CS2124

Basics of C++

Topics

- Where to find more information
- Who created C++?
- Simple C++ program
- Hello world - writing to standard output
- Variables and types
- Getting data - reading standard input
- Conditions
- Logical operators
- Loops
- Collections: vectors and strings
- File I/O

What do you *need* to know?

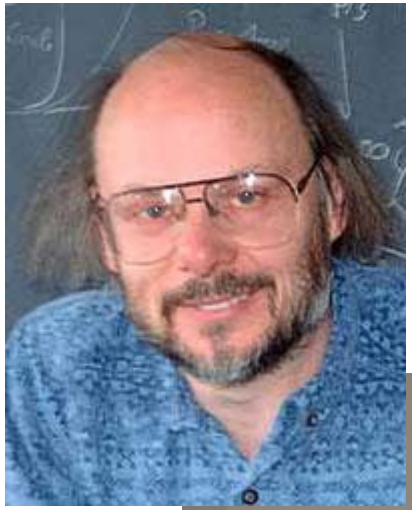
- What does the simplest program look like?
- Hello world
- Defining variables
- Getting input from the console
- Conditions
- Loops
- Collections
- File I/O

Where to find more...

- http://cis.poly.edu/jsterling/cs2124/LectureNotes/01.Intro.html
- http://www.cplusplus.com

Who created C++?

- Bjarne Stroustrup
- AT&T Bell Labs
- Later:
  - University of Texas A&M
  - Morgan Stanley
  - Columbia University (visiting)
- Inspired by C and Simula
- http://www.stroustrup.com/



Simplest Program

```
int main() {
    return 0;
}
```

- Every program *must* have a function called **main**
- Blocks of code are surrounded by *braces*: **{}**
- Statements end with a semi-colon: **;**
- Every function *must* state what type it returns, here **int**, an integer
- And main *must* return an integer. The value zero means successful completion
- If a function returns anything, then it *must* have a return statement.

Simpler-est Program

```
int main() {
    //return 0;
}
```

- Actually, every function *other than* `main` that returns something must have a return statement,
- If main does not have a return statement then it will return 0. Stroustrup was being nice to us.
- Note that a comment begins with `//`

Hello CS2124!

```
#include <iostream>

int main() {
    std::cout << "Hello CS2124!\n";
}
```

- **<<** is the output operator.
  - The target stream appears on the left, and what is printed is on the right
  - The angle brackets “point to” the output stream
- **std::cout** represents standard output, i.e. by default, the screen
- String *literals* are always surrounded by *double* quotes: **" "**
- **#include** specifies a library for compiler.
- iostream provide the definition for std::cout, among other things.

Hello CS2124! (take 2)

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

int main() {
    // std::cout << "Hello CS2124!\n";
    cout << "Hello CS2124!\n";
}
```

- using namespace std;
  - Allows us to refer to cout and many other symbols without having to type std::

Hello CS2124! (take 3)

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

int main() {
    // cout << "Hello CS2124!\n";
    cout << "Hello CS2124!" << endl;
}
```

- output can be *chained*
- endl: an end-of-line character.
  - Perhaps easier to type than: `"\n"`
  - Also “flushes” the output stream. That’s not crucial for us right now.

One key difference between C++ and Python?

- This is just a start, but…
- Every variable is declared to have a **type** and can only hold things of that **type**
  - int n;
  - double d;
  - string s;
- This is also true of
  - function parameters
  - And function return types
- *Many* other languages are like this, such as C and Java and C#.

Variables

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

int main() {
    int x = 42;
    double d = 3.14159;
    string s = "the cat in the hat";
    cout << "x: " << x <<  ", d: " << d << ", s: " << s << endl;
}
```

- Every variable is declared to *have a type* and can *only* hold things of that type
- This is also true of function parameters and function return types
- *Many* other languages are like this, such as C and Java and C#.
- We needed the include for string because strings are **not** primitive / built-in

Uninitialized?

```
int main() {
    int x;
    cout << "x: " << x << endl;
}
```

- What would the output be if we *don't* initialize an integer?
- When I ran this on the Mac, x was zero.
- But **no guarantees**!!! This behavior is **undefined**.
- Your compiler *might* be nice and warn you.
- (I am leaving out the #include's to save space on the slide.)

Uninitialized (2)

```
void foo() {
    int a = 17;
    cout << "a: " << a << endl;
}

void bar() {
    int b;  // Not initialized!!!
    cout << "b: " << b << endl;
}

int main() {
    foo();
    bar();
}
```

```
a: 17
b: 17
```

- b is using the same memory location as a
- **Your mileage may vary**!! (undefined behavior)

Getting input

```
int main() {
    int x = -1;
    cout << "x: " << x << endl;
    cout << "input an integer value: ";
    cin >> x;
    cout << "x: " << x << endl;
}
```

- **`cin`** is standard input, i.e. by default the keyboard
- Notice that the angle brackets for input “point” from the stream to the variable.

Conditions

```
int main() {
    int x;
    cout << "x? ";
    cin >> x;
    if (x == 6) {
        cout << "x is a small perfect number\n";
    }
    else if (x == 42) {
        cout << "x is the answer\n";
    }
    else {
        cout << "x is something else\n";
    }
}
```

- **if**, **else if**, and **else**
- The condition goes inside parentheses
- The code to handle the condition goes in a block.

Logical Operations

```
int main() {
    int x;
    cout << "x? ";
    cin >> x;
    if (x == 6 || x == 28) {
        cout << "x is a small perfect number\n";
    }
    else if (x >= 0 && x <= 9) { // Note, no: 0 <= x < 10
        cout << "x is an imperfect single digit number\n";
    }
    else if (!(x < 0 || x > 99)) {
        cout << "x is a two digit number\n";
    }
    else {
        cout << "x is something else\n";
    }
}
```

- and: **&&**
- or: **||**
- not: **!**

Loops: while

```
int main() {
    int x = 10;
    while (x > 0) {
        cout << x << ' ';
        --x;
    }
    cout << endl;
}
```

- Condition must be in parentheses
- Code for the loop goes in a block
- Here we are using the pre-decrement operator.
  In Python we would write: `x -= 1`

Loops: for

```
int main() {
    for (int x = 10; x > 0; --x) {
        if (x == 5) continue;  // yes, C++ has continue and break
        cout << x << ' ';
    }
    cout << endl;
}
```

- for loop has three parts within the parentheses, separated by semicolons
  - Initialization. Here x is being *initialized to* 10
  - Test for whether to enter the body of the loop. Here, only if x is greater than 0
  - Update. Here, we just need to decrement x
- Often more concise than a while loop.
- Also, note that the ***scope*** of x is limited to the for loop

Loops: do-while

```
int main() {
    int x = 10;
    do {
        cout << x << ' ';
        --x;
    } while (x > 0); // Remember the semicolon
    cout << endl;
}
```

- Always executes the body of the loop at least once.
- Consider what would happen if x started out as -1
- When you want this, it is very nice to have,
  but you will more often use the while and for loops.

Collections: vector

```
int main() {
    vector<int> v; // v can only hold integers
    cout << "v.size(): " << v.size() << endl;

    v.push_back(17);
    v.push_back(42);
    cout << "v.size(): " << v.size() << endl;

    for (size_t i = 0; i < v.size(); ++i) {
        cout << v[i] << ' ';
    }
    cout << endl;

    v.clear();
    cout << "v.size(): " << v.size() << endl;
}
```

- #include <vector>
- *Similar to* Python's list and Java's ArrayList
- *Generic* type
- Other handy methods: back(), pop_back(), capacity()

vector initialization

```
int main() {

    // Initialize the size of the vector and the value of that all the
    // entries will start with.
    // The vector v1 will be of size 7, with every entry equal to 42
    vector<int> v1(7, 42); // parentheses

    // specify the distinct values to initialize each entry to
    vector<int> v2{1, 1, 2, 3, 5, 8, 13, 21};  // curly braces

}
```

ranged for (1)

```
int main() {
    vector<int> v;

    ...

    // for (size_t i = 0; i < v.size(); ++i) {
    //     cout << v[i] << ' ';
    // }

    for (int value : v) {
        cout << value << ' ';
    }
    cout << endl;

    ...
}
```

- Don’t need the i?
- Then easier to use a “ranged for”
- Similar to Python’s for
- Requires C++11

ranged for (2)

```
int main() {
    vector<int> v{2, 3, 5, 7, 11};

    for (int value : v) {
        value = 42;
    }

    for (int value : v) {
        cout << value << ' ';
    }
    cout << endl;
}
```

- What gets printed?
- Not 42's…
- We will see how to fix this next time

strings

- #include <string>
- String **literals** use *double* quotes
  - string s = "this is a string"
- A string variable acts very much like a vector of characters
  - push_back, pop_back, size, clear
- with some additional useful features.
  - E.g. can input or output a string, but not a vector
- Note that characters are a separate type from strings.
  A character literal uses *single* quotes.
  - `char c = 'q';`

string example(1)

```
int main() {
    string s1 = "Twas brillig and";
    cout << s1 << endl;

    for (char c : s1) {
        cout << c << ' ';
    }
    cout << endl;

    string s2 = " the slithy toves";

    string s3 = s1 + s2;
    cout << s3 << endl;
}
```

- #include <string>
- string literals are in double quotes. char literals use single quotes.
- The type char holds a character
- char is a kind of integer
- Strings can be output with the << operator
- Strings can be looped over the same as vectors
- The plus operator concatenates strings

strings are *mutable*!

```
int main() {
    string animal = "bat";

    // What is the type of animal[0]?
    animal[0] += 1;

    // What will be output?
    cout << animal << endl;
}
```

cat

- The elements of a string are of type char.
- chars are ints so we can do arithmetic with them
- chars hold the ascii value of the character
- strings are *mutable*. We can modify their contents

File I/O

- **#include <fstream>**
- Opening and closing
- Testing for successful open
- Reading input
- Looping over a file

Open, test for success, read, close

```
int main() {
    ifstream jab("jabberwocky");
    if (!jab) {
        cerr << "failed to open jabberwocky";
        exit(1);
    }

    string something;
    jab >> something;
    cout << something << endl;

    jab.close();
}
```

Twas brillig and the slithey toves
did gyre and gymble in the wabe.
All mimsy were the borogroves
and the momeraths outgrabe.

Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the JubJub bird and shun
The frumious Bandersnatch!

...

Twas

Open, test for success, read a line, close

```
int main() {
    ifstream jab("jabberwocky");
    if (!jab) {
        cerr << "failed to open jabberwocky";
        exit(1);
    }

    string something;
    getline(jab, something);
    cout << something << endl;

    jab.close();
}
```

Twas brillig and the slithey toves
did gyre and gymble in the wabe.
All mimsy were the borogroves
and the momeraths outgrabe.

Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the JubJub bird and shun
The frumious Bandersnatch!

...

Twas brillig and the slithey toves

Read token by token

```
int main() {
    ifstream jab("jabberwocky");
    if (!jab) {
        cerr << "failed to open jabberwocky";
        exit(1);
    }

    string something;
    while (jab >> something) {
        cout << something << endl;
    }

    jab.close();
}
```

- We put the read into the while condition
- Magically the loop stops when there is nothing more to read
- Each read gets the next ***whitespace delimited*** token.

Vector of lines

```
int main() {
    ifstream jab("jabberwocky");
    string line;
    vector<string> lines;

    while (getline(jab, line)) {
        lines.push_back(line);
    }
    jab.close();
    // print the contents of the vector
    for (size_t i = 0; i < lines.size(); ++i) {
        cout << lines[i] << endl;
    }
    // print them in reverse?
    for (size_t i = lines.size(); i > 0 ; --i) {
        cout << lines[i-1] << endl;
    }
}
```

- Omitting test of open to save space
- Close the file as soon as done with it
- The while loop continues so long as getline is successful

Vector of strings as 2D data structure

```
int main() {
    ifstream jab("jabberwocky");
    string line;
    vector<string> lines;

    while (getline(jab, line)) {
        lines.push_back(line);
    }
    jab.close();

    for (size_t i = 0; i < lines.size(); ++i) {
        for (size_t j = 0; j < lines[i].size(); ++j) {
            cout << lines[i][j] << ' ';
        }
        cout << endl;
    }
}
```

Vector of vectors as 2-dimensional data structure

```
int main() {
    const int ROWS = 10;
    const int COLS = 10;
    vector<vector<int>> mat;

    for (int r = 0; r < ROWS; ++r) {
        mat.push_back(vector<int>(COLS));
        for (int c = 0; c < COLS; ++c) {
            mat[r][c] = r * ROWS + c;
        }
    }

    for (int r = 0; r < ROWS; ++r) {
        for (int c = 0; c < COLS; ++c) {
            cout << mat[r][c] << ' ';
        }
        cout << endl;
    }
}
```

- Filling 2d “matrix” with values from 0 to 99
- Constants are commonly in all caps.
- Note creation of temporary vector being pushed onto `mat`.

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

Other topics to add

- Type **incompatibility** issues
  - Adding a string and an int
  - Assigning a string to an int
  - Reading into an int when there is no integer immediately ahead
  - ...