1 The Build phase

1.1 Declaration vs Definition

It is often the case when writing .cpp code that we wish to say a symbol is valid without actually giving the symbol a definition. Consider the following, mostly nonsensical, code:

```cpp
#include <iostream>
#include <cmath>

double theOtherFunction(double x);
double theFunction(double x) {
    std::cout << x << "\n";
    if (abs(x) <= 1) {
        return 1;
    }
    return theOtherFunction(1/sin(x));
}
double theOtherFunction(double x) {
    std::cout << x << "\n";
    if (abs(x) >= 10) {
        return 10;
    }
    return theFunction(1/cos(x));
}

int main() {
    theFunction(2.5);
}
```

Without the declaration on line 4, we would get a compiler error because theOtherFunction would be called on line 11 without having been defined (and the compiler moves from the top of the document to the bottom). Declarations tell the compiler that a symbol, in this case “theOtherFunction,” is syntactically valid, and also tell the compiler how the symbol should be used, by passing in a double as a parameter and obtaining a double as a return value. In this kind of situation, where two functions call one another, we are forced to use declarations: any definition order without declarations will result in an undefined symbol.

In addition to functions, one can also declare classes. Templated classes cannot be declared however, as will be discussed later in the course.

1.2 Compilation vs linking

We saw in subsection 1.1 that the compiler does not require definitions; just declarations. Indeed, just by telling the compiler that theOtherFunction is a valid symbol and how it is used, the compiler
was able to make sense of the call to theOtherFunction in theFunction. This ability to separate declarations from definitions is what allows the “build phase” to be separated into a “compilation phase” and a “linking phase.” The build phase here refers to the general procedure of generating an executable from some resources (really, we should be careful using the term “compilation” as a general purpose term for creating an executable; it has a more precise meaning).

During the compilation phase, C++ code is converted into machine code (or more precisely, object code). C++ code, although it may not always seem so, is meant to be human-readable. As a result, the computer itself, which is a collection of wires and silicon chips, cannot make sense of it. Machine code is machine-readable code that can be understood by the actual chips on the computer. It consists of binary instructions that are translated into physical electrical impulses in the circuits within the computer. Thus, one can think of the compilation-phase as a translation phase from human- to machine-readable code.

The machine code output of the compilation phase is contained within files called object files (.obj or .o files). However, we cannot simply use these object files because some of the symbols in the object files may have just been declared but not defined. During the linking phase, declared symbols in various object files are defined by finding the correct definitions in other object files. The result of this is a set of fully defined object files that are tied up into an executable files (.exe file). Thus, the linker takes a set of object files as input, resolves definition dependencies between the object files, and produces an executable.

When building your own code, you may see an error like “unresolved symbol ...”. This occurs when the linker searches for the definition corresponding to a provided declaration and is unable to find anything. Often this happens when one forgets to define a function declared within a class interface, or provides a different function signature in a definition than was provided in the declaration.

1.3 Header files vs cpp files

By separating our code into .cpp files and .h (or, if you like, .hpp) files, we are saying we want the linker to do some heavy lifting during the build phase rather than having the compiler do all the work. Generally, we place declarations in .h files and definitions in .cpp files. When we want to use a particular function or class, we #include the .h file (not the .cpp file!) containing the declaration of the function or class. As a result, any files in which the .h file is included will be converted to object files during the compilation phase, and these object files will not contain the definition of the function or class we want. Therefore, we depend on the linker to resolve these dependencies.

The main purpose of all of this is to reduce compilation times and improve organization. The latter is self-explanatory so let’s focus on the former. Say we have a large collection of functions and classes defined in .cpp files we want to use. To use these functions and classes, we need to use the compiler to convert the C++ code into .obj files. By separating this collection of code into .cpp and .h files, we only have to compile the large collection once (into an archive with extension .a)—not every time we want to use it. Indeed, to use this collection in a file main.cpp, we only need to write #include “theCollection.h”, which will place many function and class declarations within our code. The linker will then provide these declarations with the correct definitions, which are contained in .obj files that we already compiled. The result is that the compilation phase does not touch the files containing the definitions; the linking phase simply finds the correct definitions
in the .obj files.

By avoiding having to compile code containing definitions, we can save a huge amount of time during the build phase.

2 Constructor Initializer Lists

Important points to remember about constructor initializer lists:

1. It is common to have an object as member data of a class in C++. When this is the case, initializer lists allow one to directly call the desired constructor for class member data, rather than calling the default constructor and then modifying the value. For example, the following code will not compile because the struct Within has no default constructor.

```cpp
struct Within {
    Within(int x) : x(x) {};
    int x;
};

class Contains {
    Contains(int x) {
        within.x = x;
    }
private:
    Within within;
};
```

By replacing the constructor Contains::Contains(int x) with

```cpp
Contains::Contains(int x) : within(x) {};
```

the code will work appropriately. Additionally, because initializer lists can directly call the desired constructor for a class, it is more efficient to use a constructor initializer list then to initialize members within the body of a constructor.

2. The order of initialization depends on the order in which class members are declared within the class. Consider the following example,

```cpp
class orderMatters {
public:
    orderMatters(int y, int x) : y(x), x(1) {};
private:
    int x, y;
};
```

The variable x is declared first within the class, so the compiler sets x to 1 before setting y to x. Therefore, by the end of construction, x and y are both equal to 1.

3. const member data and reference member data must be initialized in an initializer list. Why this is so is similar to the reason we must initialize class members without default constructors: const member data cannot be changed after initialization and references cannot be initialized without providing a member location to bind to. Indeed, the following code fails,
3 Iterators

To access the elements of an array, we just need to know the index of the element we want to access. This is because arrays are consecutive blocks of memory, where the size of each block is the size of the object that the array stores. Thus, if we know the array starts at memory address 1, and we know that each block of memory is 8 bytes, say, then we know the address of element \( n \) is \( 1 + n \times 8 \). This property is called random access: one can immediately access any element of an array. In C++, what this looks like is we have a pointer `ptr` to the first element of the data structure, and incrementing it (`++ptr`) \( n \) times gives us the \( n \)th element.

However, many containers do not have this property. We can imagine a data structure where each element is placed non-consecutively in memory, and the location of the next element is kept at the location of the previous element. Thus, in order to access the \( n \)th element, we must traverse the \( n - 1 \) prior elements of the data structure, following directions from one element to the next. To enable this kind of traversal behavior, we use something called an iterator, which is a generalization of a pointer. Incrementing a pointer moves the pointer to the next block of memory, which for an array is the next element. Incrementing a iterator moves the iterator to the next element of the data structure, which generally is not the next block of memory. (The data structure described above is called a singly-linked list.)

For example, consider two data structures—an array and the STL container std::set—and we wish to iterate through the two containers. Our code would look as follows,

```c++
int main() {
    int x; // cannot declare a reference without initialization
    const int y;
    y = 5; // cannot modify the value of a const variable
}
```

```c++
3 Iterators

To access the elements of an array, we just need to know the index of the element we want to access. This is because arrays are consecutive blocks of memory, where the size of each block is the size of the object that the array stores. Thus, if we know the array starts at memory address 1, and we know that each block of memory is 8 bytes, say, then we know the address of element \( n \) is \( 1 + n \times 8 \). This property is called random access: one can immediately access any element of an array. In C++, what this looks like is we have a pointer `ptr` to the first element of the data structure, and incrementing it (`++ptr`) \( n \) times gives us the \( n \)th element.

However, many containers do not have this property. We can imagine a data structure where each element is placed non-consecutively in memory, and the location of the next element is kept at the location of the previous element. Thus, in order to access the \( n \)th element, we must traverse the \( n - 1 \) prior elements of the data structure, following directions from one element to the next. To enable this kind of traversal behavior, we use something called an iterator, which is a generalization of a pointer. Incrementing a pointer moves the pointer to the next block of memory, which for an array is the next element. Incrementing a iterator moves the iterator to the next element of the data structure, which generally is not the next block of memory. (The data structure described above is called a singly-linked list.)

For example, consider two data structures—an array and the STL container std::set—and we wish to iterate through the two containers. Our code would look as follows,

```c++
int main() {
    int arr[5]{1,2,3,4,5}; // create an array of length 5 with the numbers 1 through 5
    std::set<int> set{1,2,3,4,5}; // create a set of size 5 with the numbers 1 through 5
    // iterate through the array, squaring each number
    // the variable arr casts to a pointer to the first element of the array
    // (create a pointer to the beginning ; check if pointer is at the end ; increment pointer)
    for (int* ptr = arr; ptr != arr+5; ++ptr) {
        (*ptr)*=2;
    }

    // iterate through the set, squaring each number
    // (create an iterator to the beginning ; check if iterator is at the end ; increment iterator)
    for (std::set<int>::iterator it = set.begin(); it != set.end(); ++it) {
        (*it)*=2;
    }
}
```

The for loops are syntactically identical, but to loop through an array we use a pointer, and to loop through the set we use an iterator. The iterator hides the details and allows us to pretend that the iterator is a simple pointer (although perhaps with limitations, such as only being able to move in
one direction). Although it is not always possible with iterators, we can move backwards through
the std::set using std::set<int>::rbegin() and std::set<int>::rend() with the same syntax as before:

```cpp
int main() {
    int arr[5]={1,2,3,4,5}; // create an array of length 5 with the numbers 1 through 5
    std::set<int> set={1,2,3,4,5}; // create a set of size 5 with the numbers 1 through 5
    // iterate backwards through the array, squaring each number
    // the variable arr casts to a pointer to the first element of the array
    // (create a pointer to the beginning ; check if pointer is at the end ; increment pointer)
    for (int* ptr = arr + 4; ptr != arr - 1; --ptr) {
        (*ptr)*=2;
    }
}
```

C++ also supplies some “syntactic sugar” in the form of range-based for loops as follows:

```cpp
int main() {
    std::set<int> set={1,2,3,4,5}; // create a set of size 5 with the numbers 1 through 5
    for (std::set<int>::iterator it = set.rbegin(); it != set.rend(); ++it) {
        (*it)*=2;
    }
}
```

Note we use a reference here because we want to modify the elements of set; without the reference
we would be creating copies of each element.