Agenda

- The main body and cout
- Fundamental data types
- Declarations and definitions
- Control structures
- References, pass-by-value vs pass-by-references

The main body and cout

C++ IS AN OO EXTENSION OF C

C++ HAS BOTH PROCEDURAL FEATURES AND OBJECT-ORIENTED FEATURES

Every C++ program must have a main()


```
// hw.cpp
#include <iostream>
using namespace std;

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

cout is pretty easy to use


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

int main() {
    string my_name        = "David Blaine";
    string my_text_editor = "Emacs";
    string my_home_os     = "Windows";

    cout << "My name is " << my_name << endl
         << "I was able to install and test g++ and "
         << "the text editor " << my_text_editor << '\n'
         << "in my home computer/laptop, which runs "
         << my_home_os << endl;
    return 0;
}
```

Fundamental data types

THEY ARE WHAT YOU EXPECT THEM TO BE, SIMILAR TO THOSE IN JAVA

The fundamental data types

- bool : true or false
- char : a character
- int : an integer
- float : a floating-point real number

- And cout will output a variable with the appropriate format for the above types

Cout is easy to use on basic types


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

int main() {
    string name = "David Blaine";
    char c      = 'H';
    int i       = 12345;
    bool smart = true;
    double avg = 3.5;

    cout << "I am " << name << endl
         << "smart = " << smart << endl
         << "c = " << c << endl
         << "i = " << i << endl
         << "avg = " << avg << endl;
    return 0;
}
```

Declarations and Definitions

EVERY NAME MUST BE DECLARED BEFORE USED

THERE MUST ALWAYS BE AT MOST ONE DEFINITION FOR EACH NAMED ENTITY

THERE CAN BE MANY DECLARATIONS

MANY DECLARATIONS ARE ALSO DEFINITIONS

Declarations


```
// variable declarations
string name; char c; int i;
bool smart; double avg;

// a type declaration (the type is a struct)
struct Date;

// a const bool declaration
const bool i_am_smart = true;

// a function declaration
int foo(int);
```

Definitions


```
// definition of function foo()
int foo(int x) {
    return x*x;
}

// definition of struct Date
struct Date {
    int d;
    int m;
    int y;
};
```

Many declarations are also definitions


```
// these declarations are *also* definitions
string name;
char c;
int i;
bool smart;
double avg;

// these declarations are *not* definitions
int foo(int);
struct Date;
extern int age;
typedef vector<int> Int_Vector;
```

Control structures

SIMILAR TO JAVA

THE ONES WE'LL OFTEN USE ARE

- IF ELSE
- WHILE LOOP
- FOR LOOP
- SWITCH STATEMENT
- CONTINUE AND BREAK
- EXIT(.) FUNCTION

Let's Reverse a String

- For each line the user types
  - Prints a copy of the line with all characters in reversed order
  - Prints a copy of the line with all words in reversed order
- Illustrates
  - Modularization
  - Functions and loops

Pointers, References, Arrays

- A POINTER IS AN ADDRESS
- A REFERENCE IS AN ALIAS FOR AN EXISTING OBJECT
- AN ARRAY NAME IS A CONST POINTER TO ITS FIRST ELEMENT

References

- A reference is an alternative name for an object

```
int i = 1;
int& r = i; // a reference must always be initialized
int x = r;  // x = i, which is 1
r = 2;      // now both r and i are 2, but x is still 1
```

- Once refering to an object, a reference can't be reassigned to refer to another object
- Main question is, why would one wants to do this?
  - Pass-by-reference semantic
  - Return a reference (later)

Default argument passing semantic: pass by value


```
void foo(int a, int b) {
    a = a + 10;
    b = b + 20;
    cout << "In foo, a = " << a << " and b = " << b << endl;
    // it prints In foo, a = 11 and b = 22
}

int main() {
    int a = 1, b = 2;
    foo(a,b);
    cout << "In main, a = " << a << " and b = " << b << endl;
    // it prints in main, a = 1 and b = 2
    return 0;
}
```

Swap() like this does not work


```
/*
 * the intended swap does not work
 */
void swap(int a, int b) {
    int temp;
    temp = a; a = b; b = temp;
    cout << "In foo, a = " << a << " and b = " << b << endl;
    // it prints: In foo, a = 2 and b = 1
}

int main() {
    int a = 1, b = 2;
    swap(a,b);
    cout << "In main, a = " << a << " and b = " << b << endl;
    // it prints: In main, a = 1 and b = 2
    return 0;
}
```

Swap() with pass-by-reference works!


```
// this 'swap' works as intended
void swap(int& a, int& b) {
    int temp;
    temp = a; a = b; b = temp;
    cout << "In foo, a = " << a << " and b = " << b << endl;
    // it prints: In foo, a = 2 and b = 1
}

int main() {
    int a = 1, b = 2;
    swap(a,b);
    cout << "In main, a = " << a << " and b = " << b << endl;
    // it prints: In main, a = 2 and b = 1
    return 0;
}
```

When to use references?

- When we want the function to modify the arguments
  - However, the name of the function has to give a very strong hint that this is the intention! (swap is a good name, foo is not)
- When the arguments are large objects
  - Save space and time
  - But if no modification is intended, put a const in front