

Introduction to Programming

PIC 10A, UCLA

©Michael Lindstrom, 2015-2019

This content is protected and may not be shared, uploaded, or distributed.

The author does not grant permission for these notes to be posted anywhere without prior consent.

Introduction to Programming

Computing is almost magical: within even the smallest computers these days, there is an intricate structure of circuitry allowing for the movement and storage of data and a means for us to view and interact with that data. The means by which we interact with the data is primarily done through programs that can translate human thought and intent into instructions simple enough to be carried out in rapid succession by the tiny elements making up a computer.

Computer Anatomy and Terminology

- ▶ **Computer:** an electronic device that can execute simple operations in rapid succession. It moves or places patterns of bits (0's and 1's) in memory.
- ▶ **Program:** an interface (means for a user to give/receive input) that achieves a given result. There are many varieties: word processors, Facebook, video games, calculators, etc.

A program will often entail a series of procedures and algorithms to achieve a given functionality: verifying a username and password, checking for messages, doing a spell check, etc.

Computer Anatomy and Terminology

- ▶ **Algorithm:** an orderly sequence of steps to achieve a given result.

Ex:

-to calculate the average number from a list given, add up all the numbers in the list and divide by the total number of items in that list.

-to put words in alphabetical order: go through the entire list and find the first word and swap it with what is at the beginning; then go through all words except for that first word, finding the first word of the remaining list, and swap that word with the second position; repeat until all the words are in place:

Start with: “cat”, “dog”, “apple”

Identify first word (“apple”) and swap with first position (“cat”):

“apple”, “dog”, “cat”

Identify next word (“cat”) and swap with second position (“dog”):

“apple”, “cat”, “dog”

Done!

Computer Anatomy and Terminology

- ▶ **Variable:** something storing/representing data, such as a user's username, or a value representing the sum of a list of numbers
- ▶ **CPU:** the Central Processing Unit - essentially the brain of the computer. Data moves through the CPU, which can direct computer processes and perform arithmetic operations.

It is a complex combination of wires and transistors, mostly made from silicone.

- ▶ **Memory:** stores data (such as phone numbers, the words of an essay, etc.) and programs (such as Paint or Visual Studio).

Memory is divided into various parts: primary, secondary, and tertiary storage.

- ▶ **Primary storage:** split into Random Access Memory (RAM) and Read Only Memory (ROM).

Computer Anatomy and Terminology

- ▶ **Random Access Memory:** the CPU can access RAM more easily than secondary storage and thus data transfer between the CPU and RAM is faster.

This memory is volatile: the data here are lost when power is lost.

Programs primarily run in the RAM; data can be read from and written to the RAM during the execution of a program.

It is quite expensive.

- ▶ **Read Only Memory:** this memory stores essential programs, such as those that must be executed when the computer is turned on.

It is virtually impossible to alter (write).

Computer Anatomy and Terminology

- ▶ **Secondary Storage:** also called the **hard disk**, it allows for long term data storage. Data are stored magnetically on metal disks.

Programs are stored in the secondary storage and then loaded into the RAM to run.

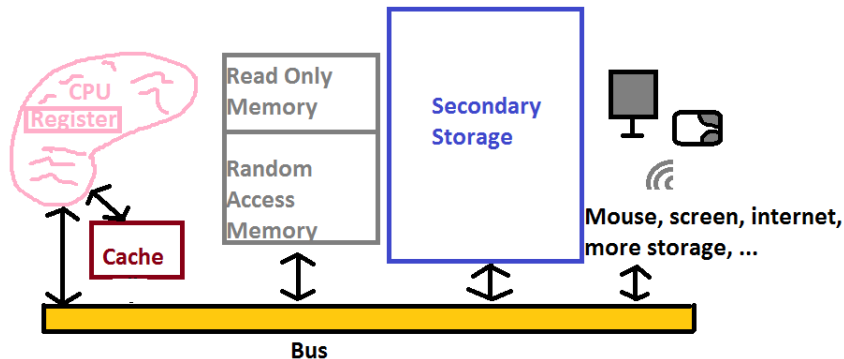
- ▶ **Tertiary Storage:** storage external to the computer like USBs, etc.
- ▶ **Register:** a storage location in the CPU for very fast processes such as simple arithmetic, storing the location in memory of a value, what point a program is at in its execution, etc.

Memory here is very limited, but it is very fast because it is located within the CPU.

Computer Anatomy and Terminology

- ▶ **Cache:** a segment of memory external to the CPU but closer than the RAM. The memory is quite limited, but its use can speed up programs.
- ▶ **Bus:** a bridge between the CPU, RAM, and external component of a computer (monitor, mouse, internet, etc.).

Computer Anatomy and Terminology



From Source Code to a Running Program

- ▶ **Machine code:** ultimately a computer can only “understand” if something is off or on - 0’s and 1’s.

For example, telling the computer to place the value 1 at a specific register location, in a given computer’s machine code (“native language”) could look something like:


| | |
|-------------------|-----------------|
| register location | value of 1 |
| <u>10110000</u> | <u>00000001</u> |

but no human wants to read this! Nonetheless, at the lowest level, this is what a program looks like to a computer.

The machine code varies from computer architecture to computer architecture.

From Source Code to a Running Program

- ▶ **Assembly language:** this is a higher level language than machine code: it looks a little more like human language, but depends on a given computer's architecture. That previous statement could perhaps be written as:

placement command location value 1

MOV AL, 1h

- ▶ **Programming Language:** a means of communicating instructions to a machine with a set of commands. The more like English the code is and the more built-in functionalities the language has, the higher the language is.

Ex: C, C++, Java, Fortran, Basic, Python, ... are all higher level languages than assembly

- ▶ **C++:** a language initially developed by Bjarne Stroustrup in 1979 to combine object oriented programming with the speed of the C language. This is the one we'll be studying in PIC 10A. It has continued to evolve over time.

From Source Code to a Running Program

- ▶ **Editor:** the environment in which a program is written.
- ▶ **IDE (Integrated Development Environment):** software that combines not only the editor environment, but also the means to create a fully functional program. *Ex:* Visual Studio, Xcode, etc.
- ▶ **Source Files:** files that contain the raw text written by a programmer that are used to generate a functional, running program.
Ex: in C++, usually these files have extension **.cpp** or **.h**.

From Source Code to a Running Program

An example of a **.cpp** file is below:

HelloWorld.cpp

```
#include<iostream>
```

```
using namespace std; // specify the standard namespace
```

```
int main() {
```

```
    cout << "Hello, world!\n"; // display greeting and append new line
```

```
    cout << "How are you?"; // display question
```

```
    cin.get(); // line added so the program pauses until the user hits ENTER
```

```
    return 0; // terminate program
```

```
}
```

From Source Code to a Running Program

The source code of **HelloWorld.cpp**, when compiled and turned into a program, would generate the following output:

```
Hello, world!  
How are you?
```

Observe how much more like English that code is than the assembly language example. In many programming languages, it is also possible to provide **comments** (prefixed the `//`) that are not visible to the computer, but can be read by the programmer.

Code should be easy to read and follow. Even someone who does not know a given programming language should understand the logic involved as a program executes.

From Source Code to a Running Program

Consider the code snippets:

First Code Snippet (bad code):

```
double x = 0;  
size_t y = L.size();  
for (int p: L )  
    x += p;  
double z = x/y;
```

From Source Code to a Running Program

Second Code Snippet (not great code):

```
double x = 0; // the cumulative total: start at 0
size_t y = L.size(); // number of elements to average

for (int p : L ) // for each number of the list
    x += p; // add it to the cumulative total

// compute the average by dividing total by number of elements
double z = x/y;
```


From Source Code to a Running Program

Third Code Snippet (good code):

```
double total = 0; // the cumulative total: start at 0
size_t numberOfElements = list.size(); // number of elements to average

for ( int value : list ) { // for each number of the list
    total += value; // add it to the cumulative total
}

// compute the average by dividing total by number of elements
double average = total / numberOfElements;
```

Remark on terminology: **total**, **numberOfElements**, **value**, **list**, and **average** are all “variables” in that they store information.

From Source Code to a Running Program

When C++ source code is converted into a functional executable (program), a series of steps take place:

- ▶ **Preprocessor:** the preprocessor follows instructions to **#include** various files and generates (larger) source files that the compiler can parse; it may define various symbols or protect against invalid definitions, etc.

Its instructions are prefixed by **#**.

It basically generates large source files with all the information the compiler (next phase) needs to do its job.

From Source Code to a Running Program

- ▶ **Compiler:** the compiler converts source code into assembly language code and along the way, it performs many steps.
 - ▶ Syntax: checks that the code obeys proper “grammar” (statements ending with semi-colons `;`); that the inputs/outputs are compatible (a number should not be used in lieu of a block of text, etc.); etc.
 - ▶ Optimization: can make code written by a programmer more efficient by simplifying or combining different commands.
 - ▶ Register allocation: determines which registers should be used for different processes.
 - ▶ Inserts “placeholders”: it may know that a variable **x** is an integer of type **int**, but it may not know its precise definition (where it lives in memory) and it leaves space for this definition to be given elsewhere.

For example: think of the compiler as only needing to know what a word is. It might not know the meaning of “cat”, but given that it knows “cat” is a noun, it can determine

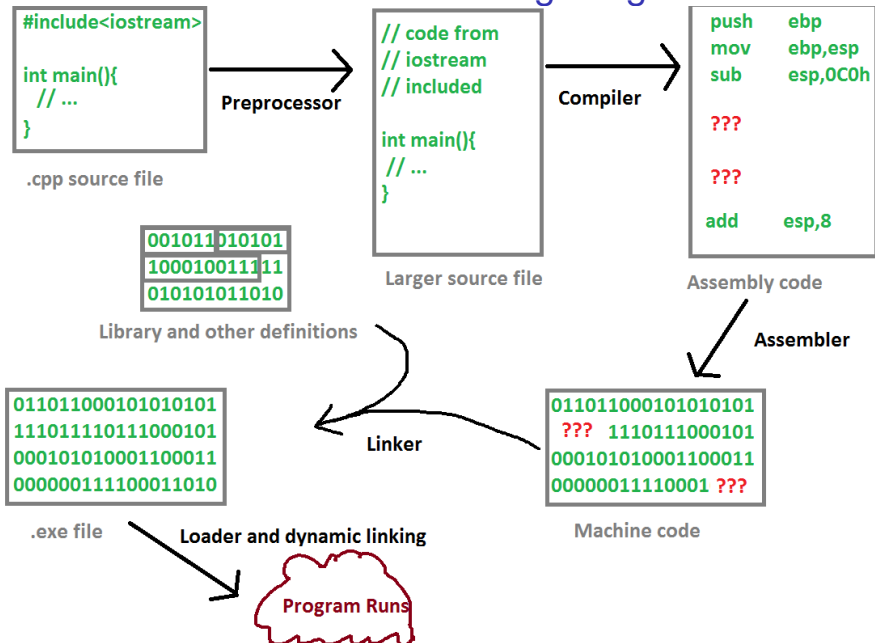
The cat is cute. is grammatically correct and

It cat is grammatically incorrect (cat can't be used like that and the sentence has no punctuation).

From Source Code to a Running Program

- ▶ **Assembler:** the assembler takes assembly language code and converts it into machine code, keeping the placeholders.
- ▶ **Library:** a collection of **object files** (files in machine code) that define various methods, functions, or symbols. The compiler may not have access to these definitions, but they are stored somewhere.
- ▶ **Linker:** the linker takes the assembler output and searches for the missing definitions, in the library and elsewhere, that the compiler did not specify. It fills in the different placeholders with the real definitions and forms a “complete” program.
- ▶ **Executable:** the output from the linker when the program is complete - a file with **.exe** extension.
- ▶ **Loader and dynamic linker:** when a program is run, the **loader** allocates memory in RAM for the program. Further linking may actually take place as the program runs - this is **dynamic linking**.

From Source Code to a Running Program



Types of Errors

When we write programs, there are various types of errors that we encounter in getting a program to work as intended.

- ▶ **Compiler errors:** these are errors that the compiler recognizes and we have to fix the code syntax/structure before the program can even be run.
- ▶ **Linker errors:** these are errors that make it past the compiler that arise when the linker cannot find a working definition for something or is perhaps given conflicting definitions, etc. These also must be fixed before the program can be run.
- ▶ **Logical (run-time) errors:** these are errors in logic whereby the code is valid and can run, but it does not do what we intend it to do.

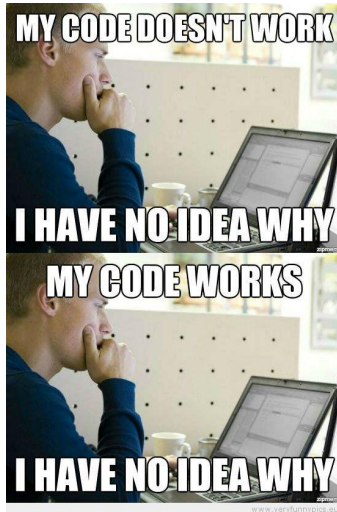
In a sense, both compiler and linker errors are compile-time errors in that they come up before a program can be built.

Types of Errors

Programming is a cycle: after planning out the code, write it, and compile it. If it does not compile, fix it. Once it compiles, check that it behaves as intended. If it does not, fix the code and recompile, ...

Types of Errors

Programming is a cycle: after planning out the code, write it, and compile it. If it does not compile, fix it. Once it compiles, check that it behaves as intended. If it does not, fix the code and recompile, ...



A Simple C++ Program

Let's go back and analyze the **HelloWorld.cpp** program:

```
#include<iostream>
```

```
using namespace std; // specify the standard namespace
```

```
int main() {
```

```
    cout << "Hello, world!\n"; // display greeting and append new line
```

```
    cout << "How are you?"; // display question
```

```
    cin.get(); // line added so the program pauses until the user hits ENTER
```

```
    return 0; // terminate program
```

```
}
```

We will analyze the code line-by-line.

A Simple C++ Program

```
#include<iostream>
```

This is an instruction to the **preprocessor** to find a file called **iostream** and to dump its contents into the current file.

iostream is a **header file** that declares important functions for displaying stuff to the console window and for accepting user input. Without **iostream**, we cannot display a message as we do in this program.

iostream: think input-output **streams**.

header files in general declare the existence of different functions, etc.: they are like a catalog that advertises what should be around to the compiler.

Back to the language analogy: a header file might classify all the words of a dictionary, but it is not required to define them. The classification is all the compiler needs.

A Simple C++ Program

```
using namespace std;
```

This tells the compiler that we are working in the **standard namespace**, abbreviated by **std**.

A **namespace** is kind of like a regional dialect. The symbols like **cout** that we use in the program have a special meaning when we're in the **std** namespace.

A Simple C++ Program

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

(we have ignored what is inside the { } for now).

All C++ programs have a **main routine**. In order for a program to run, the compiler needs to find this routine. The sequence of instructions within the main routine are executed in sequence and this is how the program runs.

The main routine is a **function**.

A Simple C++ Program

```
//
```

Commenting is essential to writing good, readable code. Comments are invisible to the compiler. The logic and intent of each section of code should be clear from the comments.

A **single-line** comment is prefixed by //

It is also possible to comment for more than a line using /* **and** */:

```
/* The line of code below will display a greeting to the user of the program  
and also append a new line */
```

```
cout << "Hello, world!\n";
```

A **multi-line comment** begins with /* and ends with the next */

A Simple C++ Program

```
cout << "Hello, world!\n";
```

cout: this is a special variable that can print data to the screen, short for console **output**.

"stuff": characters appearing between a pair of quotation marks are called **string literals**.

A **character** is essentially a single unit of text, such as a single letter from A-Z or a-z, or a single digit from 0-9, or a punctuation mark like . or ?, or even a space character.

String literals are surrounded by ": one to “open” and the other to “close”.

The string literal "" would be empty because there is nothing stored between the opening and closing quotation marks.

A Simple C++ Program

```
cout << "Hello, world!\n";
```

When **cout** combines on the left of << with some other piece of data that can be displayed on the right (such as a number or string literal), the data is displayed in the console window.

All C++ statements terminate with a semi-colon ;

The backslash (\) is an **escape character**: it is not printed, but it has an effect. It can be used to define various white space characters.

When the compiler sees \n, it inserts a new line character, allowing the next message **"How are you?"** to appear on the next line.

A Simple C++ Program

Some important characters:

\n: new line

\t: tab (roughly 4 spaces wide)

\": " quotation marks that will not be confused with closing quotes

\': ' a single quote

****: \ backslash character not to be confused with an escape sequence

\b: (conditionally supported by some compilers) moves the cursor one space to the left, provided we stay on the same line

\a: (conditionally supported by some compilers) makes an audible beep

```
cout << "ABC"; // print ABC
```

```
cout << "\b"; // go left to overwrite the C
```

```
cout << "\\"; // overwrite the C with \
```

AB\

A Simple C++ Program

```
cin.get();
```

This statement can be thought of as having two parts.

cin is a special variable that deals with user input from the console.
Think: **con**sole **in**put.

get() is a function belonging to **cin** that tries to collect a single character of user input.

cin.get() tells the compiler that we wish to obtain an input from the user. Until the user gives an input, such as hitting [ENTER] on the keyboard, the statement cannot execute. The effect of this is that the program “waits” until the user has entered something.

A Simple C++ Program

```
return 0;
```

Recall that mathematically, a “function” has an input and an output.

The **signature** of the main routine is **int main()**. This means that after it has finished running, it should give an output of type **int**, representing an integer type.

The main routine doesn't accept any inputs, and this is why there is nothing inside the parentheses **()**.

A Simple C++ Program

```
return 0;
```

In our program, we terminate the main routine (and the entire program execution) by giving an output for the main routine. We do this with the **return 0;** statement: in doing so, **main** returns a value of **0**.

Usually a value of **0** indicates a successful run of the program; other values, such as negative values, could be returned to indicate failure.

A Simple C++ Program

Some overall remarks and further details:

- ▶ Writing **using namespace std;** is not required and later we'll see times when it should be avoided. If we do not write that line, then we need to explicitly specify what namespace our variables live in and we would replace **cout** with **std::cout** and **cin** with **std::cin**.
- ▶ **cout <<** can display things other than string literals. Given a number (or something that evaluates to a number), it can print that, too:

```
cout <<15; // will print 15  
cout << 0 - (3.14 + 4 - 8.4/2); // will print -3.04
```

It treats + as addition, - as subtraction, * as multiplication, / as division, and (and) as parentheses. Note how the order of operations are preserved!

A Simple C++ Program

- ▶ **cout** << can display a single character, too:

```
cout << '&'; // will print & to screen.
```

Note that a single character appears in single quotes, whereas a string literal appears in quotation marks. To denote the character '\', we must write '\\'

- ▶ **cout** << can be “chained together”, with each part of the output separated by <<:

```
cout << "Hello, world!\n" << "How are you?";
```

The line of code above does the same as the lines of code below:

```
cout << "Hello, world!\n";  
cout << "How are you?";
```

A Simple C++ Program

- ▶ Juxtaposing two items without << between them is generally nonsense, leading to compiler errors:

```
/* juxtaposing string literal and number and character cannot be  
parsed */  
cout << "Eight minus one is " 7 '!';
```

Should be:

```
cout << "Eight minus one is "<< 7 << '!';
```

- ▶ Another way to achieve a new line, that has a similar effect to `\n`, is **endl** (or **std::endl** without the standard namespace specified):

```
cout << "Hello, world!"<< endl << "How are you?";
```

Note that **endl** is not placed in quotation marks.

A Simple C++ Program

- ▶ No spaces or new lines are printed unless we include the space or new line character as part of our output.
- ▶ The compiler reads code from left to right (most of the time), top to bottom, creating code to execute each statement (terminated by semi-colons).
- ▶ Line breaks and general layout are irrelevant to the compiler, but it cannot be emphasized enough that readability is important for the programmer!
- ▶ Some terminology: we call “hard coded” values **literals**. So:

5.5; // is a numeric literal,
"red"; // is a string literal,
'#'; // is a character literal, etc.

Recurring Code Patterns

The code snippets here will make sense as the course progresses.

A few patterns to watch for:

Variables: variables store data such as numbers or large collections of them, text, etc. There are many different types of variables and we choose a variable type based on how it is going to be used and what functionalities we would like it to have.

Control Flow: telling a computer when to do certain procedures and how many times to do them is managed by checking when various conditions are upheld or being explicit about what range of values we are interested in, etc.

Functions: in programming, a function can take zero or more inputs and give an output (or not). Functions allow for useful processes to take place, such as obtaining the square root of a number or determining the number of characters in a block of text, by writing just a short line of code.

Recurring Code Patterns

Consider a simple guessing game where a random integer from 1 to 10 is chosen and the user is prompted to guess until they are correct.

```
.....  
Guess the number from 1 to 10: 5  
Guess the number from 1 to 10: 6  
Guess the number from 1 to 10: 4  
You guessed the number!
```

An excerpt of the code appears on the next slide.

Recurring Code Patterns

```
// Randomly select an integer value from 1 to 10 inclusive
int randomFrom1To10 = rand() % 10 + 1;

// Variable to store whether the user needs to continue guessing
bool notCorrectYet = true; // start it being true

/* The user will be asked to guess until they are correct */
while( notCorrectYet ) { // while the user has not yet guessed correctly
    cout <<"Guess the number from 1 to 10: ";

    int guess; // value the user will guess
    cin >>guess; // take the user's guess

    if ( guess == randomFrom1To10 ) { // if the guess is correct
        notCorrectYet = false; // update status so loop can terminate
    }
}

cout <<"You guessed the number!"; // tell them they guessed it
```

Recurring Code Patterns

The variables used were **randomFrom1To10**, **notCorrectYet** and **guess**. The first and last were of type **int**, suitable to represent many integer values. The second variable was of type **bool**, to manage **true** or **false** values only.

This was done with the control flow structure using **while** to dictate how long to make the user guess and **if** to check whether the loop could end.

The **rand** function returns a random positive int value. The header file **<cstdlib>** must be included for the **rand** function to be used.

Recurring Code Patterns

Consider a simple program that will allow a user to enter a sentence (or line) and then it will count the number of characters within the sentence.

```
Enter a sentence: A new quarter is upon us!  
Your sentence has 25 characters.
```

An excerpt of the code appears below:

```
// prompt the user for a sentence  
cout << "Enter a sentence: ";
```

```
// define a string variable to store the sentence  
string sentence;
```

```
// collect the entire line of user input  
getline ( cin, sentence );
```

```
// output a count of the chars in the sentence  
cout << "Your sentence has " << sentence.size() << " characters.";
```

Recurring Code Patterns

The variable **sentence** was used to store the user's input. It was of type **string** (not to be confused with a string literal). A **string** is like a souped up version of a string literal, but it has special functions built-in that string literals do not.

We used the **getline** function that accepted two inputs, **cin** and **sentence** and then set **sentence** to the user's input.

We also used the **size** function belonging to **strings** to quickly access the number of characters.

The header file **<string>** must be used to use **strings**.

Summary

- ▶ A computer is an electronic device with the CPU as its “brain”, having primary storage (the RAM and ROM), secondary storage (the hard drive), and tertiary storage (external memory).
- ▶ Source code is first modified by the preprocessor, then converted to assembly language by the compiler, which also performs various optimizations; the assembler turns this code into machine code, which can then be linked with proper definitions through the linker. For the final program to run, it must be loaded into RAM.
- ▶ Code commenting and layout are important for other programmers reading code, but the compiler itself does not notice formatting or comments.
- ▶ **cout** (and **cin**), found in the **<iostream>** header, are useful for managing user input and displaying data to the console.
- ▶ String literals are enclosed in quotation marks.
- ▶ The backslash character can be used as an escape sequence to help denote white space characters and to avoid ambiguities in closing quotation marks, etc.