Session 0: C++ essentials

C++ is a compiled, statically typed, object-oriented programming language

- Programs must be translated from the human-readable programming language to binary machine code before running
    - Generally faster than alternatives, since translation is not done on the fly
    - Alternatives:
        - Just-in-time (JIT) compiled: compiled upon running program first, then never thereafter
        - Interpreted: computer runs code line-by-line every time
- All variable datatypes are known at compile time
    - Faster than dynamic typing, since computer does not waste time determining datatypes
- Datatypes can be combined into larger structures, possibly with accompanying special functions
    - Classes and structs

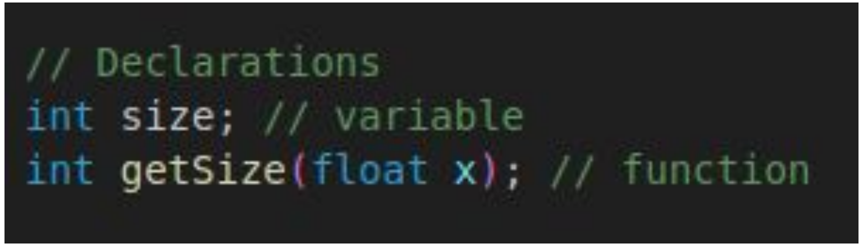
Outline from here

- Typing and declarations vs. definitions
- Compilation process and file organization
- Basic syntax
- Pointers and references
- Arrays and the std::vector
- The main function

Static typing means we have to specify the datatype of EVERYTHING

- Creating a variable, function, or other construct is called a **declaration**
- This tells the compiler what the type of that variable is, or in the case of functions, what the types of the inputs and return value are
  - Compiler needs to know this so that it can allocate enough memory for it
- Don't need to set anything to declare

Here I declare a variable called "size" that is an integer. I also declare a function called "getSize" that takes a float as an input and returns an integer.

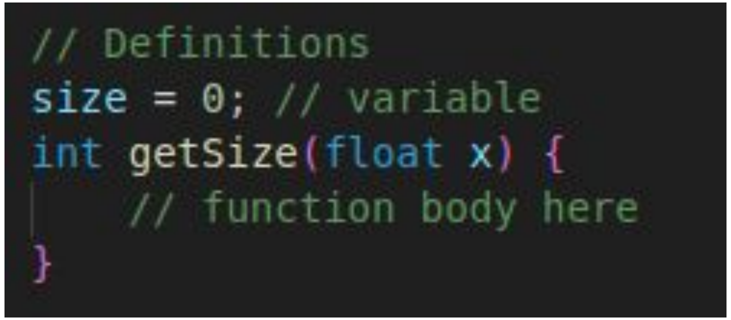


```
// Declarations
int size; // variable
int getSize(float x); // function
```

Definitions set objects in memory

- Variable definitions simply set the variable value
- Function definitions comprise the body of the function
- Note that function definitions double as declarations

Here I define the “size” variable to be 0. It must have been declared prior to this. I also add an empty function definition for “getSize”

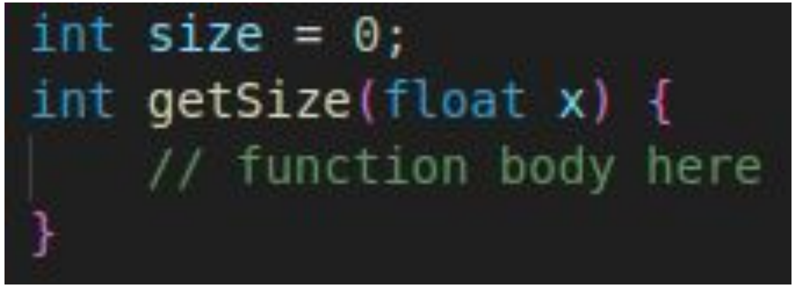


```
// Definitions
size = 0; // variable
int getSize(float x) {
    // function body here
}
```

Definitions can be made during declaration

- Convenient for variables
  - Initial value for a variable is easily read by other developers
- For functions, it's handy to keep them separate
  - We will see why in subsequent slides

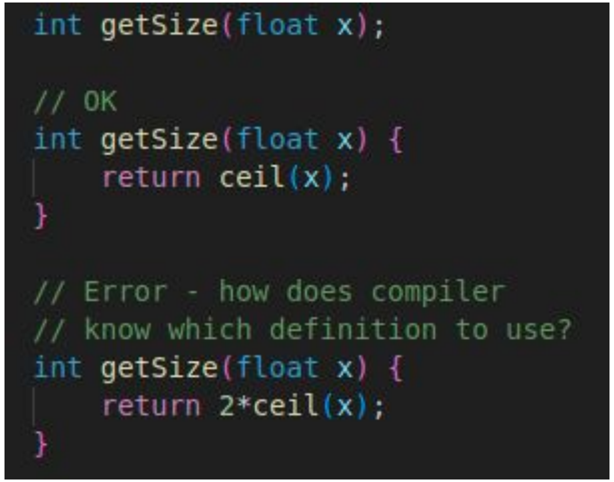
Initializing size variable at declaration and defining getSize function at declaration.



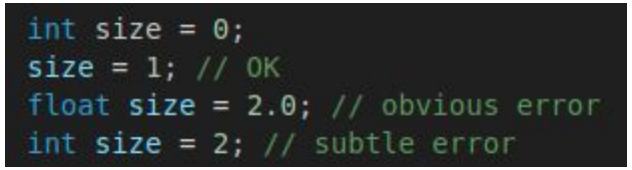
```
int size = 0;
int getSize(float x) {
    // function body here
}
```

Variables and functions differ on multiple declarations and definitions

- Functions can be re-declared, but not re-defined
  - Re-defining would cause function definition clashes
- Variables cannot be re-declared
  - Re-declaring could cause name clashes
- Other constructs (classes, structs) may behave differently, but generally follow variable rules
  - Consult a reference to verify



```
int getSize(float x);

// OK
int getSize(float x) {
    return ceil(x);
}

// Error - how does compiler
// know which definition to use?
int getSize(float x) {
    return 2*ceil(x);
}
```



```
int size = 0;
size = 1; // OK
float size = 2.0; // obvious error
int size = 2; // subtle error
```

Compilation is a three-step process

1. **Source code –** the file actually written by the user that contains function definitions – is *preprocessed* and converted into a **translation unit**
   a. Preprocessing involves reading through the file and running special commands called **macros**
   b. C++ macros generally start with #. A common example is #include, which copies file contents from another file into this file
2. Translation units are *compiled* into **objects**
   a. This converts the code into machine instructions
   b. The compiler may perform optimizations during this step
3. Objects are *linked* to form **libraries, executables,** or more **objects**
   a. Objects may call functions from other objects, so the linker tells the objects where the functions are in memory
   b. Executables can be directly run by the computer, whereas libraries are just disconnected collections of functions

Why is the C++ compilation process so complicated?

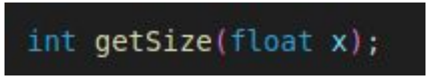
- If we were to put every function in one file, compiling that one file would take a very long time
  - Every time we touch that file, we'd have to compile the whole thing again
- The linker allows us to write code in one file that can be used by another file
- We can then touch code in one file and compile just that file to save time
- This also allows us to include external libraries, i.e. use functions written by other people, without having to compile it
- This can also be used to distribute libraries without divulging potentially confidential source code – we only need the library, not the source code, to link

Code is separated into headers and source files

- Compilers don't care about filenames, so naming conventions are project-specific
- Headers contain declarations, but no definitions
- Definitions are placed in source files
- To call a function defined in a different source file, include the associated header and call it normally
    - The header gives the declaration, and the linker will later hook up the function to the appropriate object file

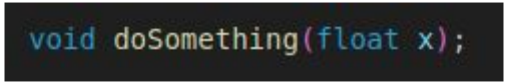
Example:

Library.hpp



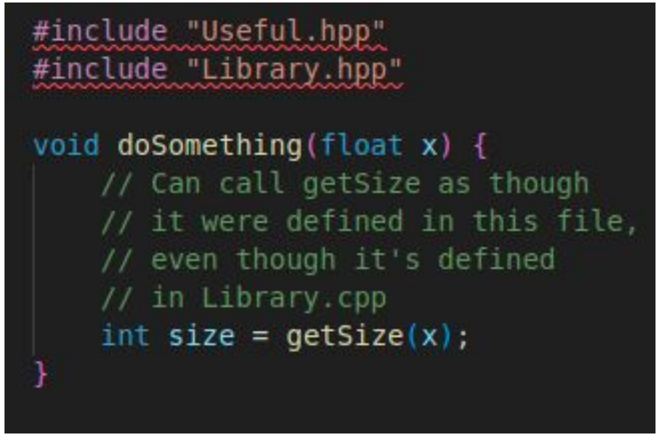
```cpp
int getSize(float x);
```

Useful.hpp



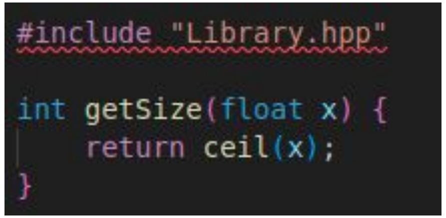
```cpp
void doSomething(float x);
```

Useful.cpp



```cpp
#include "Useful.hpp"
#include "Library.hpp"

void doSomething(float x) {
    // Can call getSize as though
    // it were defined in this file,
    // even though it's defined
    // in Library.cpp
    int size = getSize(x);
}
```

Library.cpp

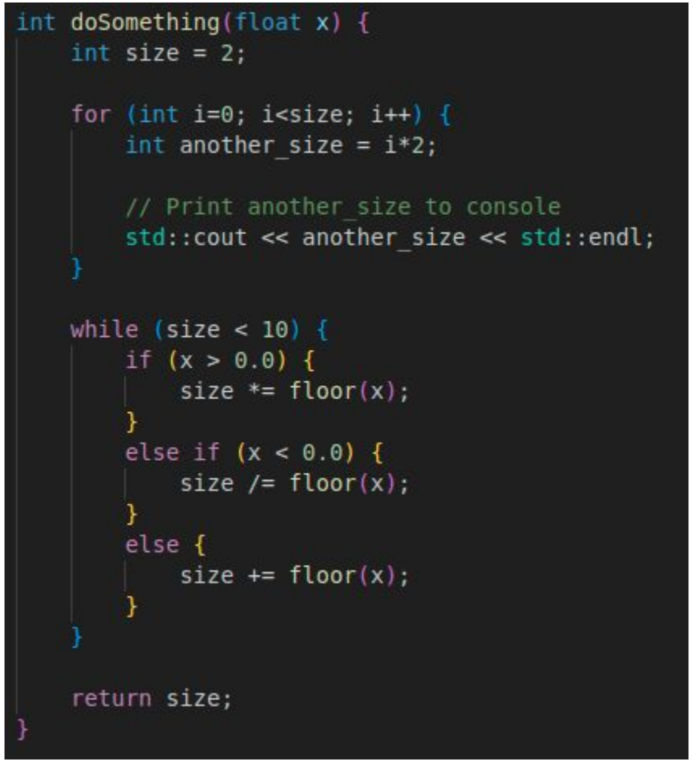


```cpp
#include "Library.hpp"

int getSize(float x) {
    return ceil(x);
}
```

C++ ignores most whitespace

Here I illustrate syntax for functions, for loops, while loops, if statements, and printing

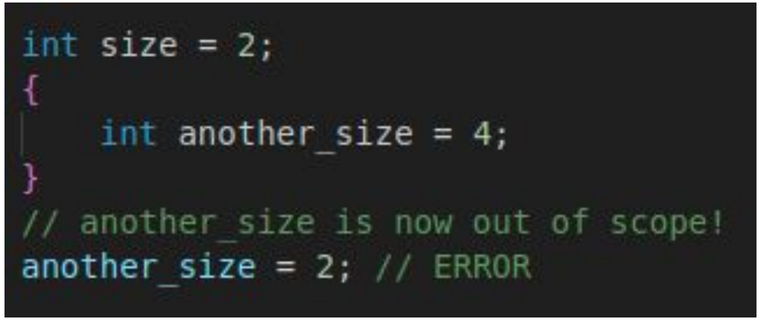
- …but whitespace is added for readability
- Expressions separated by semicolons
- Generally each expression is placed on its own line
- The conditions in while loops and if statements must be placed in parentheses, as shown
- For loops must provide three parts in parentheses
    - An initialization, where the looped variable is declared and defined
    - A condition for when to continue the loop
    - An update, which is performed every single time the block is executed
- Printing is most conveniently done via the std::cout stream



```cpp
int doSomething(float x) {
    int size = 2;

    for (int i=0; i<size; i++) {
        int another_size = i*2;

        // Print another_size to console
        std::cout << another_size << std::endl;
    }

    while (size < 10) {
        if (x > 0.0) {
            size *= floor(x);
        }
        else if (x < 0.0) {
            size /= floor(x);
        }
        else {
            size += floor(x);
        }
    }

    return size;
}
```

Curly braces typically denote the boundaries of scope

- Similar to other languages, variables are only defined in the scope in which they were declared
- Functions have a local scope – more on this next slide
- Curly braces also usually establish local scope in if statements and loops
- Variables that go out of scope are destroyed and deallocated

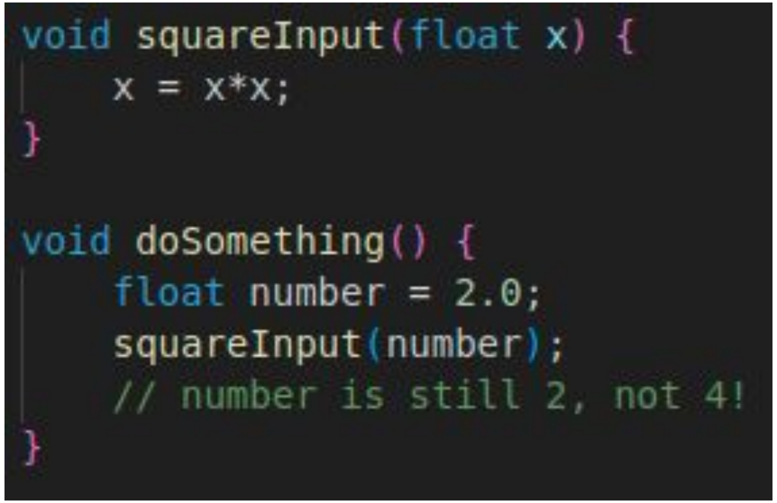


```
int size = 2;
{
    int another_size = 4;
}
// another_size is now out of scope!
another_size = 2; // ERROR
```

Functions typically copy input variables

- This means that changes made to input are not reflected later in the code
- In other words, input **plain old data (POD)** is not modified **in place**
- Does this mean we have to return everything we modify, even complex datatypes and structures?
    - Since C++ functions can only return one object, this can be difficult!

Example of maybe counterintuitive input copying

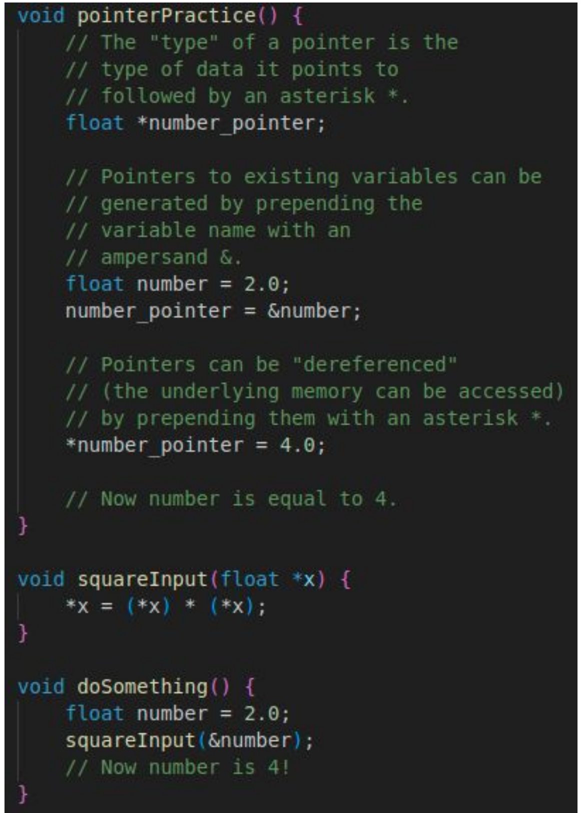


```
void squareInput(float x) {
    x = x*x;
}

void doSomething() {
    float number = 2.0;
    squareInput(number);
    // number is still 2, not 4!
}
```

Pointers allow us to modify memory in place

- Pointers are variables containing the address to chunk of memory
- These can be passed to functions
- The pointer itself is copied by the function, but the address it points to still points to the same thing
- Now we can modify variables in place!

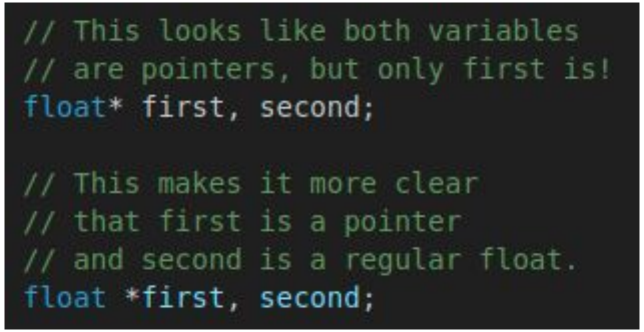
Pointer basics, plus the previous example redux



```
void pointerPractice() {
    // The "type" of a pointer is the
    // type of data it points to
    // followed by an asterisk *.
    float *number_pointer;

    // Pointers to existing variables can be
    // generated by prepending the
    // variable name with an
    // ampersand &.
    float number = 2.0;
    number_pointer = &number;

    // Pointers can be "dereferenced"
    // (the underlying memory can be accessed)
    // by prepending them with an asterisk *.
    *number_pointer = 4.0;

    // Now number is equal to 4.
}

void squareInput(float *x) {
    *x = (*x) * (*x);
}

void doSomething() {
    float number = 2.0;
    squareInput(&number);
    // Now number is 4!
}
```

Why put the asterisk next to the name instead of type?

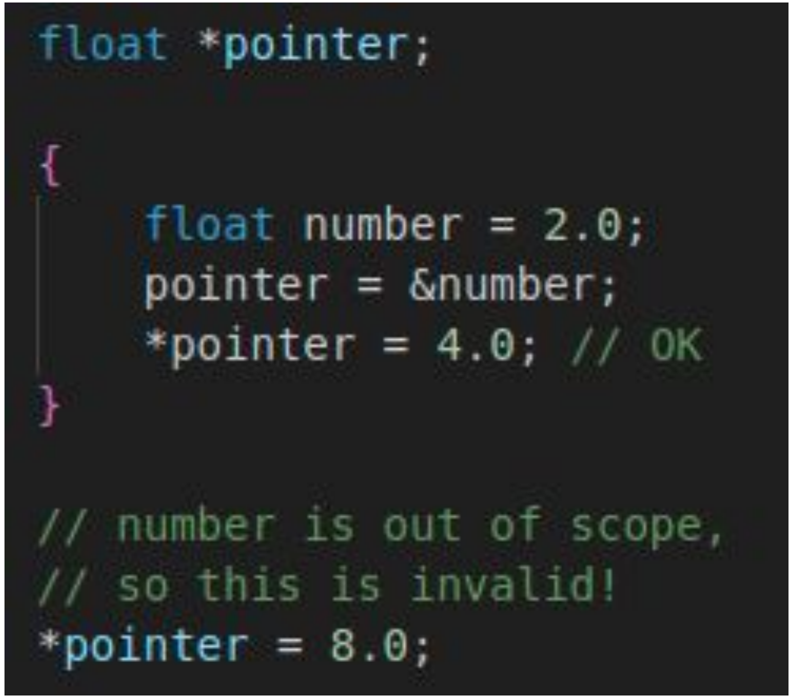
- We can declare multiple variables on one line
- Counterintuitively, asterisks only mark one variable as a pointer at a time
- C++ standard wants you not to think of first as a float*, but *first as a float (in the example to the right)
- Even though single line multiple declarations are legal, this example emphasizes that you should avoid them when not 100% clear!



```
// This looks like both variables
// are pointers, but only first is!
float* first, second;

// This makes it more clear
// that first is a pointer
// and second is a regular float.
float *first, second;
```

With great pointer comes great responsibility

- Pointers are a holdover from older languages (C and FORTRAN)
- Nothing stops us from trying to access invalid memory
- Pointer may point to nothing, or may point to memory that contains the wrong datatype
- Can be impossible or not performant to check on the fly, so lots of trust involved

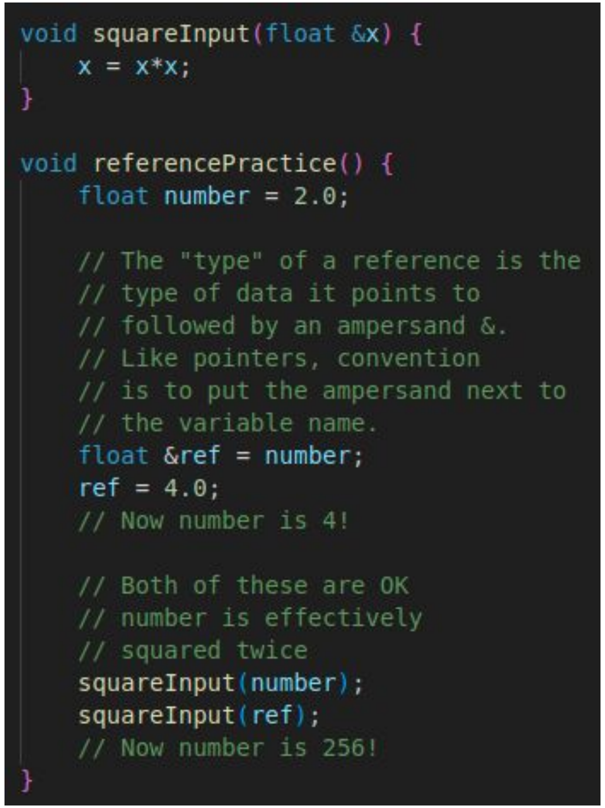


```
float *pointer;

{
    float number = 2.0;
    pointer = &number;
    *pointer = 4.0; // OK
}

// number is out of scope,
// so this is invalid!
*pointer = 8.0;
```

Enter references

Compare this to the pointer example!

- Kinda like pointers that act like regular variables
- Must be initialized with existing data, so they always point to valid data
- Can be passed to functions
- Much safer, but somewhat more restrictive
  - References cannot be reassigned, i.e. they always refer to the same variable
- Can't be used everywhere a pointer can, but should be preferred when possible

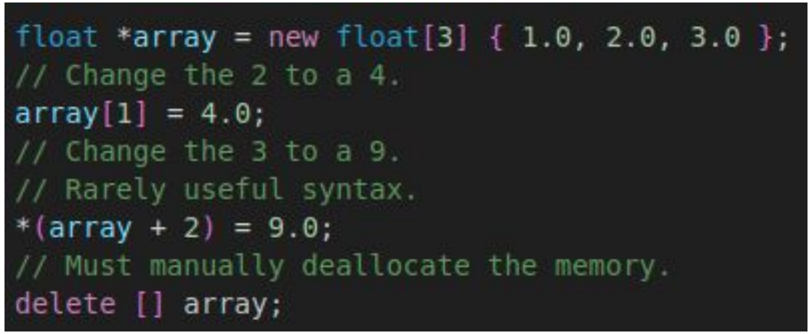


```cpp
void squareInput(float &x) {
    x = x*x;
}

void referencePractice() {
    float number = 2.0;

    // The "type" of a reference is the
    // type of data it points to
    // followed by an ampersand &.
    // Like pointers, convention
    // is to put the ampersand next to
    // the variable name.
    float &ref = number;
    ref = 4.0;
    // Now number is 4!

    // Both of these are OK
    // number is effectively
    // squared twice
    squareInput(number);
    squareInput(ref);
    // Now number is 256!
}
```

Arrays in C++ are just pointers

- Index from 0
- Points to the 0th element in the array
- Array creation is a little weird
  - We have to manually allocate the memory chunk
- Elements can be accessed using the familiar []
  - Or you can be fancy and dereference the pointer with an offset
- Since we manually allocated the memory, must remember to manually deallocate
  - Otherwise we have a leak!

Length 3 array creation, some access methods, and deallocation

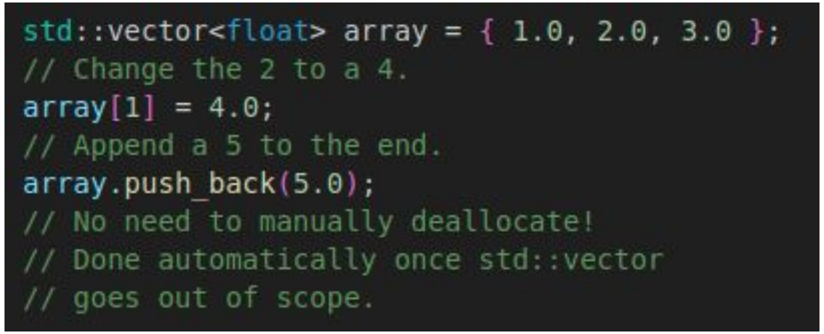


```
float *array = new float[3] { 1.0, 2.0, 3.0 };
// Change the 2 to a 4.
array[1] = 4.0;
// Change the 3 to a 9.
// Rarely useful syntax.
*(array + 2) = 9.0;
// Must manually deallocate the memory.
delete [] array;
```

std::vectors handle the allocation and deallocation automatically

- Must #include <vector> at the top of your file
- Under the hood, still just a chunk of memory with a pointer to the 0th element
- The datatype in <> is the type contained by the std::vector
  - We'll see next session how this works
  - (spoiler: it's a template)
- Unlike manual arrays, it's easy to add or remove data and change array size
- In general, we want to avoid using objects that require manual deallocation
  - That would be very un-C++-like!
  - **Resource acquisition is initialization (RAII)**

Compare to previous example! Much easier to work with

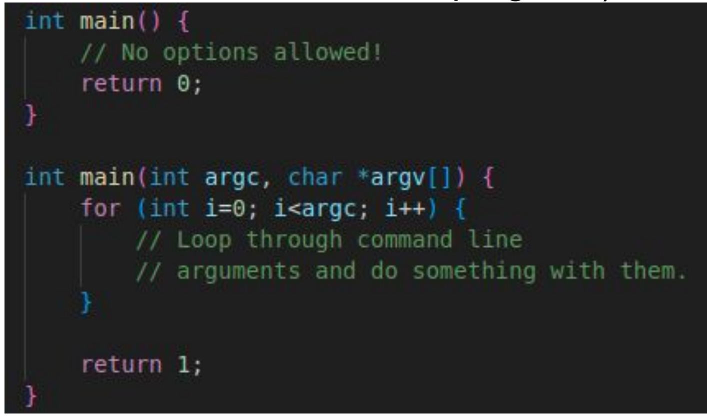


```
std::vector<float> array = { 1.0, 2.0, 3.0 };
// Change the 2 to a 4.
array[1] = 4.0;
// Append a 5 to the end.
array.push_back(5.0);
// No need to manually deallocate!
// Done automatically once std::vector
// goes out of scope.
```

The “main” function is special

- Any source file containing a function called main can be compiled into an executable
- The main function must return an integer
  - This integer is the **error code**
  - Generally, 0 means successful termination, whereas anything else means an error occurred during execution
- The inputs must either be nothing, or an integer and a 2D array of characters
  - The integer is the number of arguments in the command line
  - Always at least 1! The program name itself is the 0th argument
  - An array of characters forms a string, so the 2D array is the list of arguments as strings
- Old-fashioned holdover from C

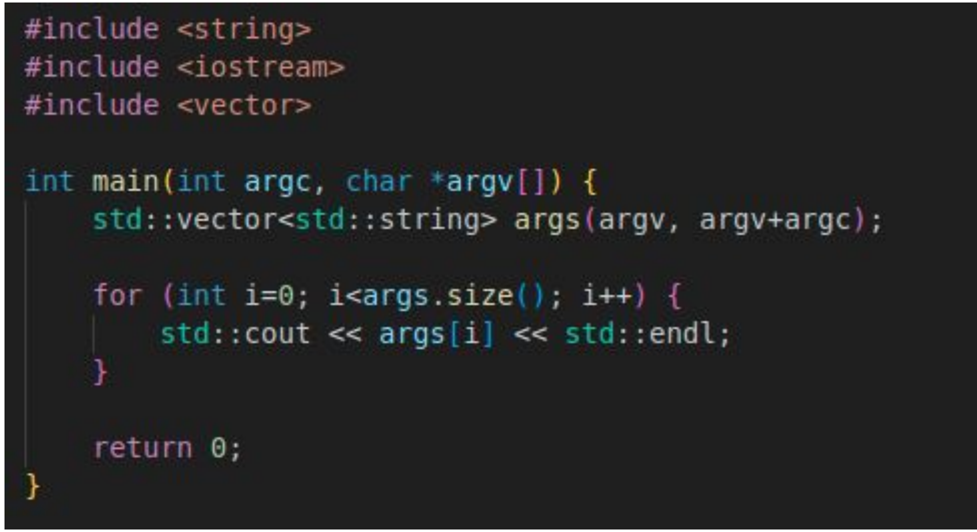
Two valid main functions (don’t actually write more than one in a program!)



```
int main() {
    // No options allowed!
    return 0;
}

int main(int argc, char *argv[]) {
    for (int i=0; i<argc; i++) {
        // Loop through command line
        // arguments and do something with them.
    }

    return 1;
}
```

Char arrays are tough to work with, so turn them into a std::vector!

- This is more convenient and easier to read
- Note the #include's; these declare std::vector, std::string, std::cout, and std::endl
- This example just loops through and prints each argument on its own line



```
#include <string>
#include <iostream>
#include <vector>

int main(int argc, char *argv[]) {
    std::vector<std::string> args(argv, argv+argc);

    for (int i=0; i<args.size(); i++) {
        std::cout << args[i] << std::endl;
    }

    return 0;
}
```

Miscellaneous takeaways

- Ignore most of the blue keywords in hPIC2 (const, constexpr, volatile, override, final, etc.). Most of these do not affect how the program behaves, and are just compile-time checks that the developer didn’t make a mistake
- Don’t reinvent the wheel! Check cppreference to see if the STL has an algorithm or container you want to use before writing it. It’ll be faster than whatever you come up with

Exercise

1. Write a function (fibonacciVector) with one int input that returns a std::vector<int> containing the Fibonacci sequence with the input number of elements. Declare this function in Fibonacci.hpp and define it in Fibonacci.cpp
    a. Make sure you #include <vector> and #include “Fibonacci.hpp” in your source file!
2. Run `g++ -c Fibonacci.cpp` to compile this function into an object file
3. Write a main function that reads the command-line arguments, calls fibonacciVector to construct the Fibonacci sequence of length specified on the command line, and prints each element out on its own line. Define this function in main.cpp (no header required)
    a. Print an error and return a nonzero code if too many, not enough, or invalid (i.e., non-integer) arguments are provided by the user
    b. Use std::stoi to convert a string to an integer
    c. Make sure you #include the appropriate headers in your source file (probably <string>, <iostream>)
4. Run `g++ main.cpp -o main Fibonacci.o` to compile and link your executable
5. Test it by running `./main 4` and play around with the numbers
    a. At what numbers does it break? Why?