# C++ Notes

# Basics

**Most basic empty program:**

```
int main() {
}
```

**Compiling and executing is similar to C**

- g++ <src_name>.cpp -o <exec_name>
- ./<exec_name>

**Things that are the same as Java**

- Comments
- Conditional Statements
- Boolean Operators (BUT && can be “and” , || can be “or” )
- Variable declarations (with no modifiers)
- For loops

# Namespace Notation

Many things in C++ follow the **“namespace::identifier”** notation

For example, the **string type** is **std::string**

- The namespace holds identifiers. For example, `std` (C++ standard library)
- "::" is the scope operator
- The identifier is the name of  the function/class/stream etc

The compiler *looks for the identifier in the namespace*

## Printing

```
#include <iostream>
int main() {
   std::cout << "Hello World!\n" << 45 << “\n” << ‘A’;
}
```

- `cout` = character output stream
- << = Insertion operator
- There is no ‘inherent’ new line, you need a “\n” each time to be like Java’s …println
- Variables are often printed like so: `std::cout << <var_name> << "\n";`

# Getting Input

```
std::cin >> <var_name>;
```

`cin` = Character input stream

>> = extraction operator

## Vector Type

| Directive | Declaration |
| --- | --- |
| `#include <vector>` | `std::vector<double> nums;`<br>`std::vector<double> nums(2);`<br>`std::vector<double> nums = {3, 6, 9};` |
| **Access** | **Methods** |
| `vowels[0]` | `.push_back(<val>) // Add`<br>`.pop_back() // Remove (no return)`<br>`.size()` |

# Functions

Wednesday, November 20, 2019 3:19 PM

# Declaration

Without declaration, a function has to come before main

With declaration, it can go anywhere

**Example Declaration:**

```
int max(int num1, int num2);
```

You can have many functions in a CPP file, then you can use them just by using their declarations in other CPP files

Groups of declarations that are re-used go in header files.

To use your own header file, it must be included with quotes

**Default arguments**

- `void intro(std::string name, std::string lang = "C++");`
- Allows for:
  - `intro("Mariel");`
  - `intro("Mariel", "Python");`
- Beware, if the first has a default only, you'll always need 2 arguments (because first pos = first arg)

## Example Definition

```
bool is_palindrome(std::string text)
{
    // Get text as char vector
    std::vector<char> cv(text.begin(), text.end());

    // Get reversed version of string
    std::vector<char> rev;
    for(int i = cv.size() - 1; i >= 0; i--)
    {
            rev.push_back(cv[i]);
    }

    // Compare strings
    bool res;
    for(int i = 0; i < cv.size(); i++)
    {
        res = cv[i] == rev[i];
    }

    // Return result
    return res;
}
```

# Math Functions

```
#include <cmath>

Pow(base, exp);

Sqrt(num);
```

# Inline Functions

Using inline advises the compiler to *insert the function's body where the function call is*.

It sometimes helps with execution speed (and sometimes hinders execution speed).

If you do use it, we recommend testing how it affects the execution speed of your program.

You should ALWAYS add the inline keyword if you are inlining functions in a header (unless you are dealing with member functions, which are automatically inlined for you).

# Function Templates

Created in header files

Slow down compile time, but speed up execution time

**Example:**

```
template <typename T>
T simple(T item) {
  return item;
}
```

# Classes

Friday, February 7, 2020 12:28 PM

Song.hpp

```
// Header files = Class declarations
#include <string>

class Song {

// By default, everything in a class is private, meaning class members
are limited to the scope of the class
private:
    // Attributes (aka member data) hold info about an instance.
    std::string title;
    std::string artist;

// public makes everything below accessible outside the class
public:
    // Constructor
    Song(std::string new_title, std::string new_artist);

    // Destructor
    Song();

    // Methods (aka member functions)
    std::string get_title();
    std::string get_artist();
};
```

Song.cpp

```
// Source/CPP file = Class definitions
#include "song.hpp"
#include <iostream>

// Constructor
Song::Song(std::string new_title, std::string new_artist) {
      title = new_title;
      artist = new_artist;
}

// Destructor
Song::~Song() {
    std::cout << "Goodbye Drama!";
}

// Method Definitions
std::string Song::get_title() {
    return title;
}

std::string Song::get_artist() {
    return artist;
}
```

## Usage

```cpp
#include <iostream>

// We include the class's header
#include "song.hpp"

int main() {

    // Instance creation and use
    Song back_to_black("Back to Black", "Amy Winehouse");
    std::cout << back_to_black.get_title();
}
```

# Reference Variables

A reference variable is an alias for an already existing variable

```
int soda = 99;
int &pop = soda;
// Pop is now a reference to soda
```

Anything we do to the reference also happens to the original. Aliases cannot be changed to alias something else.

## Pass by Value

- We did this when we pass parameters to a function normally
- Uses normal variables
- Because the variables passed into the function are out of scope, we can't actually modify the value of the arguments.

## Pass by Reference

- We pass parameters to a function using references.
- The function can modify the value of the arguments using the references passed in
- We avoid making copies of the arguments, improving performance
- Example of Pass by Reference Function:

```
void swap_num(int &i, int &j) {
    int temp = i;
    i = j;
    j = temp;
}
```

- To ensure the parameter won't be changed, we use "const". his saves the computational cost of making a copy of the argument

```
int triple(int const &i) {
    return i * 3;
}
```

# Address Of Operator

When & is not used in a declaration, it is an address of operator.

**Printing a memory address:**

```
int num = 3;
std::cout << &num << "\n";
```

## Pointers store memory addresses

```
// A variable with type “pointer to int”, initialized to empty pointer
int* number = nullptr;

// Giving it a value:
int gum = 3;
number = &gum;

// Printing
std::cout << ptr;

// Dereferencing (accessing the data the pointer points to)
std::cout << *ptr;
```