Welcome to Week 1!

This week you should ...

Required Activities

- Go through the various **videos, movies, and lecture notes** in the **Start Here folder**.
- **Introduce yourself** to the class, see link in the Start Here folder.
- Go through the various **videos, movies, and lecture notes** in the **Week 1 folder**.
- Begin **Quiz 1**. It is due Sunday at midnight.

Recommended (optional) activities

- Attend **chat** this Thursday night, from 8:00 pm - 9:00 pm Eastern Time. Although chat participation is optional, it is highly recommended.
- Post any questions that you might have in the various general **Questions/Comments Discussion Forum(s)** within the course **Discussion Board**. Please ask as many questions as needed, and don't hesitate to answer one-another's questions.
- Download the freely available **Adobe Acrobat Reader** application if you do not have it already installed on your computer, especially if you want to easily read, mark up, and print the weekly lecture notes (in PDF format) posted throughout this course.

"It's one semester mission, to seek out and explore new software ... to boldly go where no class has gone before!"

--- James T. Kurt
Captain, USS Enterprise

Why C is Important

There are many computer languages out there, but one of the most popular languages today is C. C has been around since the 1970's, yet remains one of the most widely used programming languages today. C is built for speed, it is one of the fastest languages in terms of execution. If you need something to run fast, C is your language. If I were arguing a case about why C is one of the most important programming languages today, the following items below would provide the basic facts that would support my case:

- Great Foundation Language to learn first
  - Good mix of **low level** and **high level** features
- **Large code base and support** for a variety of **platforms**
  - Microprocessors to the most advanced scientific systems, and many modern operating systems are written in C.
- Popular Object Oriented (OO) languages **C++** and **Objective C are supersets** of it
  - **C++** provides object oriented functionality with a C-like syntax
  - **Objective C** is ANSI C with a relatively small set of smalltalk-like objects grafted to it. It is considered the most dynamic of the C based OO Languages. It has become extremely popular as many **cell phone applications** are written in this language.
  - Knowing C makes it much **easier to learn OO** since you don't have to worry about much of the language syntax
- **Scripting Languages** (Perl, Python, Ruby, …)
  - **All written in C** and you can add to the language with **C based extensions** (and even embed code within C)
  - Scripting programs can also be compiled and **run as C**.
- **C#** was designed so that it could easily be learned by programmers familiar with C and C++, as well as borrowing key syntax features.
- The **Go** programming language, under development from **Google**, streamlines C's syntax (no semicolons, etc.) with a simpler form of OO than C++ or Java
- Even **Java and JavaScript** have been influenced
- Want to create video games? Many are implemented in C
- Want to install or modify an **Apache Web Server**? You'll need C

Movie - Why Study C

It seems one my students, Debbie Downer, is having concerns about why she should study the C Programming Language. After all, it was created in 1970 and she wonders if she should be studying another language. Watch how Professor Tim easily answers all her concerns and puts her mind at ease. To watch it, ***just click anywhere on the image below***, and it should create a new window to play the movie from a ***private YouTube channel***. It is always best to view PDF files using Adobe Acrobat products, such as the freely available ***Adobe Acrobat reader*** which you can easily download (but most likely already on your computer). Alternatively, you can go back to the lecture notes and there will be a direct link there as well to this same movie. Nearly all of my movies of the cartoon type variety are only 1-2 minutes long and help break up the monotony of wordy lecture notes. It is my hope this semester that you will enjoy watching each of them, if not, at least be mildly amused. Each movie has a specific point that adds to the course and your learning.

Star Trekking - Welcome to the World of C Programming

As an alternative to textual lecture notes, I like to have a little bit of fun and take the learning experience up a notch with a few xtranormal movies I've created using the licensed characters based on the Original Star Trek Series. Each week I'll post a quick 1 to 2 minute "Movie" that briefly shows how the current week's topic is used in a "real life" scenario on the U.S.S. Enterprise. Rather than have me explain it all, let's hear from the Captain of the Enterprise, James T. Kurt. You can watch the Movie by clicking the image below or by returning back to the link in this week's lecture notes. All movies have their own unique links allowing you to access and watch them in real time.

Characteristics of Programming Languages

Programming languages can be generally classified by **programming paradigm**, such procedural programming, object-oriented programming, and functional programming. These paradigms are based on different principles regarding how programming solutions can be modeled and organized.

- **Procedural programming** is based on the concept of a procedure call, which are also called *routines*, *subroutines*, and/or *functions*. Most of the original programming languages like **Fortran**, **COBOL**, and **C** are based on this paradigm.

- **Functional Programming** is mostly about functions calling functions calling other functions where each passes data to each other to get a result. The functions themselves can be categorized as "mathematical" as they get computed to return a result. In this world, there is no such thing as global data, so a function that is passed a specific set of parameters will always return the same computed result. Some examples of languages using this paradigm include **Lisp**, **Erlang**, and **Haskell**.

- **Object Oriented Programming** is all about encapsulating data and behaviors into objects. In a nutshell, for an object, data may be held in *attributes*, while code is held within functions (or procedures) which are commonly known as *methods*. The terminology differs slightly by language, but **C++** and **Java** are good examples, and most modern languages embrace this paradigm.

It should be added that most modern languages, such as **Python**, will embrace the best of all three programming paradigms.

Variable Typing

Programming languages can also be classified by how their variables are typed, whether they're **dynamically typed** or **statically typed**. Examples of types supported by most languages include *floating point*, *integer*, and *character*. **Data typing** refers to what kind of data values a variable (or container) can hold in the course of program execution. **Dynamic typing** allows the container to hold different kinds of values at any time, while **static typing** declares that the container can only hold values of a specific type.

Implementation

How a programming language is implemented at the machine level can provide an important characteristic. Examples of this include **Java** and **Smalltalk** which compile to **bytecode** that is executed in a language virtual machine, while other languages such a Fortran, C, and Go compile to **native machine code**. And then there are languages who do neither of the above which implement instructions directly and freely at run time with an **interpreter**, such as **Perl**, **Python**, and **Ruby**.

Rules

Each programming language has **rules** that need to be followed. You may or may not like the rules, think some of them are dumb, but you need to embrace and follow them for any given language. Some languages have lots of rules, while others provide a lot of flexibility. The compiler or interpreter needs to know these rules up front so it can do its job, and you need to understand the rules of your specific programming language as you code, so you can get things to compile and execute successfully.

Character Sets

Most of what you use to create the code for your program is typed in at a keyboard, but input can be entered in a variety of other methods as well. **Character sets** are the basic building blocks on any programming language. They include letters, digits, white space characters, special characters, arithmetic operators, and punctuation symbols. Most input and output devices support internationally accepted formats such as the ASCII (American Standard Code for Information Interchange) standard, as well as the various extensions that have continued to evolve from it. Of course, one can put items together to extend them as well ... for example, using `>=` to represent a greater than or equal operator.

White Space

White space is any spacing or section of a document that is unused around an object. In a typical document, it helps to separate text, graphics, words, paragraphs and other areas. In programming languages, blank spaces are used to separate variables, keywords, operators, identifiers, and other items in a program.

```
temperature = 32
```

Note in my example the spaces before and after the equal sign which are acting as a delimiter. Thus, we have a variable (temperature), an assignment operator (=), and the value it is being set to (32). You can have more than one blank space in a row, but generally one blank to separate items is sufficient. White space is also used to indent code or skip blank lines to help with readability. It really is not that much different than how I lay out my lecture notes if you think about it.

Keywords

All languages use keywords that implement a specific feature of a language. They are words that are reserved by the language that you can't use for other purposes, such as variable names. For example, in the **C** programming language there are 32 total keywords:

> *auto, break, case, char, const, continue, default, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while*

The **C++** language extends the 32 keywords from C and adds another 30 reserved words.

> *asm, bool, catch, class, const_cast, delete, dynamic_cast, explicit, false, friend, inline, mutable, namespace, new, operator, private, protected, public, reinterpret_cast, static_cast, template, this, throw, true, try, typeid, typename, virtual, using, wchar_t*

Additionally, there are some other **C++** reserved words that support various character sets and provide a few readable alternatives that extend that total. **COBOL** is the language with most keywords (357), while **Smalltalk** is the modern language with the least (6) ... although I'm sure there a few not so popular languages out there that have even less.

Case (Sensitive vs Insensitive)

Some languages like **C** are **case sensitive**, meaning case matters. The statement below has two keywords (if and printf) and a variable called temperature.

```
if (temperature < 32)
   printf ("It is freezing \n");
```

All keywords in C have to be all lower case. You could not substitute IF, iF, or If … for the keyword: if

Variables in C can be any case, but are typically lower or mixed case. For example, given a variable name called temperature … Temperature, TEMPERATURE, or any other combination would be recognized as a different variable. It would be bad practice to have two variables with names that differ only by case (Temperature vs temperature). It would be hard to read and maintain, and prone to error as you will likely mix them up while programming. Additionally, if you decide just because you can to randomly mix the case, that could be confusing as well: TeMPerATUre

Other languages are **case insensitive**, like **SQL**, that don't care about case, their keywords can be upper, lower, or mixed case. All of the below **SQL** statements would do the same thing.

```
SELECT name, organization FROM company;

Select name, organization From company;

select name, organization from company;
```

Yet, just because a language allows you to do things, doesn't mean you should. For readability, I always upper case all **SQL** keywords to make them stand out for readability:

```
SELECT name, organization FROM company;
```

Line Termination

Another characteristic about languages is that they consist of statements, but the compiler or interpreter needs to know when each statement ends. Earlier languages such as Fortran 77 simply allocated a line for each statement … another line meant another statement.

```
outsideTemperature = 32
insideTemperature = 68
```

Most modern languages end statements with a **semicolon** which allows you to put statements over multiple lines. In C, you could combine two statements on one line:

```
outsideTemperature = 32; insideTemperature = 68;
```

... but that is generally frowned upon as one could easily miss some logic. It would be better to write each statement on its own line for readability and clarity:

```
outsideTemperature = 32;
insideTemperature = 68;
```

If your statement is very long, such as in the following SQL statement:

```
SELECT name, organization FROM company WHERE organization =
‘Engineering';
```

It could be rewritten over multiple lines to make it easier to read:

```
SELECT name, organization
FROM company
WHERE organization = ‘Engineering';
```

Comments

Nearly every programming language allows for **comments** which allow a programmer to add wording that will make the program better understood by the

human reader. The key point about comments is that they are ignored by the compiler or interpreter. Comments do not get executed or expand the size of your program in any way. In early programming languages such as **Fortran 77**, a line with a c, C, *, d, D, or ! in column one represented a comment line.

Note: In my lecture notes, I will strive to indicate code in blue and comments in purple. This is not something you have to do in your actual coding, but I find that using color in my notes consistently with code helps students understand what is source code and what is a comment. Many Integrated Development Environments (IDEs) such as *Eclipse* or *Visual Studio Code* will detect and color your code automatically for you. It helps both the developer and the code reviewer.

```
C Set temperature to freezing
temperature = 32
```

It was common in **Fortran 77** to use an *exclamation point* after the statement to allow for an in line comment:

```
temperature = 32   ! set temperature to freezing
```

Later on programs like **Pascal** and **C** enclosed comments in *brackets* ... for example, here is the same statement in C:

```
temperature = 32;  /* Set temperature to freezing */
```

C++ allows for C type commenting, but one can also add // to indicate the beginning of a comment without the need to indicate the end of comment ... it just happens to be the end of the line.

```
temperature = 32;  // set temperature to freezing
```

Or the Ada programming language which uses --

```
temperature = 32;  -- set temperature to freezing
```

A programmer is always striving for that right mix of comments, none or too little makes it hard for someone to understand your program. Too many or too wordy type comments can likewise just add so much "noise" to your code that it can be hard to understand ... examples would include commenting every line, or adding comments that are not very meaningful, such as just telling me what is obvious:

```
float temperature;  /* declare a variable called temperature */
```

As opposed to a comment that is actually explaining what a variable is being used for:

```
float temperature;  /* The spaceships current outside temperature */
```

Conclusion

There are lots of ways to compare and contrast programming languages. The interesting thing is that most modern languages adopted much of their base syntax from the C Programming Language. It is why many consider C a good first

foundation language to learn, as once you know C, it can be much easier to pick up other languages. I have also found that students with limited programming knowledge and experience need to first learn the basic fundamentals that C provides, such as loops, variables, functions, structures, arrays and such, before wrapping their heads around advanced concepts such as object oriented technology, of which most modern languages are based.

TIOBE Programming Community Index

There are many different ways to determine the popularity of computer languages. One way I came up with is to just Google™ each language and see how many hits I got. When I did this, the C Programming Language won by a wide margin. When researching into this a bit more, I came across a reference to a web site that is a great indicator in the popularity of programming language, it is called the **TIOBE Programming Community Index™**. Updated monthly, its ratings are based on a proprietary algorithm that uses the popular search engines/web sites **Google™, Bing™, Yahoo™, Wikipedia™, YouTube™, and Baidu™**. Saying that, they can't make a claim on what is the best programming language, but they can make the case for how popular a programming language is, and whether it seems to be growing or declining in popularity.

I **highly encourage** you to visit the TIOBE Programming Community Index™, and check out the web page for yourself. It will not take up much of your time, and you'll be able to see a list of the most popular languages today and where they have been trending today and in the past. You'll see the popularity of other programming languages and it might help you to determine if you have the right skill sets to complete in today's competitive programming industry.

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

The below table is just one of the many tables and graphs available on this informative web site. It is updated each month and shows how languages are trending in their popularity. In this particular table, it's interesting to see that C has overtaken Java as the top language in the index as of March 2020. Again, please visit the link itself and read through the various charts, graphs, tables, and text to better understand the programming language trends reflecting the current time frame.

Note: *TIOBE has granted permission to anyone to use the information presented on their web site as long as you reference their link (which I have above)*

Here is a table of the top 20 Programming Languages and their trends as of March 2020:

Mar 2020	Mar 2019	Change	Programming Language	Ratings	Change
1	1		Java	17.78%	+2.90%
2	**2**		**C**	**16.33%**	**+3.03%**
3	3		Python	10.11%	+1.85%
4	4		C++	6.79%	-1.34%
5	6	↑	C#	5.32%	+2.05%
6	5	↓	Visual Basic .NET	5.26%	-1.17%
7	7		JavaScript	2.05%	-0.38%
8	8		PHP	2.02%	-0.40%
9	9		SQL	1.83%	-0.09%
10	18	⇈	Go	1.28%	+0.26%
11	14	↑	R	1.26%	-0.02%
12	12		Assembly language	1.25%	-0.16%
13	17	⇈	Swift	1.24%	+0.08%
14	15	↑	Ruby	1.05%	-0.15%
15	11	⇊	MATLAB	0.99%	-0.48%
16	22	⇈	PL/SQL	0.98%	+0.25%
17	13	⇊	Perl	0.91%	-0.40%
18	20	↑	Visual Basic	0.77%	-0.19%
19	10	⇊	Objective-C	0.73%	-0.95%
20	19	↓	Delphi/Object Pascal	0.71%	-0.30%

Visit the TIOBE web site to view various up to date graphs showing the popularity trends of the top programming languages over the last ten years. The latest index has shown that just one new language has cracked the top 10 list, and that is **Go**, mainly due to its internal popularity and usage at Google! Probably the fastest rising language over the years in Python, which is an easy to use scripting language that has some extremely useful library functions in the areas of Big Data, Machine Learning, and Artificial Intelligence. Note that the standard Python interpretor that is used to run Python programs is written in C (a.k.a., CPython).

So, what languages are the most important to know out there? I think C is one of them because once you know C, its fairly easy to understand how other programming languages work. Saying that, you'll get lots of different opinions out there. I thought I would end this lecture note with two videos from two of the top language developers out there. One is **Larry Wall**, who invented the Perl language, an easy to use scripting language that is very popular, and **Bjarne Stroustrup**, the creator of the C++ language, an object oriented version of the C language. Each has a slightly different opinion.

Let's first hear from **Larry Wall**, the creator and face of **Perl**.

Larry Wall: 5 Programming Languages Everyone Should

Watch Video

Know

Duration: (6:13)
User: bigthink - **Added:** 6/13/11

Finally, here is **Bjarne Stroustrop**, the creator of **C++**, discussing his five programming languages you need to know. While a few things are similar between himself and Larry, note there are also distinct differences.

Watch Video

Bjarne Stroustrup: The 5 Programming Languages You Need to Know

Duration: (2:02)
User: bigthink - **Added:** 6/13/11

Operating System

An **Operating System** (Wikipedia: http://en.wikipedia.org/wiki/Operating_system) is the set of programs which make a computer usable by people. It is probably the most important part of your computer system or device, for without it, you are just looking at a set of hardware that really cannot do very much. Most desktop or laptop PCs come pre-loaded with Microsoft Windows™. Macintosh™ computers come pre-loaded with Mac OS X. Many corporate servers use the Linux or UNIX operating systems. The operating system (OS) is the first thing loaded onto the computer -- without the operating system, a computer is useless.

In a nutshell, the operating system contains routines for input/output handling, memory management, interrupt handling, etc. Every general-purpose computer must have an operating system to run other programs. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the screen, keeping track of files and directories on a disk, and controlling peripheral devices such as disk drives and printers.

Here is a good video from YouTube™ that provides a nice introduction to Operating Systems. It should cover this topic with the detail it deserves. You will find a computer device is worthless without an Operating System.

Watch Video

Operating Systems 1 - Introduction

Duration: (3:37)
User: ishaunay89 - **Added:** 5/24/13

More recently, operating systems have started to pop up in smaller computers as well. If you like to tinker with electronic devices, you're probably pleased that operating systems can now be found on many of the devices we use every day, from cell phones to wireless access points. The computers used in these little devices have gotten so powerful that they can now actually run an operating system and multiple applications. It is amazing that today's cell phone has more computer power than the mainframe computers that were the size of a large room from my college days many moons ago.

- **Single-user, single task** - This is the simplest of operating systems that need only provide services for a single user. A good example is a cell phone, such as an iPhone™ or Blackberry™.

- **Single-user, multi-tasking** - You are probably on such a system right now as the best example is a laptop/desktop computer we all use in our day to day life. Whether you are using a Microsoft™, Apple™, or another type of computer, they all allow for multiple applications to be run and processed at the same time (for example, running Microsoft Word™ and sending the output to a printer or FAX).

• **Multi-user** - This type of system allows for many users to connect and access various applications on a computer system. The operating system allocates various resources where it detects and fixes problems so that one user does not drastically affect the resources of other users and/or applications, in effect, it keeps things balanced and running.

Programming Languages

Computers 101

A *computer* performs a given task by executing a series of instructions stored in main memory. Programmers decide what instructions to put in the computer.

- Batch
- Terminal

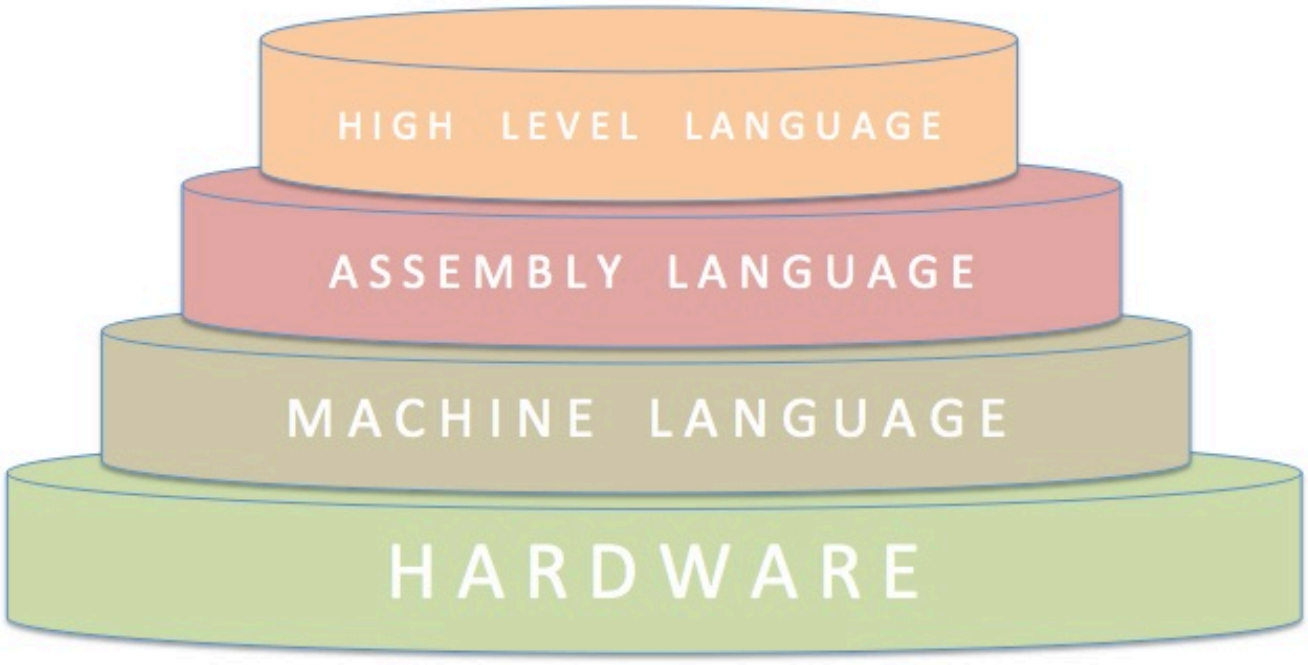
Binary Representation

All data and programming instructions are represented by combination values of binary digits (0's and 1's). Binary digits are stored addressable locations in main memory.

Three Generations of Programming Languages

- First Generation - Machine Language
- Second Generation - Assembly Language
- Third Generation - High Level

As most programming languages are loaded and executed within some portion of computer hardware, you could envision a generational hierarchy of programming languages as follows:



1GL - Machine Language

Machine Language is the language that the computer understands. Instructions are stored as binary digits in the computer. An example may be:

5B4 201

which means subtract the contents of the storage location 201 from the contents of register 4.

Machine Language Disadvantages

1. Tedious
2. Difficult to correct/maintain
3. Difficult to modify
4. Generally not portable, can only run on a specific target machine (an exception would be if both machines are running the same processor, such as an Intel)

A simple **advantage** is that it is very fast.

2GL - Assembly Language

Assembly Language was developed to handle the shortcomings of machine language. Like machine language, the programmer must prepare a separate instruction for each operation to be performed by the computer. However, instead of binary digits (or hexadecimal representation), assembly language uses more meaningful symbols. In the simplest case, each assembly language statement specifies two things:

1. The **operation** to be performed
2. The **operand**, which is the symbolic name for the data to be used in the operation

EXAMPLE: **WRITE TOTAL**

Two important features of assembly language

1. Operations are verbs (or mnemonics) that are easy to understand by humans
2. Operands are meaningful data names that are used to stand for computer addresses

Disadvantages of Assembly Language

While assembly language is fast and easier to understand than machine language, it has the following disadvantages:

1. Difficult to learn and understand
2. Coding is slow and tedious
3. Difficult to modify programs and to detect and correct errors
4. Many Assembly languages exist for various computer systems
5. Not Portable between different computer systems

Assembly and Machine Languages are geared to the computer rather than people. They do allow programmers to write efficient programs, but programs take much longer to develop.

3GL - High Level Languages

High Level Languages are programmer oriented with the instructions in procedural form of the problem to be solved. They were created to improve productivity of writing programs. Many have been created over the past 40 years.

- COBOL (1960's, Business Language, represents 70% of the world's code)
- FORTRAN (1960's, Scientific)
- C and PASCAL (1970's)
- C++ and Ada (1980s, Object Oriented and Object Based)
- JAVA, JavaScript (1990's, Object Oriented and built for the Web).
- C#, Objective C (2000's)

Advantages

- Portable - Can be run on many different machines. Code once and deploy over many machines
- Improved Productivity (Easier to understand and fix)
- Improved Maintenance
- Easy to Learn

Compilers and Equivalent Machine Instructions

Computer Languages are translated to ***Machine Language*** (Wikipeida: http://en.wikipedia.org/wiki/Machine_code) by compilers and assemblers. There is a bit of information in the first few chapters in your textbook on this, but I've always enjoyed this video that does a nice job explaining compilers in an "old school" but fun way. It also talks about Interpretors which as popularly used on many scripting languages such as Python, PHP, Perl, and Ruby these days. I think you'll get more out of watching this than "pages and pages" of written lecture notes on the topic.

Watch Video

Interpreters and Compilers (Bits and Bytes, Episode 6)

Duration: 3:36
User: n/a - **Added:** 5/17/12

YouTube URL: http://www.youtube.com/watch?v=_C5AHaS1mOA

Execution time for a program can often be measured by how many machine instructions are executed over time. A computer language's speed of execution can be measured by a factor that determines the ***average number of machine instructions*** that each ***higher level language*** would generate.

For example, an average C instruction is equivalent to 2.5 machine language instructions. *COBOL* is 3, *Assembly Language* is 1.5, database languages are 8, and some *spreadsheet macros* and languages are 50. This adds up when you compare a 1000 line C program (2500 Machine Language Instructions) verses a 1000 line COBOL program (3000 Machine Instructions). This is often referred to as **Equivalent Machine Instructions (EMI)**.

As a software engineer, we are often ask to estimate how many lines of code a task will take, as well equivalent machine instructions that can be used for planning performance estimation and tuning. Let's look at an example in C: Say you want to estimate how many machine instructions would be ***executed*** in the following three C statements:

```
    printf ("Enter a temperature: ");
    scanf ("%f", &temperature);
    printf ("The temperature converted to Celsius is %8.2f \n", (temperature - 32.0) *
(5/9)   );
```

... it would give you an average of 7.5 statements (calculated as 3 statements * 2.5 EMI factor for C). That seems simple enough.

However, let's look at the same statements within a loop (which we will learn about next week):

```
    for (i = 1; i < 100; ++i)
    {
        printf ("Enter a temperature: ");
        scanf ("%f", &temperature);
        printf ("The temperature converted to Celsius is %8.2f \n", (temperature - 32.0) *
(5/9)   );
    }
```

Even though you see four statements (you can ignore the curly braces), the actual EMI ***executed*** during the life of the program would not be 10 (figured as 4 statements * 2.5), but would be MUCH more than that! We will learn more about loops in a few weeks, but these 4 statements would each be executed 100 times. The estimated executed EMI for the statements above would be (100 * 4 statements * 2.5) or 1000 equivalent machine instructions.

In this class, I will often stress how following a few simple guidelines and rules can have a big effect on the performance of your program. We may be talking about microseconds or seconds, but ask yourself this question: What effect would a few extra seconds have if you were using a device running software on the following real life situations?

1) A delay with the Patriot Missile software trying to take out an incoming SCUD missile
2) A delay in activating your anti-lock braking system on your car
3) A delay in activating a medical device based on a specific life threating event

... I think you get the picture that it's important to not only get your program to run correctly, but just as important that its performance be optimal. In computer science, we often refer to this as ***efficiency***. As defined in Wikipedia, Efficiency "describes the extent to which time, effort or cost is well used for the intended task or purpose. It is often used with the specific purpose of relaying the capability of a specific application of effort to produce a specific outcome effectively with a minimum amount or quantity of waste, expense, or unnecessary effort."

Higher Level Languages produce a higher ratio of source statements to machine language statements when compiled.

Note that while C is considered high level, it provides features that allow the user to get close with the hardware and operating system. It deals with the same sort of objects that most computers do: characters, numbers, and addresses.

The Birth of C and UNIX

Both C and UNIX have similar backgrounds; you could consider them both part of the same family. In fact, the UNIX operating system that is widely used today was originally written in C and much of it today remains implemented in that language. Let's quickly look at how both C and UNIX came into existence. UNIX and C are still very popular today. For those of you who may not have heard of UNIX, I bet you have heard of LINUX which got its start from UNIX. In my humble opinion, LINUX is the best server out there, especially for web applications. You can run a LINUX server for weeks and months within a production environment without many issues. To learn more about this Operating System (which is out of scope for this class), check out http://www.linux.org/article/view/what-is-linux

Genesis of C

C is really not a new language. It has been around since 1970. Let's look at a brief history of the language and how it parallels the **UNIX Operation System** (Wikipedia: http://en.wikipedia.org/wiki/Unix)

- 1960 - ALGOL 60 Language
- CPL Language
- BCPL (Basic Combined Programming Language)
- **Ken Thompson** (Wikipedia: https://en.wikipedia.org/wiki/Ken_Thompson)
    - Looks for better language for UNIX
    - Rejects **FORTRAN** (Wikipedia: http://en.wikipedia.org/wiki/Fortran)
    - Condenses BCPL to B
- 1970 - **Dennis Ritchie** (Wikipedia: http://en.wikipedia.org/wiki/Dennis_Ritchie
    - Modifies B and C is born
- 1977 - UNIX is moved to INTERDATA 8/32
    - UNIX is moved to IBM 370
- 1988 - **ANSI C** Standard Developed (Wikipedia: http://en.wikipedia.org/wiki/ANSI_C)
    - There were many flavors of C

C was originally created as a language in which to write operating systems. It is a small, fairly low level language. It lacks input-output commands, string handling commands, and has no exponentiation operator. All of these things are done by functions.

ALGOL (Algorithmic Language) had more of a following in Europe than in the United States. In Europe it was often used instead of FORTRAN

For a good overview on C, with lots of information on its history ... check out this web site on Wikipedia:

- http://en.wikipedia.org/wiki/C_programming_language

Check out this entertaining link on YouTube for a nice tribute type video called "Write in C" sung to the Beatle's tune "Let it Be":

Watch Video

Dennis Ritchie - Write in C

Duration: (3:38)
User: leoyamasaki - **Added:** 10/13/11

Genesis of UNIX

Let's take a quick look at the **UNIX Operation System** which will give us the background of why C was developed.

- 1965 - MULTICS (means "Multi User") Operating System being developed by GE, MIT, and BELL Labs of AT&T.
- 1968 - AT & T withdraws from the MULTICS project
- 1969 - **Ken Thompson**
  - Needs computer for his video game "Space Travel"
  - Finds unused DEC PDP-7
  - Creates new one person operating system in Assembly language
  - Calls it UNIX instead of MULTICS
- 1970 - **Dennis Ritchie**
  - Joins in enhancing the operating system
- 1971 - UNIX Moved to PDP-11
- 1972 - **UNIX rewritten in C** from PDP-11 Assembler
  - Results were very positive

Wikipedia provides a great starting point on UNIX, which is really beyond the scope of this class, feel free to check it out at:

http://en.wikipedia.org/wiki/UNIX

Here is a nice radio podcast (about 6 minutes) from a BBC Radio program called the "Last Word" that talks about how both Ken and Dennis created the UNIX Operating System. It is well worth your time to listen to it.

Watch Video

Tribute to Dennis Ritchie, from BBC Radio 4

Duration: (5:59)
User: sambeep - **Added:** 10/22/11

Here is an interesting video of both Ken and Dennis discussing the UNIX operating system.

Ken Thompson and Dennis Ritchie Explain UNIX (Bell

Watch Video

Labs)

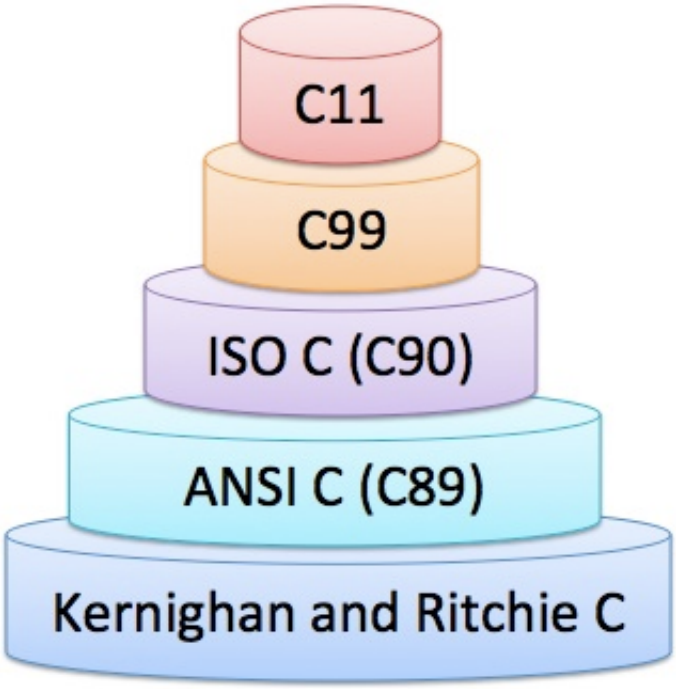
Duration: (2:19)
User: vortextech - **Added:** 4/10/11

Sadly, Dennis Ritchie passed away recently. CNET.com has a good article about his life and times (as well as the picture I inserted above) that you can review at:

- Rest in Peace Dennis Ritchie

The Evolution of C

The C Programming Language has undergone some changes over the years, but not as much as you might expect for a popular language that has been around since 1972. Let's take a look at some important accomplishments and milestones below in the evolution of the C language. Before continuing, it should be noted that much of the information below was based on research conducted by Wikipedia where you can view specific details at https://en.wikipedia.org/wiki/C_(programming_language). I "cherry picked" the best parts of the work and added my own observations as well from personal experience in using all of these versions over the years.



Kernighan and Ritchie C

In 1978, Dennis Ritchie and Brian Kernighan published the definitive book of its time titled "The C Programming Language", which is still published today. In it, they introduced a few new concepts into the C language such as long integer and unsigned integer variable types, the standard input/output library, and compound assignment operators. At a minimum, all existing C versions mapped to what was fast becoming known as the "Kernighan and Ritchie" standard, or K&R.

From there, they continued to add a few more features over the years, but they also found that other competing vendors were also modifying the base C language and extended it through new features or library modifications. This resulted in the generation of many flavors of C due to the efforts of competing vendors and compilers, such as K&R C, AT&T System 5 C, and IBM C ... just to name a few. As a programmer

during this time, you had to reasonably expect that your program might not work the same if you tried to port it to another type of machine that might also be utilizing a different compiler flavor.

ANSI C (C89) and ISO C (C90)

In the 1980's, work began on the C Programming Language to promote a common standardization of all the flavors. The American National Standards Institute (ANSI) ratified a standard for *ANSI C in 1989*, which is also known as *C89*. In the following year, the International Standards Organization (ISO) made a few modifications and adopted it as ISO/IEC 9899:1990, which you might know as *C90*.

Both of these standards utilized K&R C as the base standard with the goal of integrating all the other useful features developed by other vendors into a commonly accepted and standard C programming language.

C99

The C language went through a long period of relatively few changes until ISO put out what is commonly referred to as the *C99 standard.* It introduced some new features such as the *long long integer type*, acceptance of *C++ style comments*, and *variable length arrays* to name a few. Another feature that has become popular is the ability to declare variables whose scope is only known within a specific loop or conditional statement body. We'll investigate that interesting feature in depth over the next couple of weeks. Check out specific details on C99 at: https://en.wikipedia.org/wiki/C99.

C11

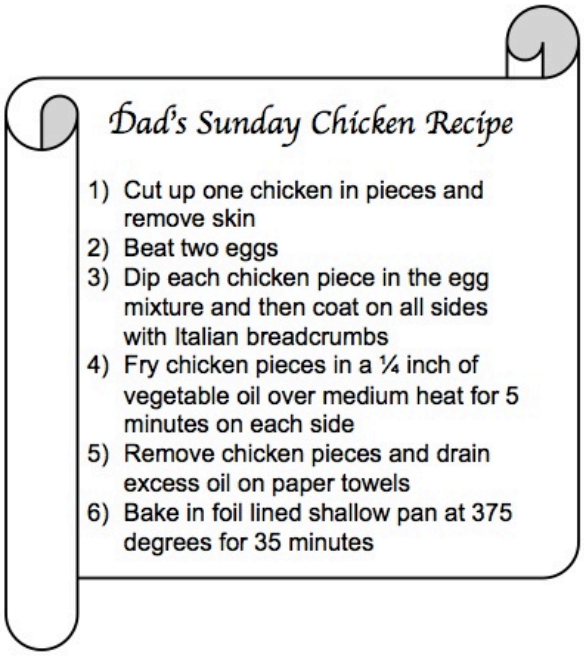
C went through another long period of little change until there become a need to standardize yet again as competing vendors were adding new features to help keep the language up with the modern times to support concepts such as *multi-threading*, *bounds checking functions*, and *improved Unicode support*. This was known to the world as the *C11 standard*. If you want to learn more about C11, consult https://en.wikipedia.org/wiki/C11_(C_standard_revision).

Summary

C has gone through a few changes over the years, but remains a relatively stable language in terms of the number of versions that have evolved. Some older C compilers may not implement everything or even incorporate parts of these standards, but if you are using a modern compiler and plan to code in the real world sometime in the future, then it is a good idea to familiarize and incorporate their features and capabilities to get the maximum language benefit. I will be calling out some of the features of the ***C99 and C11 standard*** from time to time as they apply to specific topics being covered in the weeks that follow.

Programming in C

Computer programming is much like working with recipe to create to create a snack or fancy dinner. While some recipes are simple, others are more complex. Below is a sample recipe my Dad would make nearly every Sunday to feed our family.



Since I had a lot of brothers, it would be served with Chicken Rice-A-Roni™ and corn. Most cooking recipes are simply a series of steps that need to be completed. Before we get into some specific details, listen to Larry Wall, the creator of the Perl Programming Language, as he explains computer programming in about 5 minutes.

Larry Wall: Computer Programming in 5 Minutes

Duration: (5:23)
User: bigthink - **Added:** 6/13/11

Let's look in on a specific problem to solve in the real world. What if I asked you to write a program in C to figure out if a number was Odd or Even? How would you start? What would you do? I've sketched out some simple steps you might take to get started.

Step 1: Understand the Problem to be solved:

- *Is a number **odd** or **even**?*

Step 2: Design an Algorithm

- *If a number can be divided by 2 with no remainder, ... it is **even,** otherwise it is **odd***

- ***Sanity Check:*** *5/2 is* ***odd****, there is a remainder of 1, 6/2 is* ***even****, there is no remainder*

- *The* ***modulus*** *operator in C (a % symbol) returns the remainder (we'll cover this next week, see Chapter 4).*

Step 3: Code the Algorithm

```
if (number % 2 == 0)  /* no remainder */
   printf ("It is Even \n");  /* even */
else
   printf ("It is Odd \n");  /* odd */
```

Summary

Before you start coding your program, you must do two things:

- **You must understand the problem to be solved before you can devise an algorithm.**
- **You must devise an algorithm before you can implement it in the source language.**

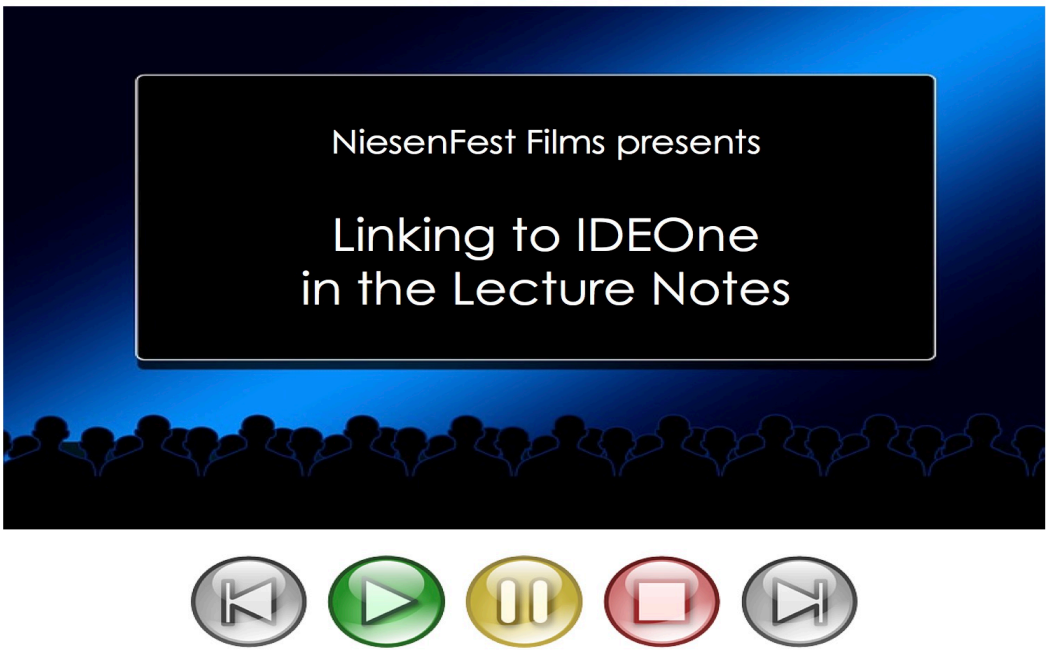
One of the biggest mistakes I see with novice programmers is they simply start writing code without any idea of the problem they are solving, as well as any thought of a workable design to solve it. Before coding, spend some time understanding the problem to be solved as well as a design that can be used to properly implement your program. In the real world, coding should only be about 15-20% of the total effort to deliver a complex system or application. The rest of the time should be spent on the architecture, requirements, design, test, integration, and validation of the system.

Video - IDEOne Overview

An introduction and quick tutorial on how to use the IDEOne application, a freely available application of the Internet that will allow you to create, run, and share simple programs implemented in many computer languages. This video is purely for educational purposes only.

Video - IDEOne Linking from Lecture Notes

This video shows how easy it is to access the many code examples in the class lecture notes as they contain active web links to IDEOne. Learn how to take advantage of this feature so you can quickly access and run code shown in the notes. You will also learn how you can then make modifications to the code and run your own code examples. The best and only way to learn coding is through hands on experience. Will you make some mistakes? ... of course you will, but you will learn a great deal from correcting those mistakes and doing your own "What Ifs".

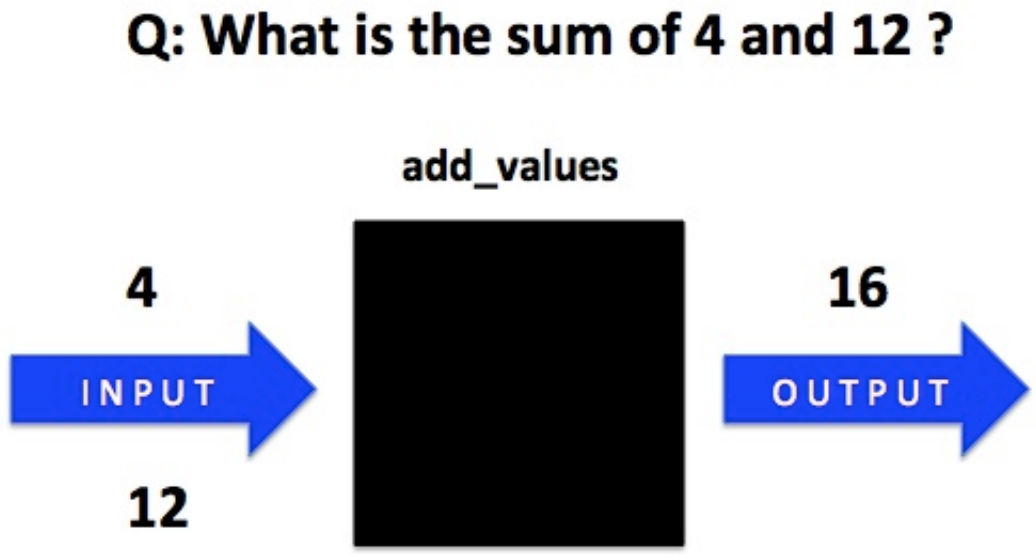


Functions

Your program may use existing **functions** from the **C Standard library** or you may create your own. Functions are like subroutines in other languages.

- A Function does **ONE thing**
- A Function has **well defined INPUT**
- A Function has **well defined OUTPUT**

To illustrate how a function works, below is an example of how one could design a function to add two integer values together and have the function return its sum. I created a black box design that would contain the actual code to implement the function. There is a specific code syntax to implementing functions and we'll cover them in great detail in Chapter 8. For now, just assume you can call a function and pass it some values and it will return a specific result.



Once a program designed to do one function has been written and tested, it is stored in a library with other functions.

Vendors and third party sites also provide libraries that can be linked into your program.

Anyone who needs one of these functions can use them.

Some examples of C library functions include sqrt (square root), printf, and scanf.

When we cover functions later in this class, you find out that a C program is really a bunch of functions that call other functions that starts with a function called main.

In summary, there are functions available in the standard C Standard Library that can be called, there are functions from third party vendors or sites that can be linked into

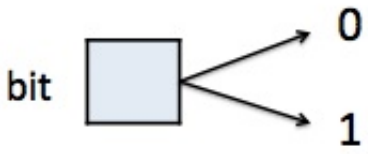
your program (for example, Oracle provides a set of library routines that can be used to connect and access an Oracle database using C), and there are functions that you as a programmer can create. We'll cover many of the available standard C Standard Library functions in this class, and show you how to properly create your own functions.

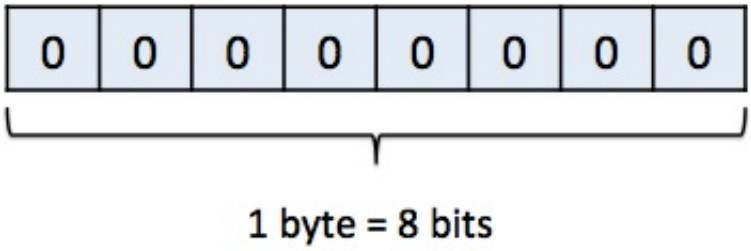
For details on the C Standard Library, once again, check out Wikipedia:

http://en.wikipedia.org/wiki/C_library

Computer Memory

Here are some basics on computer memory in relation to the C Programming Language. The smallest unit of memory is a **bit**. It can only contain a value of 0 and 1. Think of it like a light switch that you can turn either ON or OFF. In the world of computer science, a value of 0 is FALSE, and a value of 1 is TRUE.





If you group 8 bits together, it forms a **byte** of memory. Byte locations are **addressable** and the C Programming language can access memory at those locations. One of the smallest data variable types in C is a type of **char**, which consists of one byte of memory. From a byte location, C also supports a wide range of bit operators and functions (see Chapter 12 in your book).

Most modern computers use an 8 bit byte and a 32 bit (4 byte) word for storage of data. 64 bit words are becoming the norm on many computers today.

Below is a very good video on YouTube that explains the history, current state, and future of computer memory. It even references the old punch cards I used to use in my college days - I said it before, and I'll say it again ... "Boy, do you folks have it easy today."

Watch Video

What is Computer Memory?

Duration: (5:24)
User: artoftheproblem - **Added:** 11/29/12

Program Structure

Let's take a quick look at the overall structure of a simple C program. C is simply a bunch of functions calling other functions that starts with a function called main. All C programs can be made up of as many functions as you need, but at a minimum, a C program must contain a main function, so it knows where the execution starts and finishes.

```
int main ( )

{

        /* some statements */

}
```

This diagram shows the start of the C Program structure.

- C doesn't care which column you start your statements on.
- The open and close braces { and } are like begin-end constructs in languages like Pascal and Ada.
- C is case sensitive, MAIN is not main.

Standard C Program

Below is the typical format you should use for all C programs developed this semester. The first program every new C Programmer creates is Hello World. It simply prints "Hello World" to the screen. Feel free to try it out using IDEOne at:
http://ideone.com/pcg8l9

```
#include <stdio.h>

int main ( )
{
    printf ("Hello World \n");

    return (0);

}
```

In the program above there is a **function** called **main** because every C program must begin with a function called main (not Main or MAIN, case is important). Inside the main function you will find a function called **printf** that is a function in the C library. Its function definition can be found in the **header file** called **stdio.h** (an actual file whose contents are made known to your program). The **"\n"** in the printf statement is a **special character** that is used to indicate that a **carriage return** (new line character)

will follow the display of **Hello World** on the screen.

It is important to also have a **return statement** since the main function returns an integer value (we'll cover this later, but take it on faith for now). The return statement in our example will return a value of 0 back to the calling process, which in most cases is the operating system. A returned value of zero implies that the program terminated normally. Non-zero values can be returned in other places within your program to indicate that a program terminated abnormally.

Important Points

- A **semicolon** is used to end each C statement.
- The **\n** is a **special character** which causes the output device to do a carriage return and a line feed. It looks like two characters (a backslash and an n), but is considered to be one character.
- Since **main** is also a function it has parentheses - even though no values are passed to main in our example. Note that C is case sensitive, all C programs must start with a function called main, not Main or MAIN. You could also write it as: **int main (void)** ... which is actually the proper way to write it as **void** indicates that the function has no **parameters** that need to be dealt with (more on this when we cover functions in detail). Most of you will likely be using a C++ compiler, where **int main ( )** is the norm and **void** is frowned upon as its use is redundant and unnecessary.
- **printf** is a function in the **C Standard Library** that simply prints or displays its argument to STDOUT (standard output), which is usually the terminal screen.
- The **#include <stdio.h>** statement provides common types and constants need to make I/O routines like printf, scanf and other C Library routines work correctly. Note that it does not end with a semicolon and the **#** must be in column 1 (at least for older compilers). Take a look at the chapter on input/output for more information.

Comments

Comments within any source code are ignored by the compiler and are not translated into equivalent machine instructions. They are used to provide documentation initially to both the developer (and their team), and any maintainers of your program in the future. Comments can appear in any column and start with the **delimiters**

```
/*
```

... and end with ...

```
*/
```

Comments help the person reviewing the code to determine the original programmers *"intent"* of what they were trying to do. It helps one to better understand exactly what a section of code is trying to do. In this class, I will expect a fair amount of comments to be added to your programs on your homework and exams. However, there is a fine line between too many comments (commenting every line of code ... it makes your code unreadable) and just enough comments. Another suggestion with comments is to not make them take up more than 80 characters on a single line, so you can easily read them on-line (without horizontal scrolling) or print them out. Try it out at: http://ideone.com/zGmbUn

```
#include <stdio.h>
int main ()
{


    /* This is a block comment,
        it may extend
           over
        many lines */

    /* this is another comment */

    printf ("Hello World \n");  /* this is an inline comment */

    return (0);
}
```

Brian Kernighan (yep, the same person who co-authored the C Programming book with Dennis) wrote a very useful book for programmers called: *The Elements of Programming Style (**ISBN-13:** 978-0070342071).*

A quick summary of its lessons along with its ISBN reference can be found at Wikipedia at: http://en.wikipedia.org/wiki/The_Elements_of_Programming_Style_(book).

I would at least like you to take a quick look at this link and feel free to refer back to this

page now and then to help you write better programs. There are 56 one line "Words of Wisdom" that represent some important programming lessons to understand that will help both the novice and experienced programmer.

Note:

Some older compilers may have limits on how many characters you can place between comment tags. Keep this in mind if you are working on maintenance of an older system. You can fix this by having the start and end comment tags on the same line, and then span the words on multiple lines. Here is an example:

```
/* This is a block comment  */
/* that spans over many     */
/* lines.                   */
```

You can also use **C++ style comments**. C++ is an object oriented extension of the C Programming Language that you are welcome to study after this class. Any C program will compile and execute successfully within a C++ compiler and environment. Saying all that, because you will likely be using a C++ compiler, feel free to optionally use the C++ style comments.

Each line of a C++ comment must start with a single start tag delimiter: //

```
// This is a block comment
// that spans over many
// lines.
```

Note that you do not need an end tag delimiter like found in traditional C (i.e., */). Most people prefer working with the C++ style comments. Again, while both will work within a C++ environment, be consistent, don't mix and match different comment styles (C and C++) within the same program.

Tip

Another common usage of comments is to **disable code**. Perhaps you are having trouble with compiling a section of code that is giving you a lot of syntax errors or just isn't logically working correctly.

Here is how you comment out three lines of code. These three lines of code will be part of a comment. The important thing to understand is that any item with a comment is **ignored** by the compiler. Therefore, the three lines of code you see below will 1) Not be checked for syntax errors, 2) Will not be translated to machine code upon a successful compile and linkage of your program, and 3) Will not be executed in any way when the program is run. I've keep all the code below in PURPLE rather than blue to indicate that its all a comment.

```
/*
printf ("Enter Diameter and Perimeter: ");
scanf ("%f %f", &diameter, &parameter);
```

```
area_of_circle = 3.14 * diameter * parameter;
*/
```

To have the compiler recognize the three code statements above, all you would need to do is remove the beginning and ending comment tag (/* and */). Using a comment like this is easier than adding a beginning and ending comment on each line. In that case, you would have delete the comments on each line. I would not recommend commenting out code like it is shown below because it can be 1) Time Consuming to delete each comment tag, and 2) Easy to miss a tag or two in a large group of statements.

```
/* printf ("Enter Diameter and Perimeter: "); */
/* scanf ("%f %f", &diameter, &parameter);  */
/* area_of_circle = 3.14 * diameter * parameter; */
```

With comments removed, the compiler will now recognize these three lines of code below and run them through the compilation process. Note that I now have them displayed in BLUE to indicate they are code statements to be compiled, not ignored as a comment.

```
printf ("Enter Diameter and Perimeter: ");
scanf ("%f %f", &diameter, &parameter);
area_of_circle = 3.14 * diameter * parameter;
```

Your First Program

Let's take a stab at developing your first program in C. Try to write a simple C program to does the following:

EXERCISE 1: WRITE a C Program with one printf statement which prints:

HELLO

WORLD

EXERCISE 2: Do it instead with two printf statements

... ***answers*** are on the next lecture note page.

Exercise Answers

Here are the answers to exercises 1 and 2:

Exercise 1 Answer: ***http://ideone.com/NaLKWC***

```
#include <stdio.h>
int main ( )
{

    printf ("HELLO \n WORLD \n");
    return (0);

} /* main */
```

Exercise 2 Answer: ***http://ideone.com/yQee60***

```
#include <stdio.h>
int main ( )
{

    printf ("HELLO \n");
    printf ("WORLD \n");
    return (0);

} /* main */
```

If you are using Microsoft Studio

Microsoft Visual Studio is a great tool as it is a fully **integrated development environment (IDE)** that lets you create, compile, test, and display simple to complex programs. With it, you can work with the C, C++, C#, and Visual Basic programming languages. One initial issue with using a tool like Microsoft Visual Studio is that output will print in an output window and then quickly disappear. To stop that from happening, you could put one of two commands just before the return statement: Either a **system command** using the **pause** command, or using an input/output library routine called **getch**.

Please note: You do not have to use Microsoft Visual Studio or Microsoft Visual C++

```
#include <stdio.h>
int main ( )
{

    printf ("HELLO\nWORLD\n");

    system ("pause");   /* pause for a few seconds */
```

```
    return (0);

} /* main */
```

Here is one using getch ... which I feel is better because the output will will remain on the screen into you hit any character on your keyboard.

```
#include <stdio.h>
int main ( )
{

    printf ("HELLO\nWORLD\n");

    getch ();   /* wait until a character is entered by the user */

    return (0);   /* success */

} /* main */
```

If you want to use Visual Studio/C++, here is a good tutorial on YouTube. While the author writes his code as C++, just substitute my code above and it will work just fine. You can create a C++ file (it has a *.cpp extension) within Visual Studio and add your C program there to compile and run. Remember, since C is a subset of C++, it can run on any C++ compiler. The initial stumbling block for new users is all the initial steps needed in Visual Studio to get to the point where you actually add your C code. Hopefully, this video provides some clarity. There are also ways to simply run it as a C Program as well.

Watch Video

Visual C++ 2012 - Tutorial 01 - "Hello world!" - C++Tunisia (English)

Duration: (4:27)
User: cplusplustn - **Added:** 7/21/12

Simple Web Compiler Option

A quick way to try this out is with a web site called codepad (http://www.codepad.org). All you need to do is cut and paste or type in code into it and select a program type of C (it works for many other languages).

A better alternative to codepad is IDEOne (http://www.ideone.com), since it adds the ability for you to specify input that can be read when your program is executed. With both codepad and IDEOne, just remember to select the language C up front when you compile/run your program. Its all web based and works in any web browser ... VERY EASY to use!

I would recommend that you try both codepad and ideone and get the two programs working in those environments.

To get the full development experience, I would recommend that you download a free

C++ compiler (it can compile and run C) such as Visual C++ Express from Microsoft. There are other freely available C/C++ compilers out there as well (in case you use LINUX or other operating systems). Note that a C++ compiler can compile and run any C program ... you will probably not find a C compiler out there, but there will be plenty of C++ compilers out there.

To do the homework assignments and the exams, you need to either use a full compiler, or IDEOne, since codepad does not handle input.

Compiling and Building Programs in C with gcc

The are many compilers out there that will work with C and C++. One of the most popular ones out there is the **GNU Compiler Collection (GCC)**, which is a collection of programming compilers. It is commonly available on UNIX and LINUX systems, and you can even find versions to run on most systems. Let's look at **gcc** to help us understand how to compile and work with multiple files. If you have just one file to compile, **gcc** would do all four steps of the compilation process (preprocessor, compiler, assembler, and linker) to create an executable. The key point here is that you must have at a minimum a function called **main**, as any C program is essentially a bunch of functions calling other functions that starts with a function called "**main**". Let's take a look at each of the four C compilation steps below:

- **Step 1 (Preprocessing)** - The first step which is somewhat unique to C will pre-process the source code, that begins with the # directive, initially by expanding include files (such as stdio.h), removing comments, and expanding macros, the latter of which we will cover in a future week.

- **Step 2 (Compiling)** - The second step will take the C source code that has been preprocessed and generate assembly language, which is a human readable language specific to the target machine. A statement in C will generally translate to many equivalent assembly instructions. Below is a simplified translation to assembly language of our initial conditional statement along with comments to the right. There are various flavors of assembly language out there.

- **Step 3 (Assembly)** - The third step will take all the assembly code and covert it to machine code, a set of zeros and ones which is better known as object code. If you are a computer or robot, you'll have no problem figuring out the code that is actually being processed ... as now the code is in a form a machine can quickly understand and process. I am greatly simplifying things here, but you don't want to be reading machine code as it is not viewed very well on your screen as text.

- **Step 4 (Linking)** - The fourth and final step will link all our object code together, incorporating various functions and libraries into an executable file that you can invoke to run your program. There are two types of linking methods that can be integrated into your executable. In ***static linking***, library object code as needed is copied into and lives within your executable. With ***dynamic linking***, only information about what is being linked is placed in your executable, whereby, at run time, the executable will retrieve what it needs. Don't be too concerned at this point about static vs dynamic linking. In our last lecture week, I'll show you how to create, link, and utilize both static and dynamic libraries.

Four Step C Compilation Process - Hello World Program

Remember our first program? The **Hello World** program? It includes a call to **printf**, which is part of the standard C library. The function prototype for **printf** can be found in the stdio.h file. On most UNIX systems, header files can be found in the directory: ***/usr/include***

Most of the library files to compile your program on a UNIX machine are found in ***/usr/lib*** or ***/usr/local/lib*** ... but it varies depending on the administrator who is setting up the system. The file that contains most of the C standard library functions is called" **libc.a**" (if static linking is used) or **libc.so** (for dynamic linking). It contains pre-compiled functions stored in corresponding **object files** (a file that is in a compiled state) and stored them all together in a container we call a **binary archive file** (hence, the *.a or *.so) extension.

Thus, when you compile your program, a C compiler like **gcc** will link in that **libc** file automatically by default so it is available with the code you have developed to link them all together to create an executable if everything is successful. Given our very simple C program below:

```
#include <stdio.h>
int main ( )
{
    printf ("Hello World\n");
}
```

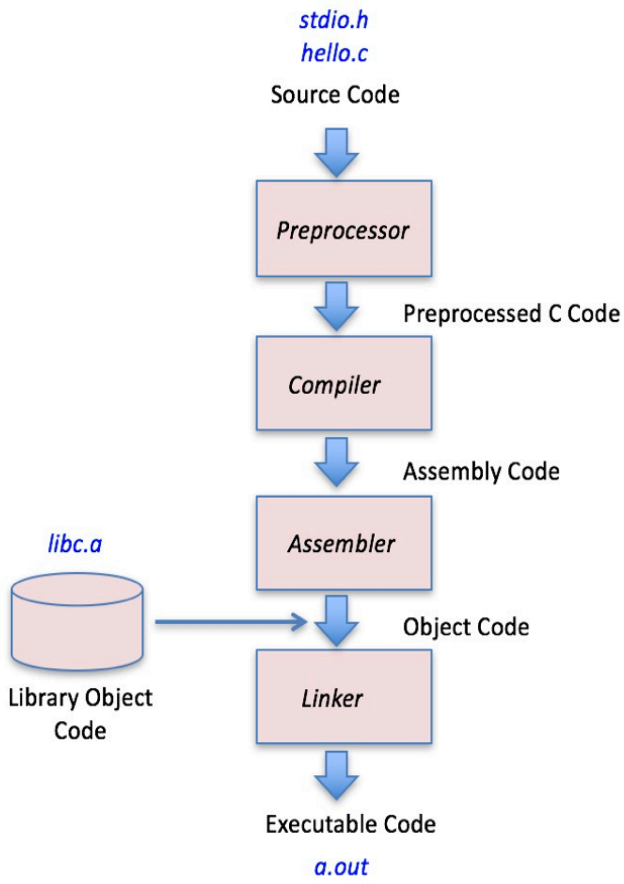
You could use **gcc** to compile the program in the following manner:

```
gcc hello_world.c
```

Alternatively, it would be equivalent to:

```
gcc hello_world.c -lc
```

Where **-lc** would be shortcut for linking in the **libc** library. When you use the -l option with **gcc** to indicate a library to include, the "**lib**" in **libc.a** (or **libc.so**) is not required, as well as the file extension part: (***.a** or ***.so**). Yet both **gcc** statements above are equivalent as the object files in the C Library are automatically included by default, whether you name the **libc** library file or not. This will compile the source code in the file **hello_world.c** first, and if successful, will link in the standard C library (**libc**) by default, which is where the compiled **printf** function is found. It will successfully compile, build, and create a default executable called "**a.out**" that can be used to run your program. Below is an illustrated view of this "compile and build" process in C:



Below is a sample session on a UNIX system, from the command line, compiling and building the Hello World program. We have a single file called **hello.c** in my directory, and you can use the "**ls**" UNIX command to show all files in a directory. When I compile the file **hello.c** using **gcc**, a default executable file called "**a.out**" will be created (assuming everything compiled and built correctly) that I can use to run the program at the command line.

gcc hello.c

```
> ls
hello.c
>
> gcc hello.c
>
> ls
a.out  hello.c
>
> ./a.out
Hello World
>
>
```

You can also redirect an executable file using the -o option with gcc.

gcc -o hello.exe hello.c

Other Useful Tips

There are many available options that you can run with **gcc**, for a full list, just do an Internet search on:

gcc C Compiler Options

Should you want to see the translated assembly language of your C code, you can use the **-S** option with **gcc**.

gcc -o hello.exe -S hello.c

It will create a file called **hello.s** that you can open up and see the assembly code.

Even better, if you want to see the C code and associated translated assembly code together to help you follow along easier, try:

gcc **-Wa,-adhln** -g hello.c > hello.s

The **-Wa** option for **gcc** will pass multiple options to the assembler (**-adhln**), add useful debugging information (**-g**) and then redirect (using ">") to a file called **hello.s** which you can then view using your favorite editor (as it is just a text file).

If you wanted to convert your C source code to corresponding compiled code format (known as an **object file**), you can do:

gcc -c hello.c

It will generate an object file, which has a *.o file extension:

hello.o

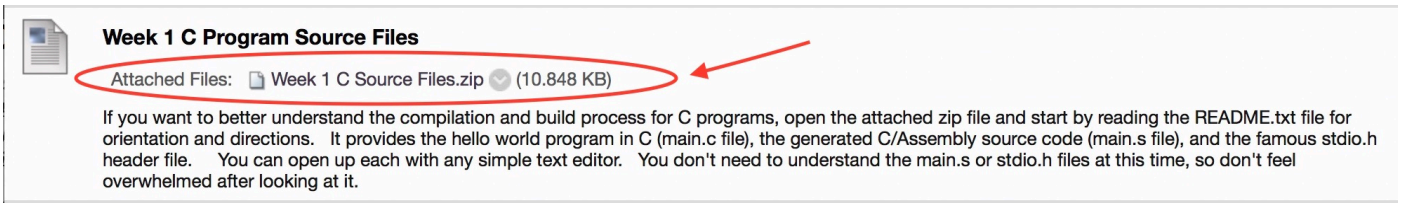
... we can link multiple object files together which will be useful if your program is made up of multiple files.

One final tip, you can open and look at any **C source file** (*.c, *.h, or *.cpp extension), you can look at any **assembly file** (usually a *.s extension), but I would not recommend tying to view or edit an **object file** (*.o) or an **executable file** (a.out, *.exe, etc.). Something will display, but you won't be able to make any use of it as a human, and

your computer screen might not be too happy trying to render it.

Week 1 C Program Source Files

We initially covered some basic C programs this week, such as the simple **Hello World C program** found in the file "main.c". We also covered the concept of **assembly language** and how it is created from your C code. You can find an assembly code translation of our Hello World program in a file called "**main.s**". Finally, you will also find a copy of the mysterious **stdio.h** file that we include in most of our initial programs to support library input/output functions. To access the files, download the folder stored in the zip file available back in our current week's notes. You won't be able to access the zip file from this PDF file. Instead, head back to your Week 1 lecture notes page for this week and you will find a BlackBoard Item container right near the end of the list of lecture notes near the Quiz area, it looks like this:



Just click on the attached zip file. Click on it and it should download a folder that you can place anywhere on your computer, or you can save the zip file directly to your computer. On some computers, you will get a variety of options by "right clicking" on a link. When you open the zip file, it will create a folder called "Week 1 C Source Files". The folder will contain the three files mentioned above, as well as this file which you should read first:

- **README.txt** - general information on each of the files included to give you an orientation of their purpose

Don't be too concerned about trying to understand the assembly language file that you find in the "**main.s**", as that is beyond the scope of this class. Likewise, you won't really understand everything you see in the **stdio.h** header file, but think of it as things to come later on in the class. At the very least, feel free to just download the zip file and take a look at all the files within that folder using your favorite text editor.

Summary

C has been around since the 1970's, yet remains one of the most widely used programming languages today. C is built for speed, it is one of the fastest languages in terms of execution. It is a great language to learn first as it has a nice mix of low and high level features that let you get as close to the hardware and internal operating systems as needed. This week we looked at the C programming language at a very high level. In the weeks that follow, we will concentrate on the syntax and features supported to give you a complete "hands-on" experience.

For Next Week

1. Get the **textbook** (new, used, ... any edition will work).
2. If using the latest version of the textbook (4th Edition), read **Chapters 1-2** to sync up with what we covered this week.

   Otherwise, if using an earlier version (such as the 3rd Edition), read **Chapters 1-3.**

3. Go to the **Start Here** area and read the **class syllabus**
4. Go to the **Start Here** area and **Introduce Yourself** ... it is worth 1 point of your grade.
5. **Get a C or C++ compiler** or get access to one ... *CFree* is very easy to use and set up. *Visual Studio Community* is another good free product to download, but use whatever one you wish. Overall, there are MANY options out there, use whatever works BEST for you. There is no need to purchase one, most are freely available.

   See the list of many compiler recommendations in the **Resources** folder on our class home page.

   If you wish, you could also use the **IDEOne (http://www.ideone.com)** compiler that runs in any web browser. It is simple and you'll get going right away. It will work for everything you need for this class.

6. Run the **hello world program** in your compiler that we covered in the lecture notes
7. Take the **Week 1 Quiz** and submit for grading
8. **Relax** and **get fired up** for next week!