i location contains 10
```

- Uninitialized variables contain "garbage"
  - Whatever was there in memory before!

```
int j;       // j contains "garbage"
```

Variables

- Tip: if you use the `-Wall` command-line option to `g++`, it will warn you every time you compile code which uses the value of uninitialized variables
- (OK to have uninitialized variables if they are later assigned before use)
- `-Wall` means "enable all warnings"

Operators

- Most operators in C++ are the same as they are in Python
- C++ has some additional operators that Python doesn't have
  - pointer operators (will see later)
  - `++` and `--` to increment/decrement an `int` variable
    - like `+= 1` and `-= 1` in Python
    - prefix: `++i;` postfix: `i++;`
  - `||` and `&&` instead of `and`, `or`
  - `!` instead of `not`
  - Plus a few others…

Operators

- Operators in C++ generally have the same precedence that they have in Python
  - if they exist in Python!
- So `x + y * z` still means `x + (y * z)`
- For new operators, usually you're best off using parentheses or looking up precedences in a table
- 16 different precedence levels!

Operator overloading

- C++ also allows you to overload operators on different data types
- In the simple program, there was the line:

```
cout << "Hello, world!\n";
```

- The usual meaning of the `<<` operator is *"bitwise left shift"*
- Here it has been overloaded to mean *"print to the output stream* `cout`*"*
  - which corresponds to terminal output

Expressions and statements

- Expressions in C++ are things that have a value
  - operator expressions *e.g.* `x + y`
  - function calls
- Statements are things that do something
  - assignment statements
  - function calls (not inside another expression)
  - operator expressions (not inside another expression)
  - `return` and `break` statements
  - conditionals: `if`, `else`
  - loops: `while`, `for`
  - `class` definitions

Expressions and statements

- Simple statements:
  - assignments
  - function calls (not inside another expression)
  - operator expressions (not inside another expression)
  - `return` and `break` statements
- … end in a semicolon
- Contrast with Python: end of line doesn't matter!

The simple program again

- What about this line:

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

int main() {
    cout << "Hello, world!\n";
    return 0;
}
```

The `using` declaration

- The line:

```
using namespace std;
```

- Basically says that unless otherwise specified, functions/data come from the `std` namespace
  - `std` means "standard"
- Without this line we need to write

```
std::cout << "Hello, world!\n";
```

- instead of:

```
cout << "Hello, world!\n";
```

`cout` and `cin`

- `cout` is what is called an "output stream"
- It is a kind of object which lives in the `std` namespace and is used to send output to the terminal
- `cout` means "character output stream"
- For input from the keyboard, you can use the "character input stream" `cin` with the `>>` operator

`cout` and `cin`

- Here's a simple program to demonstrate `cin`:

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

int main() {
    string s;
    cout << "Enter a string: ";
    cin >> s;
    cout << "You entered: " << s << "\n";
    return 0;
}
```

`cout` and `cin`

- Save this into a file called `io.cpp`
- Compile it by typing:

```
% g++ io.cpp -o io
```

- Or (better) with warnings enabled:

```
% g++ -Wall io.cpp -o io
```

- And run it like this:

```
% ./io
```

- It will ask you for a string, then print it back out
- [Demo]

`cout` and `cin`

- Let's look at the `main` function:

```
int main() {
    string s;
    cout << "Enter a string: ";
    cin >> s;
    cout << "You entered: " << s << "\n";
    return 0;
}
```

cout and cin

- Declare a string variable `s`:

```
int main() {
    string s;
    cout << "Enter a string: ";
    cin >> s;
    cout << "You entered: " << s << "\n";
    return 0;
}
```

`cout` and `cin`

- Print the line `"Enter a string: "` to the terminal:

```
int main() {
    string s;
    cout << "Enter a string: ";
    cin >> s;
    cout << "You entered: " << s << "\n";
    return 0;
}
```

cout and cin

- Read a string from the terminal and store it into the string variable **s**
  - by default, only takes up to the first space character

```
int main() {
    string s;
    cout << "Enter a string: ";
    cin >> s;
    cout << "You entered: " << s << "\n";
    return 0;
}
```

`cout` and `cin`

- Print `"You entered: "` to the terminal, followed by the string `s`, followed by a newline character

```
int main() {
    string s;
    cout << "Enter a string: ";
    cin >> s;
    cout << "You entered: " << s << "\n";
    return 0;
}
```

cout and cin

- Note that you can chain the << operator to print multiple values in a single line!

```
int main() {
    string s;
    cout << "Enter a string: ";
    cin >> s;
    cout << "You entered: " << s << "\n";
    return 0;
}
```

`cout` and `cin`

- Return `0` from `main()` and exit the program (successfully)

```
int main() {
    string s;
    cout << "Enter a string: ";
    cin >> s;
    cout << "You entered: " << s << "\n";
    return 0;
}
```

Loops

- C++, like Python, has two basic loop statements: **`for`** and **`while`**
  - There is also a **`do`**/**`while`** loop which is slightly different but rarely used
- The **`while`** loop in C++ is almost identical to Python's **`while`** loop except for syntax
- The **`for`** loop in C++ is completely different from Python's **`for`** loop!
  - Though identical to Java's **`for`** loop, if you know Java…
- We will explore this by writing a simple function to compute powers of integers

Loops

- Here is our function:

```
// Raise b to the power of e.
// Assume e >= 0.
int power(int b, int e) {
    int result = 1;
    while (e > 0) {
        result *= b;
        e -= 1;
    }
    return result;
}
```

Loops

- Function name:

```
// Raise b to the power of e.
// Assume e >= 0.
int power(int b, int e) {
    int result = 1;
    while (e > 0) {
        result *= b;
        e -= 1;
    }
    return result;
}
```

Loops

- Function return type:

```
// Raise b to the power of e.
// Assume e >= 0.
int power(int b, int e) {
    int result = 1;
    while (e > 0) {
        result *= b;
        e -= 1;
    }
    return result;
}
```

Loops

- Function argument list (note types!):

```
// Raise b to the power of e.
// Assume e >= 0.
int power(int b, int e) {
    int result = 1;
    while (e > 0) {
        result *= b;
        e -= 1;
    }
    return result;
}
```

Loops

- **`while`** loop (note syntax!):

```
// Raise b to the power of e.
// Assume e >= 0.
int power(int b, int e) {
    int result = 1;
    while (e > 0) {
        result *= b;
        e -= 1;
    }
    return result;
}
```

`while` loop

- **`while`** loop has the test in parentheses:

```
while (e > 0) {
      result *= b;
      e -= 1;
 }
```

`while` loop

- **`while`** loop has the body in curly braces:

```
while (e > 0) {
    result *= b;
    e -= 1;
}
```

- Sequences of statements inside curly braces are called a *block*
- Indentation doesn't matter

Loops

- We can use our function in a program like this:

```
#include <iostream>
using namespace std;
// Definition of power() left out.
int main() {
    int i, n;
    cout << "Enter a positive integer (base): ";
    cin >> i;
    cout << "Enter a positive integer (exponent): ";
    cin >> n;
    int p = power(i, n);
    cout << i << " to the power of "
         << n << " = " << p << "\n";
    return 0;
}
```

`for` loop

- We'll re-write our function to use a **`for`** loop:

```
// Raise b to the power of e.
// Assume e >= 0.
int power(int b, int e) {
    int result = 1;
    for (int i = e; i > 0; i--) {
        result *= b;
    }
    return result;
}
```

`for` loop

- Initialize the loop variable `i`:

```
for (int i = e; i > 0; i--) {
    result *= b;
}
```

- Note that you can declare types here too (`int i`)
- This means `i` can only be used inside the `for` loop

`for` loop

- Test condition:

```
for (int i = e; i > 0; i--) {
    result *= b;
}
```

- Continue as long as the test condition is true

`for` loop

- Loop variable modification:

```
for (int i = e; i > 0; i--) {
    result *= b;
}
```

- This gets done after the body of the loop executes, each iteration

`for` loop

- Loop body:

```
for (int i = e; i > 0; i--) {
    result *= b;
}
```

- Statements surrounded by curly braces (a block)

`if` and `else`

- We still haven't discussed how to do conditionals (`if` and `else`)
- C++ conditionals are identical to Python's except for syntax
  - and identical to Java's
- One nice thing: no `elif` keyword!
  - use `else if` instead
- We'll rewrite our `power` function one last time to show how to use `if` and `else`

`if` and `else`

- Here we go:

```
// Raise b to the power of e.
int power(int b, int e) {
    int result;
    if (e < 0) {
        cout << "invalid exponent: " << e << "\n";
        result = 0;  // or: return 0;
    } else {
        for (int i = e; i > 0; i--) {
            result *= b;
        }
    }
    return result;
}
```

`if` and `else`

- Test condition in parentheses:

```
if (e < 0) {
    cout << "invalid exponent: " << e << "\n";
    result = 0;
} else {
    for (int i = e; i > 0; i--) {
        result *= b;
    }
}
```

`if` and `else`

- Block in case test is true:

```
if (e < 0) {
    cout << "invalid exponent: " << e << "\n";
    result = 0;
} else {
    for (int i = e; i > 0; i--) {
        result *= b;
    }
}
```

`if` and `else`

- Block in case test is false:

```
if (e < 0) {
    cout << "invalid exponent: " << e << "\n";
    result = 0;
} else {
    for (int i = e; i > 0; i--) {
        result *= b;
    }
}
```

Next time

- Now you know the most absolute basic features of C++
- We'll get back to C++ in about a week (after Thanksgiving):
  - arrays
  - pointers
  - memory management
- Next lecture: Odds and ends, part 2