Once upon a time...

Once upon a time, in the early 1970's, there was a guy named Dennis Ritchie, who worked at AT&T Bell Labs. He had nothing better to do but develop operating systems and computer languages. The guy was a genius. Anyway, one day (in the early 1970's) he brought forth a high-level computer language that we now know as "C". (It was originally called "B", but was changed to "C". Please don't ask me why, because I don't know.)

By the way, I plan to explain what a "high-level language" is in one of the next sections.

C did not become very popular until the late 1970's because C compilers were not available until then. (I also plan to explain what a "compiler" is, in one of the next sections.)

C programs originally ran on UNIX based systems. UNIX (an operating system) will also be discussed in one of the next sections, was also developed in the early 1970's at AT&T Bell Labs.

Lots of C Compilers were developed and were not completely alike. This became a problem because a program that was written (and compiled to machine code) on one system, would not work on another system. So, if I wrote a checkbook balancing program, for example, and developed in on my microcomputer (PC or Mac) at home, and then tried to run it on a Sun Workstation (for example) running UNIX; that very same checkbook balancing program would not work on the UNIX-based machine. This is not very good as far as program *portability* is concerned. It is important to be able to port (copy, move) programs from one machine to another.

Another analogy (getting away from computers) is if you asked a bar tender to make you a *gin and tonic* , and you got a drink with 5 ounces of tonic, and 1 ounce of gin. Then you went to a different bar tender and asked for another *gin and tonic* (expecting to get 5 ounces of tonic and 1 ounce of gin) and this time got a drink with 4 ounces of tonic and 2 ounces of gin. Well. Well. This is not very standardized, now is it? You'd be upset right? Right? RIGHT??? Yes. We expect certain standards to be followed in life. The same holds true in the computer world.

So, in order to rectify the problem of non-standard compilers, the American National Standards Institute (ANSI) stepped in and formed a committee in 1983 to standardize C compilers, so that people' s code would work on different computers. With an ANSI (pronounced ann-see) definition of C, you are assured that anyone who sells a true ANSI C Compiler has followed the standard. This is one of C's strengths. That it is a *highly portable* language. (More on this later.) So now, if you write a program on one computer (MS Windows based, for example), you can compile and then run it on another (VAX VMS, for example)! What joy!!! Could life get any better than this!?!?!?

It is because of its portability that C is widely used in industry. Another reason C is such a desirable language is because its instructions set (the C commands that we use to write our programs) allows the programmer to manipulate hardware directly (such as memory). This is similar to the *low-level* programming of assembly language; however, we're using a high-level language. Because of this capability, C offers an enormous amount of power and flexibility. (If this made no sense at all to you, don't worry. It will later. :-))

The C language resembles Pascal, in that it is a *highly structured* language. That is, both the code and the data are strongly organized (structured).

Examples of structured code include

- Sequence Structures (code runs in sequence)
- Repetition Structures (for, do, and while loops)
- Selection Structures (if, else, else if, switch)

Examples of structured data include:

- Arrays
- Structs (records)

The C language has some features that no other language has, making it more powerful than all other high-level languages. These features include:

- Pointers
- Operations on Bits.

Both of the above features are quite advanced, and are not covered in this *Intro to Programming with C - Part 1* course. They are covered in follow-up course: *Intro to Programming with C - Part 2.*

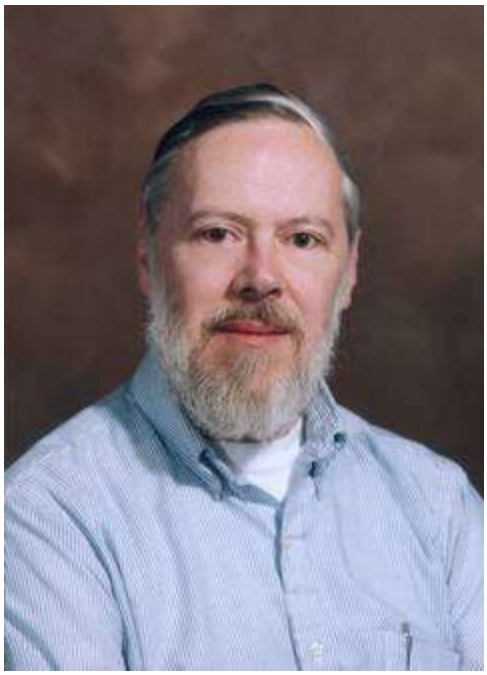
In this course, we will learn how to program in ANSI C. It is a good course for both the novice and experienced programmers. The textbook offers a lot of examples, some good, and some not-so-good. I will try to point out *typos* as much as possible, as this textbook contains quite a few of them. If you notice a typo, please feel free to let me know, in case I had missed it.

One last comment here. I strongly believe in the issue of *readability* of your software. That is, you can write a program that works, and works well, but if it is not *readable,* then you might as well have handed in a program that did not work (almost!).

Readability is just as important as (and I would argue sometimes *more* important than) correctness. If your code is not commented well, if it is not indented well, and if it does not follow other very straight-forward standards for readability (such as descriptive variable names), then your program will be very difficult to read, debug, and modify. This is useless code. 99% of all code that is written is eventually looked at again, by either the original programmer, or by someone else. Code needs to be updated for a variety of reasons (bugs, upgrades, etc.). In just a few short days, you can easily forget why you used a particular segment of code. So, it is so important to follow all of the readability standards when you code. I will stress this all along the way. This is very important to me, because it is very important in industry. If you are going to learn programming from me, I want to be sure that you are learning the right way to program.

This takes a bit more time and effort, but as you will soon see, it is time well spent!

Do you know who this is??



Dennis Ritchie!

Five Generations of Programming Languages

Machine, Assembly, High-Level, 4GL, and Natural Languages.

Programming languages have evolved in generations. Each new generation resulted in an improvement in the new languages that made it easier for people (programmers) to use them. The goal of a programming language is to communicate with the computer. The problem is that a computer, being an electro-mechanical device, can only understand a series of 2 state electrical signals (represented as: 1/0, high/low, on/off, true/false, 0volts/5volts, etc.). This section discusses how programmers are able to write code which a computer understands, without having to write it in its native "machine code".

This section of lecture notes discusses 5 generations of programming languages as listed below, and described in more detail in these lecture notes:

1. First generation 1945 Machine language (1's and 0's).
2. Second generation: 1950's Assembly languages (low-level instructions)
3. Third generation: 1960's High-level languages (COBOL, C, BASIC...)
4. Fourth generation: 1970's Very-high-level languages (NOMAD, FOCUS)
5. Fifth generation: 1980's Natural languages

First Generation: Machine Language

First generation languages were written in machine language. Machine language is a series of 0's and 1's. Although this is fine for computers, it is very difficult for programmers to comprehend. It wasn't too bad on very small computers, with a small set of machine instructions, but as computers grew in size and performance it became impractical to program using machine languages. Also, machine language is written for a particular machine, and a particular microprocessor, so machine code written for one computer cannot run on another computer. In other words, machine language is not portable. Although we no longer write programs in machine language, computers still require that a program be in machine language for the computer to execute it.

Below is some sample code written in Machine language for a particular computer. It will multiply 2 numbers together and store the product in a hardware register:

```
11110010  01110011  11010010  00010000  01110000  00101011
11100000  10010001  10000100  10000111  10010000  01010101
10100001  00001111  01010010  10010010  10000100  00100011
10010000  10001000  10000100  00001000  11001100  00100100
```

Second Generation: Assembly Language

Second generation languages are characterized by the use of symbolic instructions rather than 0's and 1's. The symbolic instructions are mnemonic (e.g., load, sum,),

which make them easier to learn and remember than machine language. Each instruction in assembly language corresponds to a machine operation Although assembly language gives programmers great control over the hardware, it is costly in terms of programmer time; it is also difficult to read, debug, and learn. Assembly language is used primarily today in some parts of system software. Like machine language, assembly language is designed for a specific machine and specific microprocessors, and is not portable.

Assembly language programs require a translator to convert the assembly language code into machine language, so the computer can execute it. These translators are known as **assemblers**.

Below illustrates some sample code written in assembly language. It too is written for a particular machine, and will multiply 2 numbers together and store the result in some hardware register.

```
PACK 210(8,13), 02B(4,7)
PACK 218(8,13), 02F(4,7)
MP      212(6,13), 21D(3,13)
SRP     213(5,13), 03E(0),5
UNPK 050(5,7),  214(4,13)
OI      054(7),X'FO'
```

Third Generation: High-Level Languages

Well, luckily for you, we will NOT be coding in any 1st or 2nd generation languages in this course. We will however, be coding in "C", which is considered a third generation, or high-level language. Third generation languages became more "English-like" and were much easier to write and debug than assembly language. Different third generation languages were written for different specialties. Some are better for business applications (COBOL), some for scientific and mathematical applications (FORTRAN, C), etc. Like assembly language, each third generation language requires a translator to convert the English-like, high-level code (**source code**) to machine code (**executable code**). Translators for third generation languages are called **compilers**. A few third generation languages use **interpreters** instead of compilers. A compiler converts every instruction in the program to machine language before any instruction is executed. Interpreters convert a single instruction, then executes it. This process is repeated for every instruction in a program. A program written in some high-level languages (such as "C"), can be compiled on one machine and executed on that machine. The source code can then be ported (copied) to a different computer, compiled on that different computer, and executed. High-level language code is considered to be portable across different hardware platforms. Some high-level languages include: FORTRAN (1954), COBOL (1960), BASIC (1965), Pascal (1970), C (1972), Ada (1979), C++ (1985).

Below is some sample code written in a variety of high-level languages. It represents the above code which is the multiplication of 2 numbers, and storing the product in some hardware location (in this case, in a variable in memory):

COBOL: MULTIPLY HOURS-WORKED BY PAY-RATE GIVING GROSS-PAY ROUNDED

C: gross_pay = hours_worked * pay_rate;

PASCAL: gross_pay := hours_worked * pay_rate;

Basic: g = h * p

As you can see above, the difference between one high-level language and another is something we call "Syntax". That is, how each instruction is represented so that the compiler can recognize it as a valid executable instruction.

Fourth Generation: Very-High-Level Languages

First, second and third generation languages are all classified as procedural languages. In a program written in a procedural language, not only must each instruction be coded correctly, each instruction must be in the proper sequence. Fourth generation languages (4GL) are non procedural. All you need to do is "tell" the program what you want it to do, and the language will create the program for you. Some categories of fourth-generation languages include:

- report generators (report write)
- query languages (SQL)
- application generators (*visual* whatevers)

I have limited experience with languages in this category. So, this is about all I have to

offer here. (Besides, "real" programmers code in 3rd generation languages!)

Fifth Generation: Natural Languages

Allows users to communicate with the computer using conversational commands that resemble human speech. Natural language development is part of the study of artificial intelligence. It allows commands to be framed in a more conversational way. (Huh?

Again, "real" programmers ...)

Hardware vs. Software, and more!

Before we begin discussing some fundamental computer terms, I want to take a step back (no, I didn't say I have to "stumble" back. I know what you're thinking, that I've had too many of those 4 oz, 2 oz gins). Well, no. What I really want to do is define some very basic concepts, to ensure that we are all on the same page.

First of all, I want to define the difference between a "Computer Programmer" and a "Computer User".

Today, lots of computer literate people know how to *use* a variety of computer programs at work and home (such as word processors, spreadsheets, graphics packages, and games). People who use these programs are considered computer *users*. People who *write* (or create) these programs are called *computer programmers*. Knowing how to use the computer is not the same as knowing how to communicate with the computer. An analogy: Just because a person is able to drive a car, does not make him/her an expert mechanic! We have to know how to communicate with the computer, in its own language, before we can become a computer programmer. (Hence, the reason you are all here taking this course!)

Next, I would like to define what a computer actually is. I know most of you already know (I saw a few of those eyeballs rolling), but again, I want to be sure that we are all on the same page!

There are many definitions of a computer. One definition is: A computer is an electronic device capable of performing computations at speeds millions/billions of times faster than humans can.

For example, a computer can perform millions of additions per second. It would takeÂ us humans decades, even using a calculator to do the same job!Â Every job that a computer is required to do is broken down into very simple instructions (add, subtract, test if zero, etc.) We sometimes measure the speed of a computer in MIPS -- which stands for "Millions of Instructions per Second". Soon we will be going to "BIPS" -- Yes. You guessed it. "Billions of Instructions per Second." Amazing, isn't it?! (My kids can't even complete a single instruction from me all day! No wonder I like computers so much!)

To solve a problem using a computer, we as programmers, must express the solution to the problem by using the instruction set for a particular computer language. The C programming language has a wonderful instruction set, consisting of hundreds of basic operations, as well as more complex functions (discussed later) available to the programmer to help solve the problem. When we write a program, the computer will do exactly what we have instructed it to do. Nothing more, nothing less.

So, basically, a **computer program** is actually just a **collection of the instructions necessary to solve a specific problem.**

A good practice before attempting to solve any problem is to first think about the problem and the solution. Then come up with a plan. A design. In computer programming, this design is called an "**Algorithm**". An Algorithm is **an approach or method for solving the problem**. Even if you simply jot down a few lines on paper, it is so important that you think about the solution before attempting to code it up! For example, if I gave you the following problem: Write a program which prompts the user for a number (input), and determines if the number is odd or even. Then display the results (output). A QUICK algorithm might look something like:

1.0 Prompt user for a number.

2.0 Determine if number is odd or even.

2.1 Divide number by two.
2.2 If remainder is zero then
number is even.
Otherwise
number is odd.

3.0 Display results.

Now, we could attempt to write some C code (using its instruction set) which follows the above logic. If you have any problems with your "logic", you will most likely see the problem in your algorithm. It is a lot easier to fix a "logic" problem during design, then once you have coded up your solution. Your algorithms can be as detailed or as high-level as you desire for this class, however in the "real-world" you may be asked to create algorithms using software tools, or you may be asked to write more detailed algorithms (called pseudo-code), or perhaps even flowcharts!!

In case you're wondering, for this class I do not require you to submit algorithms along with your programming assignment. I am going to assume that you are writing your algorithms before you code. Remember, taking the time to design will save you loads of time when you test your program. Without a good design, you code could be riddled with design flaws, giving you logic errors. It will be very difficult to fix the problem at this point, during your coding phase. It would have been better to find the logic flaw during

your design phase. (Have I stressed this enough yet?!?!?)

Now, we can finally, get to the real reason we are in this particular section of lecture notes: To define the difference between Software and Hardware.

Hardware is **the actual machine (system unit) and its supporting devices**. Examples of hardware items are listed below:

- Keyboard (I)
- Mouse (I)
- Printer (O)
- All-in-one Printer (I/O)
- Monitor (O)
- Touchscreen Monitor (I/O)
- CPU (P)
- Memory chips (S)
- Scanner (I)
- System bus (P)
- Disks (S)
- Microphone (I)
- Speakers (O)
- etc.

Legend:

(I) = **Input device**: Data is brought into the computer via this type of device.
(O) = **Output device**: Data is sent out from the computer via this type of device.
(I/O) = **Input and Output device**: Data can be both input to and output from this device.
(P) = **Processing device**: Data is processed using this type of device.
(S) = **Storage device**: Data is stored in this type of device.

Software is **the set of instructions that make the computer do something**.
Examples of software items include:

- Word Processors (MS-Word, Corel Wordperfect, Lotus AmiPro)
- Spreadsheets (MS-Excel, Lotus 123)
- Databases (MS-Access, Borland's dBASE)
- Games (Solitaire, Mind Sweep)
- Operating Systems (DOS, Windows, UNIX, Mac OS X)
- Compilers (cc, Borland's Turbo C++, MS-Visual C++)
- etc.

The basic rule of thumb: If you can touch (break) it, it is hardware!

System Software vs. Application Software.

Are you still with me?? Good. Grab another cup of coffee, and get comfy. (Some of this stuff can get pretty dry after a while.) It won't be long however, before we really get our feet wet with some real life programming!

Two Basic Software Types

Recall computer software is the set of instructions that makes the machine do something. These executable programs are also known as "computer programs" or "program files". All executable software that is written (by programmers) can be grouped into one of two categories: Application software and system software.

Application Software is the software written (by programmers) which allows users to perform specific tasks which generally produce some end result (i.e., information). Examples of application software include: Microsoft Word, Firefox, Solitaire, etc.

System Software is software that controls the computer and enables it to run application software. System software, **which includes the operating system**, allows the computer to manage all of its hardware, such as printer, monitor, keyboard, CPU, memory, etc. Examples of system software include DOS, Windows, Mac OSx, UNIX, etc.

Before you purchase a computer, you should have in mind which types of application software programs you want to run, as well as which system software you want to use. In the past, if you bought an IBM/compatible PC, you had no choice but to use DOS. Now you have more choices. Most users choose the latest-and-greatest version of Windows. Others choose LINUX (a version of UNIX). Others are loyal to DOS, and would "never upgrade to Windows in a million years!". (This quote came from a former student of mine.) Apple users do not have as much flexibility. They get whatever operating system software is currently being sold for that machine. At the time I entered these lecture notes, it was called "Mac OS 9".

So, you should be sure you know which operating system you want to use before choosing a microcomputer. You should take the time to try out the different operating systems, and different application software. Lots of people buy a certain microcomputer because their friends, family, or co-workers liked it. What is really important is what the user of the computer likes! Most microcomputers come with the system software included in the cost of the computer. Some include some application software as well. Software should play just as an important role as hardware when purchasing a microcomputer.

A few final words about System Software

There are 4 basic tasks of System Software:

1. It acts as a liaison between the user and the hardware.
2. It allows the user to manage files on disk (create, copy, rename, delete, move, modify etc.).

3. It provides the means for users to start up application programs.
4. It provides the user interface.

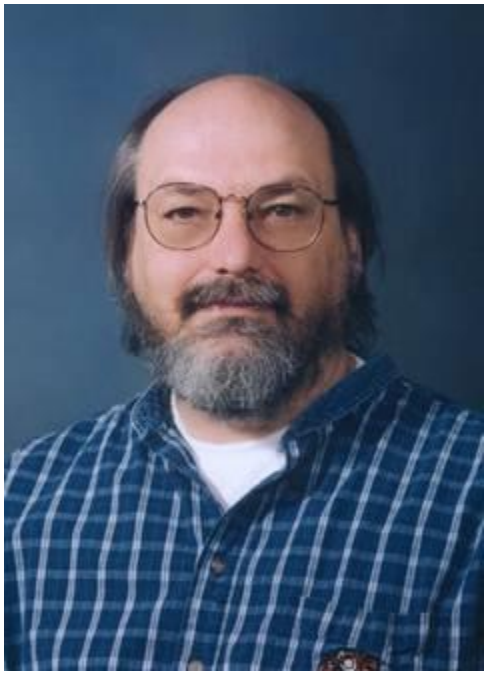
It is the system software that you are communicating with when you type a character on the keyboard. It interprets the keystroke, and issues the appropriate response. It is the system software that allows you to start up the application program(s) you want to run, and then communicates your commands to the hardware devices when you want to print, for instance. It is the system software that determines what your screen will look like when you power up, and how you will interact with the computer (user interface). It is the system software that provides you with the means for managing your files on disk.

This system software does not exist in one huge executable file. It consists of many executable files which, when used together, meet all of the necessary system software requirements. To get a feel for how "large" the system software is, take a look at the directory on your hard disk in which it is stored. (For Windows 7 users, there are about 116,233 files taking about 34.3 GB of storage!)

In this class, we will NOT be writing system software. All of our code will be general purpose, application software (at its most fundamental level).

FYI: Did you know that UNIX (one of the most powerful, and portable operating systems) is written in C?!?!

Do you know who this is?



Ken Thompson -- One of the original developers of the UNIX operating system.

Creating a program -- A 4-step process.

Before you can attempt to compile a program, you must first create it. This section discusses the 4-steps required to get a program up-and-running. Of course, this all

assumes that you are starting off with a good design!(*see section on algorithms*)

Step 1: Create (edit) the source code.

During this phase you will enter your source code. Depending on the development environment you are using, you must start up some sort of text-editor so you can enter your code. An editor is an application program, similar to a word-processor, except the information (code) is stored in text (ascii) format, with no additional graphics. With the Turbo environments, once you start the Turbo program, you are automatically put into a text processing window. With UNIX, you must start up a text editor yourself (such as vi, or emacs).

Once you have your text editor up-and-running, you can start to type in your code. Some text editors are specifically setup for you to enter C code (such as Turbo C and MS Visual C). These editors can be very helpful as you are typing in your code, as they give color-coded words and segments of code so you can more easily detect a "syntax error". (More on this later.)

Once you have entered all of you code, you must save your file on disk, giving it a name. It is a C standard to give your "source" file a descriptive name, and end it with a ".c" extension. That is, if you want to call your program "fun", you should name your source file: **fun.c**. (Your compiler may want to 'automatically' place **.cpp** at the end, please *save* your file *as* **fun.c** with a **.c** extension so that it will compile in standard ANSI C and not in C++, which is how .cpp programs compile)

Now, you have your "source code", in your source file (**fun.c**), you are ready to "compile" your program, which is step 2 of the process, and described next.

Step 2: Compile your source file.

Recall, the computer only understands a series of 1's and 0's. However, our source file contains "english-like" instructions in it. So, we must convert the source code into "binary", or machine code. Luckily, there is a software program written to do this for us, otherwise we would need another (huge) cup of coffee. (Or perhaps at this point we would grab the vodka! :-)) The program that is available to us to convert our high-level code to machine code is called the: ?????????? Yes. You guessed it. It is called the compiler! A **compiler** is simply an **application program which converts high-level code to a particular computer's machine code**. So, compilers are machine (computer) dependant. That is, each computer (with associated operating system running on it) has its own compiler. So, if you have a microcomputer running MS-Windows, then you will buy a compiler which runs on MS-Windows. If you have a

mainframe computer running UNIX, then you need a compiler which runs on UNIX. If you have a Mac computer, then you need to purchase a compiler that runs on a Mac.

How to start up your compiler will again vary depending on which compiler you are using! For example, with the Turbo C compiler and many others, you simply can click on the "COMPILE" menu. If it compiles with no errors it can then be run(executed). This is also usually just a menu pick - RUN (or Execute!). (You may need to create project space such as in MS Visual Studio)

In C-Free simply select the ***"Build"*** tab, and from the drop down menu select ***"Run".*** The compiler will compile, link, and execute the program. If compile errors are encountered there will be errors in red highlights below the code window. Your cannot always tell what your error is from the error listing. You will have to go back and correct the errors, and try again. That's it! You may read the rest of this section for what will happen in other typical compilers. Or jump down to "Step 4…"

After the compiler is invoked (started), and it will convert your source code, to machine code. (AKA - binary code, or object code.)

The compiler actually does the following:

1. Invokes (starts) the "preprocessor" (another program which checks a few things - more later).
2. Checks for errors. -- If any of the source code does not follow ANSI C Standards (remember those?), the compiler will alert you that there is an error. This type of error is known as a "**compiler error**" or "**syntax error**". At this point, you must go back to step 1 and fix any of the problems that you might have. Typically the compiler is pretty good at giving you a hint as to where your problems lie. After you make your changes, you must re-compile your source code to see if there are any additional syntax errors. Don't be surprised if you start off with only 3 syntax errors, fix those, and when you recompile, you end up with 10 different syntax errors! This is a normal process in the computer programming world!
3. Once the compiler no longer detects any syntax errors, it will then go ahead and convert your source code to object code (AKA: binary, or machine -- they all mean the same thing). Typically a file with a ".o" or ".obj" extension is created. So, in our example, after compiling file **fun.c**, you would end up with a file called fun.o (or **fun.obj**). This object code is not yet executable. That is, it is not yet in its final form for execution. Now the "linker" program must be called to put together (link) multiple files that make up your program. In most cases, you cannot put all of your source code into 1 file because normally, programs are very large, and your source file would be huge and unmanageable. So, we split the program into parts and each programmer gets a portion of the program to work on and will create his/her own source files. Each source file is compiled separately, and then "linked" when they are all complete. This brings us to the

next step in the process, which is to "link" all of the object files together. (We will briefly discuss "compiling larger programs" in a later chapter.) ***Warning!*** You may still have runtime problems with your code. Certain compilers are notorious for ignoring sections of code it does not understand. You must read the compiler *warnings* as well as the *error messages*.

4. **Step 3 - Link your program.**

Normally the linker is invoked (started) automatically by the compiler. Again, the linker, like the compiler (and the editor) is simply an application program which creates an executable file. You can choose whether or not to automatically invoke the linker after compilation.

Now that I've completely confused you...

Please do not get too stressed out about this particular phase. It will make more sense as we go along. What I want you to understand at this point is not necessarily HOW the linker is invoked, but WHAT happens when the linker is invoked. Once the linker is invoked, it checks if there are other program files that need to be combined with your program file to make a completed executable file. Once all of the "pieces of the puzzle" (so to speak) are located by the linker, the linker will then make another binary file called the "executable" file. This is the program file that is executed (started) so that you can see your results.

Typically, an executable file will be the same name as the source (and object) file, with an extension of ".exe". So, in our example, the linker would create the file: **fun.exe** (This is the default executable filename if you are using a Windows based compiler/linker like C-Free).

Like with the compiler, you may get errors during this link phase. The types of errors you might get are different than "compiler" or "syntax" errors that you get during compilation. The types of errors you might get during the link phase are normally "undefined" functions. In this case, you need to go back to Step 1 and fix your problem. We will discuss functions in more detail when we get to chapter 8. Another type of "**linker error**" is one that is caused as a result of not having enough hard-disk space for the linker to store the executable file. This should not be the case in our class, as our programs will be relatively small. However, think about the size of some of the MS-Windows based application programs, for example. MS-Word 2000 is 8.6 MB (megabytes)! That is a hefty sized program. So, as you can see, the linker may have problems trying to fit the executable file on a crowded hard-drive. Once the linker successfully creates an executable file, you can now go ahead and attempt to execute (run) it!

Step 4 - Execute (Run) the Program.

"HOW" you run your program depends once again on the operating system you are using. If you are using a Windows-based environment, you would double-click on the executable filename. However, if you are still within the C-Free environment, you can just click on the "Build" menu, select “Run”, and your program will run.

Once your figure out how to actually run the program, the executable file created by the linker is copied from hard-disk to **RAM** (Random Access Memory). The **CPU** (Central Processing Unit -- which executes your program 1 instruction at a time) wants to run the program from memory, instead of from the hard-drive because it is much faster if the program resides in RAM.

FYI: The hard drive is a mechanical device. Memory and the CPU are electronic devices (integrated circuit chips). So, if the CPU had to run the program from the hard-drive, it would be slowed down by the mechanics of the hard drive. If the program is instead copied into RAM, the CPU can run at its most efficient pace, since both the CPU and RAM are electronic devices.

If all goes well, (and it probably WON'T the first time you run your newly created program), the program should run to completion, presenting you with the desired results. However, as in the last 2 phases, things can go wrong during program execution. ***There are 3 types of errors that occur during run-time:***

1. **Fatal error** -- This occurs when your program "crashes" during runtime. You know this has happened when you see some sort of nasty message on your screen from the operating system such as: 7FFE 12A4 Core Dump. Don't panic. You simply go back to step 1 and try to figure out what went wrong. (Sometimes you may actually have to reboot your computer!)
2. **Infinite Loop** -- This occurs when your program "hangs" or gets stuck at a particular segment of code. You know this is happening when your program does not ever finish. This typically happens when you incorrectly use one of the repetition structures (loops). Sometimes your loop is generating output, and you will see lots and lots of text scrolling on your screen. Some of the time you will need to re-boot your computer and return to step 1.
3. **Logic error** -- This occurs when your program runs to completion, however your results are incorrect. For example, if you wrote a program that prompts the user for 2 numbers, and the program then calculates and displays the "sum" of the two numbers, and if you ran your program and the user typed 15 and 20, and your program displayed 55 for the sum, then you know you have a logic error. (Was that a run-on-sentence or what?!?!) These errors are sometimes difficult to find if the program logic is complex. This is the type of error you want to resolve during the "design" phase (hint, hint).

Now, throughout all of these steps, you will be making lots of mistakes. Not that I don't have faith in you, but I speak from my very own experience. Just know that the BEST thing you can do is make lots of mistakes. ***Yes!*** I have not lost my mind. The reason I say this is because when you make a mistake, you will troubleshoot your program and

find the solution to the problem. This process (known as "debugging" your code) is where you actually learn how to program. You will quickly become a good troubleshooter, easily fixing your own, and even other people's code. Because, once you make a mistake, you rarely forget how to fix that particular mistake. So, when you repeat that mistake, or if someone else has made the same mistake, you will know how

to fix it. Trust me.

In Summary...

4-Steps to creating programs:

1. Create (edit) your source code ---> fun.c
2. Compile your source code ---> fun.o
3. Link your object code ---> fun.exe (usually links automatically after compile)
4. Run your program (steps 2,3,4 happen as one single step in C-Free if there are no compiler errors)

Types of errors:

- Compilation (or Syntax) Error (.exe file is not created)
- Linker Error (.exe file is not created)
    - Undefined functions.
    - Insufficient disk space.
- Runtime Error (occurs when you run the program and it fails to execute/executes incorrectly)
    - Fatal error.
    - Infinite Loop.
    - Logic error.

Please also note any *warnings* that are generated - these may be indicative of possible errors you will encounter when you run the program. Also, always remember to *save* your program source. If you do get a run-time error, in some cases, you may have to re-boot your computer. If you haven't saved the source - it may be gone after re-booting.

What you will submit to me is the ***fun.c*** source file **only!** (**not** the *fun.exe* file, or any other files created by the system you use.) I will then compile and run the program on my system.

NEXT WEEK: -- Now that you (hopefully) understand some of the basic fundamental concepts of computers and programming, we will be pressing on and typing in our first program!

Programming is fun. And programming in C is even more fun!

The best way to learn the C Programming language is by EXAMPLE!

You will notice lots of examples throughout this course. In addition to ME providing YOU examples, YOU should attempt to edit, compile, link and run the sample code. This way you can get lots of practice with coding and debugging. This is what we call "hands-on" learning. It is the only way you will truly learn how to program.

PROGRAM 2.1
(almost)

Let's take a look at (dissect) this program in the textbook. Although it looks simple enough, it illustrates a lot of very important programming concepts of the C language.

The program looks as follows. I have included line numbers which are NOT part of the code. I have included them as to make it easier to explain each line of code in the program.

```
1.  #include <stdio.h>
2.  void main(void)
3.  {
4.      printf("Programming is fun.\n" );
5.      getchar( );
6.  }
```

The results (output) of this program, after you edit it (type it in), compile it, link it, and then run it, would be:

Programming is fun.

Note: With most compiler environments your output will be sent to some sort of 'output screen' (window), and you may need to figure out how to get back to that output screen to view your results. Your compiler should have information for you as to how to view the output of your programs (like pressing F5).

With C-Free you will always see the output screen, and you will also get: ***Press any key to continue . . .*** which is not typical of other compilers.

Now, let's take a closer look at the program above.

Line 1: **#include <stdio.h>**

You should include this statement at the beginning of every C program. It tells the compiler that we plan to use one or more of the general functions in the **Standard C Library**. We will discuss the Standard C Library in more detail in a later topic within this lesson. Also, we discuss functions in great detail during Week 9.

In this case, the predefined function that we are using in the above program is the **printf** function, which is the **output** function in C.

Actually, what happens is the compiler automatically initiates a separate program called the **preprocessor**. The preprocessor looks for a # character in the first column of each line. If found, the preprocessor executes the statement. Because of this, the # character **must** be in the **first column** in the text file. You can have many of these preprocessor commands, but each must be on its own line in your file, and must begin with a # in column 1. We do not discuss the preprocessor per-se in this course, however, if you would like more information regarding the preprocessor, please refer to Chapter 14 in the textbook.

So, as far as you are concerned, at this point, all you have to know is that the above statement (line 1 in the code) must be included in every one of your C programs.

Line 2: **void main(void)**

This statement **must** be included in every C program. It (**main**) tells the CPU (processor) where program execution begins. The first '**void**' could be '**int**' instead, as in the text book (more on this later...).

The parentheses following the word **main ( )** indicates that *main* is the start of a function. Regarding functions: at this point, you should just realize that **every C program contains one or more functions, one of which must be main**. This is a very important concept in C programming.

All of the code written in C must reside within one or more functions. Again, we discuss functions in more detail in week 9.

Also, the words: **void** are very much related to functions, and will be discussed in greater detail in week 9. Some compilers complain about the first word void, and this may have to be omitted. Again, more on that later.....

At this point you're probably saying to yourself: *This stuff is easy!.* As well you should be. Because, guess what??? It is easy!!!!

Lines 3 and 6: **{** and **}**

These are the **begin** and **end** characters. They indicate that the segment of code between them constitute a **block of code**. In this case, they define a **function block**. Every function which is defined (in this case, function **main**), must have **begin** and **end** indicators. This simple requirement helps to keep your code structured. (Note, in the Pascal programming language, another strongly structured language, you actually type in the words BEGIN and END instead of using the braces.) We use the **begin** and end characters for a variety of reasons in C. At this point, all you have to know is that when you define a function, you must surround the function's code within these lovely braces! :-)

Also, how can I forget, please remember to **indent** the code within a block (braces) by 2 to 10 spaces. This greatly improves the readability of your program. Once you pick a number of spaces to indent, please use that number throughout your entire program, for consistency.

Line 4: **printf(" Programming in C is fun.\n");**

Well, as you can imagine, this line is the heart of the program. Let's take this one step at a time. Although it appears somewhat complex, you will soon realize that this line of code, like all others so far, is quite logical. (At least I hope so.)

First of all the **printf** statement: **printf** is a function (a segment of code) that is provided to us when we purchase a C compiler. It is a part of the Standard C Library group of functions. It is an output function whose purpose is to send text to the output device (monitor/screen).

In this statement, function **main()** is actually making a **function call** to function **printf().** When we make a function call, the function name is used (**printf**). Following the function name, you need to enclose function information (a.k.a. function arguments or function parameters) between parenthesis. In this case, function **printf** expects to receive a string of text (a group of characters) as function information. Function **printf** will then send the string of text to the output device (monitor). For this reason, we enclose the information (string of text) within the parenthesis. Also, notice that string of text is enclosed within quotation marks. This also is a requirement of the **printf** function.

So, if you accidentally typed in the above line of code as:

printf (Programming is fun.\n);

Then you would get a ***compiler*** or ***syntax*** error during compilation because you forgot to surround the text in quotation marks! This is tedious, yes, but remember the compiler expects things a certain way, and you must follow the rules. You will quickly learn that all of these very tedious rules are actually a good thing, as it ensures that you as a programmer is coding exactly as you intend to code. (I try telling my kids that all of our rules at home are actually a good thing, but they don't buy it either! :-))

So, when the printf function is called with the above line of code (Line 4), it will output each and every character it finds between the quotation marks, EXACTLY as you type them in. It does this for all characters in the string, but will stop when it reaches the **** (backslash) character.

The **backslash** character is called the **escape character** in C. It is an indicator to the printf function to temporarily halt the outputting of standard characters and to output something else. Depending on what character follows the backslash, is what else will be output. In our example, the character **n** follows the backslash. The entire sequence **\n** is considered an **escape sequence.** In C, the **\n** escape sequence tells the printf statement to output a carriage return and line feed (a.k.a. the ***new-line*** character). So, when the printf statement reaches that particular portion of the string, it will simply send a new line character to the output device. This is why when you look at the output above, you see that the prompt is displayed on the line *following* the text. It is the **\n** that caused the new line to be output causing the prompt to be moved to the next line.

Look at Table 9.2 on page 215 in your textbook.

This table contains a list of valid escape characters in C. Any of these can be included in your printf statements to output a variety of non-standard characters. For example, if you want an audible alarm (bell) to be output with your text, you simply include a \a in your printf statement. For example:

printf (" \a Invalid data entered.\n Please try again.\n");

First, you would hear a ‘bell’, and at the same time you would see the following on your display:

Invalid data entered.

Please try again.

Notice that you can embed escape sequences within the text in your printf function call. They do not have to be just at the beginning, or at the end of the text string. (You may not really hear a 'bell' ring – that was for the old CRT terminals.)

Finally, notice that the call to function printf ends with a semicolon (**;**). All C statements, within a function block, must be terminated by a semicolon. If you leave the semicolon out, you will get a compiler error.

Line 5: **getchar();**

Some compilers will 'flash' the output screen so fast that the user cannot see it. If you suspect this is happening with your program, insert a **getchar();** statement into the 'end' of your program.

We have already covered some of the basic fundamentals of a C program. You are well on your way! If you did not understand any of the concepts discussed so far, please post any/all questions to the discussion board.

We will now make a slight modification to the above program to illustrate some additional concepts of C.

PROGRAM 2.2

This program is similar to the prior one, but it includes an additional line of code in function main(). The code looks as follows:

```
#include <stdio.h>
int main(void)
{
   printf(" Programming is fun.\n" );
   printf("And programming in C is even more fun.\n" );
   getchar();
   return 0;
 }
```

The statement of code: **return 0;** is needed when the main program is type 'int' as above. The ***return*** statement will be explained in more detail when we discuss functions in Chapter 9.

So, as you can imagine, the output of the above program would look like:

Programming is fun.

And programming in C is even more fun.

I want you to notice 2 things here:

1. Each statement in the function block of function main was indented, and each was terminated with a semicolon.

2. If you left out the '\n' at the end of the first statement, BOTH of the above lines would have been output as one continuous line. For example, if your 2 printf statements looked like:

```
printf (" Programming is fun." );
printf (" And programming in C is even more fun.\n" );
```

Your output would look like:

Programming is fun.And programming in C is even more fun.

So, as you can see, you must be very careful how you code. Remember: the computer will only do what you tell it to do. Nothing more. Nothing less. :-)

The next program illustrates how you can have more than 1 line of text output, with a single printf statement. I already provided you with an example of this concept when I discussed the "audible bell" above. But, it doesn't hurt to provide additional examples!

PROGRAM 2.3

You can take a look at this program and see how multiple lines of text can be output using a single printf statement. I will provide a different example here so to reinforce this concept. Let's simply modify the code in the previous program so that it outputs 2 lines of text using only a single printf statement. The code would look as follows:

```
#include <stdio.h>
main()
{
   printf ("Programming is fun.\nAnd programming in C is even more
fun.\n" );
   getchar( );
   return 0;
} /* end main */
```

As you can imagine, the output of the above program would look like:

Programming is fun.
And programming in C is even more fun.

Again, the reason this works is because you can embed *escape sequences* within a **printf** text string.

I added a comment on the last line ***(the end block marker)*** and you should always do that. Also, leaving a blank in front of main is the same as putting **int** there, so we still need the **return 0;** statement.

Please try to create, compile and run the above programs on your own computer. Vary the programs a bit and see what types of output you produce. This is an excellent activity for ensuring that the concepts discussed so far are completely understood. The concepts discussed throughout this course will remain far too abstract if you do not take the time to practice what you are learning.

Creating and Running a Program In C-Free:

1. Run C-Free

2. Select: File > New

3. Type in the above (or copy and paste it in.)

4. Select: Save as > name it something like fun.c (the .c extension is important!) Put it in a convenient folder.

5. Select: Build > Run (or just press F5)

If the program is good it will compile and run in a separate window. If not there will be error messages (that do not always tell you what the error is, but usually the line numbers are helpful)

That is it!

Also, throughout the textbook you will see some QUICK TIPS. Please take the time to read them as they provide some tried-and-true programming advice!

The next set of lecture notes introduces the very important concept of program **variables**. Perhaps you should go get a very strong cup of coffee at this time!

Variables in C - An introduction.

This section of notes will introduce the concept of program variables in C.

Variable: A location in memory where a value can be stored for use by a program.

We have all used "memory" before, with our calculators! It is the same idea with computer programs. The only difference is that with calculators, we have just one memory area to store our calculated value for later use.

So, we simply press the "memory +" button (or something like that) on our calculator and it stores the current value in memory. The value can later be retrieved by pressing the "memory" button on our calculators. With computer programs the idea is the same. That is, we have calculated values that we want stored while the program is running, so the program can make use of the values throughout program execution. The main difference between computer programs and calculators is that with computer programs, we may have *several* different calculated values that we want stored (and later retrieved).

In order for the program to keep track of values to be stored in memory, we (the programmers) assign a "name" to a memory location that will hold our calculated value. These named memory locations are known as "variables". We (programmers) do not need to know exactly *where* in memory these variables are being reserved, we simply have to know how to access them. We access values of variables by using the variable's *name*.

Not only do programmers need to specify a name for a variable, we also need to indicate the "*type*" of data the variable will contain. That is, we need to specify whether the variable (memory location) will contain numeric data, or non-numeric data (character or string data). If it will contain numeric data, we need to specify if the number will be a whole number (integer or "**int**") or if it will be a number with a decimal place (real or "**float**"). C is a "strongly typed" language, like Pascal (unlike Basic or FORTRAN). Although data typing can seem quite tedious, it ensures program and data integrity. In other words: it's good for you!!! ☺

Note: The value of variables can change as the program executes (hence the word "variable"), however, the name and data type of the variable cannot change.

I think it is a good time to introduce an example to help explain these concepts. The next program in the textbook will be used as an example and is shown below.

PROGRAM 2.4

```
1.  #include <stdio.h>
2.  void main(void)
3.  {
4.      int sum;
5.
6.      sum = 50 + 25;
7.      printf ("The sum of 50 and 25 is %i\n", sum);
8.      getchar( );
9.  }
```

When the above program is compiled, linked, and then executed, the following is output:

The sum of 50 and 25 is 75

Again, let's *parse* the C code above, so we thoroughly understand the new concepts introduced here.

Lines 1, 2, 3, 8, and 9: Nothing new.

These 5 lines were discussed in previous notes. Refer to previous notes for more details if needed.

Line 4: **int sum;**

This statement indicates a very important concept in C. That is, that you must **declare** any variable that you want to use in your program.

Declaring a variable: It tells the compiler the **name** and the **data type** of the variable that you want to create. (We will find out why data typing is so important in Week 3.) Variables must be declared before they are used. Variable declarations typically appear in the beginning of the function block, following the begin brace.

WARNING: In ANSI C all declarations of variables must precede all other programming statements.

This would be wrong:

```
#include <stdio.h>
  void main(void)
  {
      printf("Welcome to my program\n\n\n");
      int sum;
      sum = 50 + 25;
      printf ("The sum of 50 and 25 is %i\n", sum);
```

```
    getchar( );
}
```

In the correct above, variable **sum** is declared to be of type **int**. That is, a location in memory will be reserved to hold an integer (whole number) value while the program executes. The programmer can store information in the memory location known as **sum**, and can retrieve information from that memory location, as long as it is of type **int** (or a whole number). We will discuss the different data types in great detail in Week 3.

During declaration, a location in memory (variable) is reserved to hold whatever the program wants to eventually store in it. There is no "initial value" automatically assigned to the variable. It is not automatically initialized to zero, or any other value. If you want the variable to contain valid data, you must store that data into the variable. We store data in a variable by "assigning" the variable a value. (Shown below in Line 6.)

Also notice that during declaration, the variable's type is listed first, then the variable's name, followed by a semicolon. Variable names should be descriptive. There are several variable naming rules and conventions, which are discussed below. One final note here, a program can have many variables declared. This is also illustrated during Week 3.

Line 5: (this is a blank line)

I know what you're thinking. You think that I've had too much coffee. Well, for your information, I put a blank line in the program on purpose. :-) Blank lines are very important in programs; not for what they do (they actually do nothing), but for how they look. Blank lines help to make your program more "readable". There's that "R" word again! We typically will put a blank line following our variable declarations so to keep them visually separate from the processing code.

Line 6: sum = 50 + 25;

This line is the heart of this program. It illustrates the concept of "assigning values to variables". This particular program adds 2 numbers (notice, both integers) 50 and 25, and stores the result in variable **sum**. In C, we use the **assignment operator** "**=**" for storing results into variables. The statement is actually executed (by the CPU) from the right side of the assignment operator to the left. So, 50 and 25 are added, and the result of that calculation is then assigned to (stored in) the memory location that we have designated as variable **sum**.

At this point, the program is able to access that calculated value simply be referencing the variable's name (**sum**).

One point to be made here, and will be made again throughout this course, and that is that the data type of the information on the right side of an assignment operator should be the same as the data type of the variable on the left side of an assignment operator. This ensures portability of code. In our example, I pointed out that both of the values on the right side of the assignment operator are whole numbers (**int**s), and the variable on the left side (variable **sum**) is also of type **int**. We will discuss this in more detail during Week 3.

Line 7: printf ("The sum of 50 and 25 is %i\n", sum);

Okay, here's where it gets a bit tricky. (Just a bit though. I know you can handle it!)
We have visited the **printf** statement before, but we now see 2 different components. First of all, there is a very odd **"%i**" before the \n. Also, the string ends at the end quote, but then there is a comma and a variable name (sum). Hmmmm. Are they related you ask. The answer is YES!!! Let me explain how.

First of all let's discuss the **%i**. The **printf** function will output everything it finds between the quotation marks until it reaches a backslash, in which case it processes the escape sequence (discussed earlier). Another character that will temporarily halt the output of a **printf** statement is the percent sign. The **percent sign** tells the printf statement that a "**value**" is to be displayed at that particular spot in the output. The "**i**" after the percent sign tells **printf** that the **type of data** to be displayed is an "integer". The combination of the percent sign and the letter "i" (**%i**) is known as a format specifier. It specifies the format (type) of the data to be displayed at that point in the output string.

So, we have a format specifier of **int**, and **printf** is now ready to output some value of type "**int**", but *where* does **printf** find that particular value. Well, it looks at the parameter (argument) following the end quotes of the string being displayed. (When you have more than one argument being passed to a function, they are listed one after the other, separated by commas. - More on this later.) So, the **printf** function looks for a comma and then finds the name of the integer variable where the value to be output is located (variable **sum**, in this example).

At this point the value stored in variable **sum** is retrieved by the **printf** function and output in the exact spot that the format specifier (**%i**) is located in the output string. Then **printf** continues outputting any additional characters that appear in the output string. In our example above, only the "newline character" (**\n**) is output following the value of variable **sum**.

So, to refresh your memory, the **printf** statement looked like:

printf("The sum of 50 and 25 is %i\n", sum);

And the output of this **printf** statement looks like:

The sum of 50 and 25 is 75

If I wanted an exclamation point to be output at the end of the statement above (because, this stuff is so exciting!), then I would have to tell **printf** just that. The **printf** statement would now look like:

printf("The sum of 50 and 25 is %i!\n", sum);
^ notice the exclamation point here

And the output of the **printf** statement now looks like:

The sum of 50 and 25 is 75!
^notice the exclamation point in output.

So, you can see how very tedious, yet comforting all of this is, knowing that the computer will do what you tell it to do. Nothing more, nothing less. (Is it my imagination, or have we heard this before?!?!) ☺

Now, let's ensure our knowledge of variables by reviewing one more example. The following program builds on the previous program, using more than one variable in the program.

PROGRAM 2.5

```
#include <stdio.h>
void main(void)
{
     int value1, value2, sum;

    value1 = 50;
    value2 = 25;

    sum = value1 + value2;
```

```
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);
    getchar( );
}
```

When the above program is compiled, linked, and then executed, the following is output:

The sum of 50 and 25 is 75

Notice, the output is the same as the previous program. But, there are some very obvious differences in the source code.

First of all, 3 integer variables are being declared in this program: **value1**, **value2**, and **sum**. Note that you can declare each variable on its own "line" in the program as follows:

```
int  value1;
int  value2;
int  sum;
```

instead of: **int value1, value2, sum;**

This is a programmer's personal preference. Either method is acceptable.

Here 3 memory locations are being reserved to store 3 separate integer values.

Now, in the following line:

value1 = 50;

we *assign* the integer value **50** to integer variable **value1**.

With the line: **value2 = 25;** we *assign* integer value **25** to integer variable **value2**.

Finally, with **sum = value1 + value2**; we *add* the value of variable **value1** (50) to the value of variable **value2** (25) and *assign* the results of that addition to variable **sum** (75).

Now, the **printf** statement. Don't be afraid of it. Its bark is bigger than its byte! ☺

The **printf** looks as follows:

printf("The sum of %i and %i is %i\n", value1, value2, sum);

If you take this one step at a time, you see how it follows all of the concepts discussed so far. That is, the **printf** statement will continue to output characters to the display until it comes across a percent sign or a backslash. In this case here, it will reach a percent sign first (**%i**). Printf will temporarily halt its output and "look" for an argument following the output string (hopefully containing the name of an integer variable).

Notice the first argument in this **printf** statement following the output string is in fact variable **value1**. So, the value of variable **value1** will be displayed at the spot that the *first* **%i** is located in the output string.

Next, **printf** will continuing outputting characters until it reaches the *second* **%i**. It will look for a second argument following the text string, and finds variable **value2**. It grabs the value of variable **value2** and outputs that at the location of the second %i.

Finally, **printf** continues outputting characters and finds the *third* **%i**. In this spot, **printf** will find variable **sum** as the next value to be displayed. Then, the **\n** (newline) is output. Whew!

The last program shown is my preferred way of doing the problem*. The shortest way is not necessarily the best right now:*

NOT Desirable:

```
#include <stdio.h>
void main(void)
{
        printf ("The sum of %i and %i is %i\n", 50, 25, 50+25);

        getchar( );
}
```

Although the result is the same, and the program is correct, this kind of shortcut undermines the purpose of the exercise - which is to learn to store data in variables, compute with the assignment statement, and print the data contained in variable locations. *I personally do not like computations within **printf** statements, even though allowed in C.* **Please do not submit program #1 in this manner.**

RULES FOR FORMING VARIABLE NAMES

- Variable names in C must begin with either a letter or an underscore.

- Variable names in C may contain letters, underscores or digits (0-9).
- Variable names in C should not exceed 31 characters.

Well, those are the rules. However, one can truly run-a-muck (?) with only the 3 rules listed above. Keep in mind, C is case-sensitive, so variable name "sum" would be different than "Sum". That is, if in your program you declared variable: **int sum;** and then later in your program you made the following assignment: **Sum = 50 + 25;** you would get a compiler error telling you that variable "Sum" is not declared. And you may get an additional another compiler warning telling you that variable "sum" is not used!

So, with that in mind, we have a few programming conventions (standards) that we use when forming variable names. They help maintain the readability as well as the correctness of the program.

STANDARDS FOR FORMING VARIABLE NAMES

- Use only lower-case letters in variable names, along with digits and underscores if desired.
- Do not begin your variable name with an underscore. Begin with a character.
- If you have multi-word variable names, do not group it all together as a single word.

Example: If you want a variable to contain the "grand total" of all numbers, **do not** call your variable: **grandtotal**. Instead, use one of the following 2 standards:

You could name your variable: "**grand_total**" or "**grandTotal**" . In the first case (grand_total), you put an underscore between the 2 words. In the second case (grandTotal), you make all subsequent words with a capital first letter, and skip the underscore.

Whichever standard you choose, be sure to use it throughout your program. Do not mix different standards here, because the inconsistency could reduce the readability.

The textbook does a good job explaining these concepts. Please refer to the programming examples in Chapter 2 of your textbook for additional information and/or clarity.

The next topic in this chapter relates to **"Commenting Your Code".** It is a very good programming habit to get into from the start. (And essential to maximize the points you get on your programs!) ☺

Hi-Ho. Hi-Ho. Let's document our code! We code all day and get good pay. Hi-Ho. Hi-Ho.

Maybe it's time for me to take a break. What do you think?!

The sole purpose of documenting your code (adding comments) is to improve the readability of your program. Comments are ignored by the compiler, and do not cause the computer to perform any action when the program is executed. The compiler actually "tosses out" the comments when the object code is generated, as they provide no use to the program itself.

So, what's the big deal? Why even take the time to bother with information that is ignored by the compiler? The answer is ***because we are not computers.*** We are humans. Although C is considered a high-level language, it can still be somewhat cryptic. It is nice if programmers add some "human" language to the program, so we as humans can easily understand it.

You must realize that code that is written is not just written once and never looked at again. Code is usually reviewed by peers(or bosses! Or ***ME!),*** it is sometimes updated because of problems or enhancements. Even if your program works perfectly, lots of times your customer will want some sort of upgrade or change. For all of these reasons, code will need to be modified. Now, I know what you're thinking. (Yes, I am psychic ☺) Your thinking that if you write code, you will have no trouble updating it in the future, even if it is not documented well. Well. Well. Well. Are you in for a shock! Even veteran programmers, who after just a few days being away from their code, will look at the very code that they just coded a few days previous and say to themselves: "Now what the heck, pmf (pardon my French) is this code doing?!?!" (Sometimes the word "heck" is substituted with other, more harsh words, that I cannot use in this course.) And, keep in mind another possibility. You may not be the programmer assigned to update the code that you wrote! This is very common. So, when you code, realize that other programmers will be working with your code in the future. How unsettling! This is the real world folks!

So, to ensure that your code is easily readable, understandable, modifiable and maintainable, you must document your code. Now instead of wondering WHY you should document your code, I hope that you are all eagerly asking yourself HOW do I document my code. That's the spirit!

Documenting your code is actually quite easy, if it is done from the start. That is, as soon as you type in your code, add in your comments. There are 2 types of comments:

1. Block Comments.
2. In-line Comments.

BLOCK COMMENTS

Block comments are comments that you use to describe a segment of code that follows the comment. It typically consists of several lines of information. You start a block comment with the following symbol **/*** (slash and asterisk) and end a block comment with the following symbol */ (asterisk and slash). Some good programming standards for "where to put block comments" include the following:

- At the beginning of a function, sort of as a function header (description).
- Before the "variable declaration" section.
- Before a "loop" statement (discussed in Weeks 5 and 6).
- Before an "if" statement (discussed in Weeks 8 and 9).
- Before any calculations are to be performed.
- Before any user input (prompts - discussed in Week 4).
- Before any program output.
- To the right of an "}" (end brace) to describe what block of code the end brace is ending. (This will be a -2 deduction if you do not!)

The last one is the one most often left out. The next program in the textbook illustrates a program with some block comments. It is actually the program we just reviewed in the prior set of notes, with some comments added. This particular example shows mainly single-line block comments. However, block comments can span several lines. I will type the example here and modify the comments slightly to illustrate how block comments can span several lines if necessary.

PROGRAM 2.6

Slightly modified - using "Block Comments"

```
/* This program adds two integer values
   and displays the results to the user. */

#include <stdio.h>
void main(void)
{

    /* Declare variables. */

    int  value1, value2, sum;

    /* Assign initial values to variables and
       computer the sum. */
```

```
    value1 = 50;
    value2 = 25;
    sum = value1 + value2;

    /* Display all values and the
       calculated sum. */

    printf ("The sum of %i and %i is %i\n", value1, value2, sum);

    getchar( );

} /* end main */
```

Naturally, I have some comments regarding block comments. The following list provides some additional programming standards:

1. Indent comments at the same level as the code that follows it. In the above code, the comment above function main starts in column 1, just like function main does. The comments within the program are indented the same number of spaces as the code in the function main's block. As of now, this is not too big of an issue, however it will become a more important issue once we discuss **loop** and **if** statements. At that time, I will provide an example to you regarding proper indentation of block comments.

2. Skip a line before and after a block comment. This provides a "roominess" in your program which adds to the readability of the code. You don't want your code to be too "squished", otherwise it is more difficult to read.

3. Do not make your block comments very "computerese". That is, they should not contain any language which appears to be code.

For example, a **good** comment would be: **/* Display the calculated sum. */**

a **not-so-good** comment would be: **/* printf the output. */**

4. Be sure your comments are descriptive, and not vague. I don't know how many programs I've received from students that contained the following 3 block comments in them:

/* Prompt user for information */

/* Perform calculations */

/* Output results. */

These appear to be okay, however they really are quite vague. Particularly when you realize that almost EVERY C program prompts for information, performs some calculations, and displays some results. Your block comments should be specific to your program.

In the "Programming assignment point deductions" section of the "How You'll Be Graded topic in the "Start Here" lesson, I have listed the points that will be deducted from your programming assignments if you do not follow the commenting rules.

IN-LINE COMMENTS*

In-line comments are typically used when you have just a few words to say on a single line. They do not span multiple lines. You start an in-line comment with the following symbol: ***//*** (two slashes). Since in-line comments can only span a single line, there is no "end" symbol.

*In-line comments are not strictly ANSI (at least the old ANSI) so please use only block comments in your programs, I have adopted a -1 per // comment in the point deductions to discourage their usage.

I used to allow in-line comments, so some of my examples may still have them in there, so be careful when you cut & paste one of those programs it will not compile cleanly in all compilers. The book uses them too!!

Data Types in C

Recall, there are 3 basic data types: **Integer**, **Floating Point** and **Character**.

INTEGER data types:

- **int**
- **long int**
- **short int**
- **unsigned int**
- **unsigned long int**
- **unsigned short int**

The "int" data type:

Variables that are declared type "**int**" (or integer) are those that will contain *whole numbers*. That is, numbers without decimal places. To declare a variable to be of type integer, you simply use the abbreviation "**int**" before the variable name.

For example:

```
void main(void)
{
    int  value1;
    int  total = 0;
```

In the above code segment, 2 variables are declared to be of type **int**: namely **value1** and **total**. Notice that not only is variable **total** declared to be of type integer, it is also being assigned the value zero. This is a valid C statement, and is called "Initialization during declaration". Sometimes you want your variable to contain an initial value before it is used (such as keeping a running total), so we have the ability to initialize during declaration.

When outputting (printing to screen) the value of an integer, you use the "*format specifier*" **%i** in your **printf** statement. You can also use an "older" format specifier for integers known as **%d**. There is a slight difference between the two, however, in all of the programs for this class, and in most "real world" programs, you can interchange the 2 without any problem. We will discuss these differences later in the course.

So, for example, (as seen in a previous programming example), the following segment of code will declare, assign, and output a variable of type **int:**

```
Void main(void)
{
   int  num;
```

```
    num = 2000;
    printf ("The number is %i. Goodbye!\n", num);
    getchar( );
}
```

The output of the above program segment would be:

The number is 2000. Goodbye!

Now, to spice thing up a bit. Most machines (computers) will reserve 2-bytes (16-bits) of memory for each integer variable. If this is the case, the range of numbers that your computer can store in 16-bits is: -32767 to 32767. If you had a program that needed to work with numbers larger than that (say, in the millions), then declaring your variable to be of type **int** to hold this very large number would not work. You need a "bigger" integer. So, C provides a variation of the **int** data type called: ***long***.

The "long int" data type:

Values that are declared type "**long int**" or simply "**long**" are those that will contain *large whole numbers*.

For example:
```
long  grand_total;
grand_total = 1000000;
printf ("The Grand Total is: %li\n", grand_total);
```

In the above examples, variable **grand_total** is declared to be of type **long integer**, and is assigned the value of 1 million. Also, when outputting the value of a long integer, you use the "*format specifier*" **%li** (or **%ld**) in your **printf** statement.

The output of the above segment of code is:

The Grand Total is: 1000000

Most machines will reserve 4-bytes (32-bits) of memory for each long integer variable. If this is the case, the range of numbers that your computer can store in 32-bits is: -2,147,483,647 to 2,147,483,647. (BTW - These "sizes" are defined on page 433 of your textbook.) If you had a program that needed to work with numbers even larger than that (say, 3 billion), then declaring your variable to be of type **long** to hold this very large number would not work. You need an even "bigger" integer. So, indeed, C provides a way to handle this dilemma. We will discuss it below in the section titled: **unsigned long**. But first, I want to backtrack a bit and mention what we do if we know that a value that we plan to use will be very "small". That it will probably not exceed 100. Well, why reserve 4 bytes (**long)** or even 2 bytes (**int**)of memory for that variable, when a single byte will do. C provides a data type which is *half* the size of an **int**, and that is type ***short***.

The "short int" data type:

Values that are declared type "**short int**" or simply **"short"** are those that will contain *very small whole numbers*.

For example:
```
short  age;
age = 25;
printf ("You are %hi years old.\n", age);
```

In the above examples, variable **age** is declared to be of type short integer, and is assigned the value of 25. Also, when outputting the value of a short integer, you use the "*format specifier*" **%hi** (or **%hd**) in your **printf** statement.

The output of the above segment of code is:

You are 25 years old.

Some machines will reserve 1-byte (8-bits) of memory for each short integer variable. If this is the case, the range of numbers that your computer can store in 8-bits of memory is: -127 to 127. Some machines do not like to work with "odd" memory addresses (a single byte at a time), so will allocate 2-bytes even for a **short** integer variable. (This is transparent to the program, as you will see when we present some additional examples.) In this case, the size of data type **short int** is the same as that of a regular **int**: 2-bytes of memory per variable.

The "unsigned int" data type:

Values that are declared type "**unsigned int**" are those that will contain *positive whole numbers*. That is, numbers without decimal places AND numbers that are *greater than or equal to zero*. You use this data type if you know your values will always be positive, and you need to work with numbers slightly higher than 32,767. With a regular **int**, you get 16-bits to work with, but only 15 are usable for your number. The last bit is used as the "sign-bit". If zero, your number is positive, if 1 your number is negative. (Programmers do not typically concern themselves with how data is represented in memory, but I think you should know what is going on.) When you declare a variable to be of type unsigned int, you are freeing-up that 16th bit for data. So, your range of valid values for this type of variable is: **0 - 65,535**.

To declare a variable to be of type unsigned integer, you use "**unsigned int**" before the variable name.

For example:
```
unsigned int   value1;
```

When outputting the value of an unsigned integer, you simply use the "*format specifier*" **%u** in your **printf** statement.

As you guessed it, we also have an ***unsigned short integer*** and and ***unsigned long integer***. At this time, I will simply summarize (in table form) below the various integer data types, their format specifiers for the printf statement, and their range of values. I think you get the gist of this by now.

Refer to chapter 3 for additional details (Table 3.1 on page 28).

Data type	Format Specifier	Range of values
short int	%hi (or %hd)	-127 to 127
int	%i (or %d)	-32767 to 32767
long	%li (or %ld)	-2,147,483,647 to 2,147,483,647
unsigned short int	%hu	0 to 255
unsigned int	%u	0 to 65,535
unsigned long int	%lu	0 to 4,294,967,295

Now we move on to "Floating Point" Data Types. Luckily, for your sake, there are only 3 types of floating point data variables.

FLOATING POINT data types

- **float**
- **double**
- **long double (???)**

The "float" data type:

Floating point data is data that is numeric, but contains a *decimal place*. Another term for this type of data is: **real**. Some examples of floating point data include: 3.3, 3.0, 271.67452, -.001, etc.

To declare a variable to be of type floating point, you simply use the abbreviation "**float**" before the variable name.
For example:

```
void main(void)
{
    float  value;
    float  sum = 0.0;
```

In the above code segment, 2 variables are declared to be of type **float**: namely **value** and **sum**. Notice once again, that I initialized variable sum to be equal to 0.0 during declaration. An important point here is that I used 0.0 instead of 0

because my goal, to ensure that my C program is 100% portable, is that the data type of information on the right side of the assignment operator (**=**), is the same as the data type on the left side of the assignment operator. And, since variable **sum** is of type **float**, the value zero should also be of type float, hence **0.0**.

For data of type float, C will maintain 7 decimal places of accuracy for you. That is, if you set variable value above equal to: 10.123456789, C would only be able to store: 10.1234567. The remaining 2 decimal places would be inaccurate.

When outputting (printing to screen) the value of a floating point number, you use the *"format specifier*" **%f** in your **printf** statement.
So, for example, the following segment of code will declare, assign, and output a variable of type **float:**

```
void main(void)
{
   float  num;
   num = 25.26;
   printf ("The number is %f. Goodbye!\n", num);
   getchar( );
}
```

The output of the above program segment would be:

The number is 25.260000. Goodbye!

Controlling the number of decimal places displayed.

Why am I showing 6 decimal places? It has nothing to do with the "accuracy" that I mentioned above. What is going on here is that C will *assume* that you want to see 6 decimal places in all of you floating point output, when you use %f. This is not always the case. So, I need to inform C that I only want to see 2 (for example) decimal places in my output. I do this by putting a "**.2**" (point 2) between the **%** and the **f** in the format specifier. For example, to display the above number with only 2 decimal places, my **printf** would look like:

```
   printf ("The number is %.2f. Goodbye!\n", num);
                              ^ Notice the .2f here
```

Now, I would see as my output**:**

The number is 25.26. Goodbye!

What if I wanted to see only **1** decimal place using the above variable (**num**). My **printf** statement would look like:

printf ("The number is %.1f. Goodbye!\n", num);
^ Notice the .1f here

And my output would be: (pay attention here -- it's not what you think...)

The number is 25.3. Goodbye!

Notice, the output was not 25.2, but 25.3. That is because C will *round-up* if the number of places you are outputting are less than the accuracy that C has maintained for you, and if the next decimal number is 5 or higher. So, in this case, since variable num is of type float, C automatically maintains 7 decimal places of accuracy. The number is stored as: 25.2600000 (7 decimal places of accuracy), however you are only asking for 1 decimal place. C will round up for you here, because the second decimal place (the number 6) is greater than 5.

One last thing here, recall, C will maintain up to 7 decimal places of accuracy with a float variable. If you opted to output *more than* 7 decimal places of your variable of type float, you will get data beyond the 7th decimal place displayed, it would just be bogus. (That's an official computer term by the way - bogus.)

For example:
```
float  results;
results = -345.123456789;
printf ("The results are: %.12f\n", results);
```

Your output would be:

The results are: -345.123456720699

Notice the accuracy was maintained up through the 7th decimal place, and the last 5 decimal places output in this example are, well, as I put it earlier... bogus!

Now, let's say you need more than just 7 decimal places of accuracy. C provides another floating point data type which will give about 16 decimal places of accuracy. It is type **double** described below. I know what you're thinking (did I tell you I was psychic?!?!), you're thinking that "**double** is to **float**" as "**long** is to **int**". Oh... you weren't thinking that? You were thinking about a bagel and cream cheese? Oh well, my psychic powers aren't too precise. I'm still working on them. ☺

For the 2 or 3 of you who *did* notice the similarities between **double** and **long**, you are right on track! Good for you.

The "double" data type:

Values that are declared type "**double**" contain *floating point numbers with lots of accuracy*.

The format specifier for variables declared as type double is: **%f**. Yes. That's right. %f. It is *the same format specifier as for variables declared as type float*. Although C will maintain 16 decimal places of accuracy for variables of type float, if you use **%f** as your format specifier in your printf statement, C will once again display 6 decimal places (by default).

For example:
```
double  results;
results = -345.123456789;
printf ("The results are: %f\n", results);
printf ("The results are: %.2f\n", results);
printf ("The results are: %.12f\n", results);
```

The output would be:

```
The results are: -345.123457
The results are: -345.12
The results are: -345.123456789000
```

Notice on the first line of output, the 6 decimal places output were: .123457, not .123456. The reason is because you are asking C to output less precision than C has maintained for you, so C will round-up if needed. Since the number 7 (following the number 6) is greater than 5 then the number 6 is rounded up to 7, in this case.

The "long double" data type:

Values that are declared type "**double**" contain *floating point numbers with lots and lots of accuracy (at least you would hope so)*.

The textbook mentions a data type called: **long double**. It is supposed to provide more precision than the double, but this is typically not the case. Most machines cannot provide more than 16 decimal places of accuracy, so the long double qualifier in front of the variable name is typically ignored by the computer, and simply type double is used. If you want to give it a try, the format specifier for long double is: **%lf** (or **%Lf**).

A tidbit about Scientific Notation:

One last comment regarding floating point numbers is that you can represent (output) floating point numbers using "*Scientific Notation*". If you use the **%e** or **%E** format specifier in the **printf** statement on a variable of type floating point (or double), C will output your results using scientific notation. Please read page 27 for more information regarding scientific notation.

Next, we try something completely different. A non-numeric data type called **char**.

CHARACTER data type

- **Char**

Not all C programs are numeric-only programs. More often than not, you will have to represent data that is non-numeric in your program. When we think of non-numeric data, we often think of "strings" of data. Such as someone's name. However, C does not provide a "**string**" data type. Instead, C provides a more basic "character" data type (called **char**), where within a variable of type **char**, you can store a single character. If you want to work with "string data" (i.e., more than a single character in the data), then you must group individual characters together to form your string. This is done using "arrays" which is covered in Week 7 of this course. For now, we will discuss the most basic non-numeric data type called: **char**.

The "char" data type:

Values that are declared type "**char**" contain *a single character.* The *range of values* of data of type **char** is: *any character on your keyboard* (and then some)! The *format specifier* for data of type **char** is **"%c**". To assign a character variable an initial value, you must enclose the single character in **single quotes**. However, the single quotes will not be output during your printf statement. For example:

```
char   answer;
answer = 'Y';
printf ("Your answer is: %c\n", answer);
```

The output of the above program segment would be:

```
Your answer is: Y
```

Notice that we enclosed the character **Y** in single quotes in the program, however the quotes were not output in the **printf** statement. The reason we need to do this is because if we didn't, then C would assume that we were assigning variable **answer** equal to *variable* **Y** (because **Y** is a valid variable name), and we would get a compiler error that we have not yet declared variable **Y**!

That's all I have regarding data types. If you have any questions at all regarding data types, please post them in the discussion board for Week 1. I realize that programming concepts can seem fairly abstract, and I want to be sure that you have a thorough understanding of each-and-every concept so far. We will be

using all that we have learned so far in all future examples and programs. So, I want to be sure that you "get it".

Next, we will discuss the type of "arithmetic" (math) you can perform in C!

Arithmetic in C

+, -, *, /, %

In C, your programs will often perform very basic mathematical operations in order for you to solve your problem. Less often you will be required to do some very complex mathematical operations, in which case you will be using functions that have been written for you, and are available in the **math.h** Standard C Library for your use. Those are not the functions that I am referring to in this section. In this section, we will discuss the very basic mathematical operators available to you for performing simple math in your program. Keep in mind, most C programs perform some type of arithmetic calculation discussed below.

In C we use familiar arithmetic operators (similar to regular math). The following table shows the math operators used, and their purpose:

OPERATION	OPERATOR
Addition	**+**
Subtraction	**-**
Multiplication	*****
Division	**/**
Modulus (remainder)	**%**

Here is a very fundamental rule for you. It is only 5 simple words, but they are VERY IMPORTANT. Here they are: ***Integer math yields integer results*.**

Now, you probably need to know exactly what that means. (I don't blame you.) What I mean by that statement is if you have a calculation where all of the values being used in the calculation are integer, then the result is of type integer. If however, at least one of the values in the calculation is a floating point number, then the result is of type float.

An example will help explain this further:

```
int  sum;
int  num1 = 10;
int  num2 = 4;

sum = num1 + num2;  /* addition */
```

In this case, the value of variable **sum** after the addition is **14**. Notice both **num1** and **num2** are of type **int**, so, the result of this addition is of type **int**. Good thing,

because we are trying to stuff the result of the addition into a variable of type **int** (variable sum)!

Let's try another calculation using the above variable declarations:

```
sum = num1 / sum2; /* division */
```

In this case, the answer will still be of type **int** because "integer math yields integer results", and *both* **num1** and **num2** are integers. So, the result of integer division is also of type **int**. The value of **sum** in this case would be 2. (10/4 = 2).

What about:

```
sum = num1 % num2;   /* modulus or remainder */
```

As you probably guessed. The remainder of 10 / 4 is also **2**. So, the result of the math operation above would be 2.
If we modified the data type of **num2** to be **float**, and had the following code segment:

```
int    sum;
int    num1 = 10;
float  num2 = 4.0;

sum = num1 + num2;
```

Well, here we have a problem. Now, variable **num2** is no longer an integer. So, the result of this addition is of type **float**. We are trying to stuff a floating point result (which would be 14.0) into integer variable **sum**. Some compilers would handle this okay (they would chop off the .0, and stuff the number 14 into variable **sum**). But not all! So, you want to keep your code as portable as possible by NOT doing this sort of thing.

Recall another very important rule: *The data type of the data on the right side of the assignment operator should be the same as the data type of the variable on the left of the assignment operator.*

In the above (incorrect) example, if we want to resolve the problem, we have 2 choices:

1. Modify the data type of variable **sum** to be of type **float**. Then the data type on both sides of the assignment statement would be the same. (Recall, only 1 of the variables in a mathematical expression needs to be **float** for the calculation to yield a **float** result.)

2. If you wanted to keep **sum** as type **int**, and you wanted to keep **num2** as type **float**, you can do something quite "magical" in the programming world. It is called "*type casting*". (This sounds like something right out of a Harry Potter book!) Actually, it's pretty straightforward, and VERY handy for ensuring that your data types match. What you do is you "temporarily" modify the data type of the variable in question, to be the data type you want it to be. This type-casting has no effect on the original data type of the variable. How do you do this you ask? Well like this. I will type-cast variable **num2** to be of type **int** just during the addition, so that both sides of the assignment operator will be of type **int**:

sum = num1 + (int)num2;

Do you see the data type **int** in parenthesis right in the addition expression?!?!? That is how we typecast. I have told the C compiler to *temporarily* make variable **num2** to be of type **int**. Only in this statement will it be of type **int**. If this program segment had additional lines of code using variable **num2** following this statement, it would be considered its original data type, which is of type **float**. You can typecast any variable to be of any data type you choose.

A good programming practice: *put a space between the operators and variables.* Notice, the two statements below, which one do you think is more readable?

1. **sum=num1+num2;**
2. **sum = num1 + num2;**

The more readable statement is #2 because of the spaces surrounding the operators and variables. Very simple to do, yet very effective!

One last point in this section, there is an "*order of precedence*" when working with mathematical operators in C. It follows the same precedence as in regular math. That is, if you have an expression with more than one math operator, C processes the expression from left to right, and will do multiplication and division first, then addition and subtraction.

Example:

int result;

result = 3 + 4 * 2;

In this case, **result** would equal **11**, because the multiplication has a higher precedence than the addition. If you in fact wanted to do the addition first, then you must use paranthesis, which has a higher priority than the multiplication (and division). It would look like:

result = (3 + 4) * 2;

Now, **result** would equal **14**, the desired value (if you wanted to add first).

My advice to you (another readability issue), is to *use parenthesis* if you have more than 1 operator because it is *more clear*. For example, even if you wanted to perform the multiplication above first, it is much more clear that you intend to do the multiplication first if you type in your code as follows:

result = 3 + (4 * 2);

In this case the parenthesis are not necessary, since C will do the multiplication first anyway, however, you must agree, that by adding the parenthesis, it is a heck of a lot more clear that the multiplication is being done first.

Oh yes; one last thing (really). **Do not** attempt to **divide by the number zero**! What happens to your program if you try varies from machine to machine. Some will give you a compiler error (if the compiler detects it). Other machines will compile and link fine, and then give you a fatal error during runtime. Some machines will hang during runtime, and you will have to reboot. This can be a very time-consuming process. You should simply try to avoid it.

Please review and step through program 3.2 on page 31.

This will reinforce the concepts discussed in this section.

That's all for basic math in C. We will use basic math throughout our entire course. Again, if you have any questions regarding this segment, please post them in the discussion area for this week.

That’s all for lecture notes this week. And now, the real fun begins. It’s time to start working on your first programming assignment!

Next week we will be discussing 2 topics that the textbook chooses to discuss in later sections of the textbook. I feel however, that it is a good point in the course to review both of these topics. The topics discussed next are: "Program input" (prompting the user), and the "Standard C Library".

"scanf" - The input function in C!

Just as C provides the output function **printf** in the Standard C Library for processing program output, it also provides the *input function* **scanf** for processing program input.

When you make a call to function **scanf** from within your program, at program execution time the CPU will simply wait at that statement until the user types something in. Once the user types something in, and presses the "Enter" key, then program execution will continue with the next executable statement following the **scanf** statement.

One word of advice here: Always provide the user with some text (prompt message) by using a **printf** statement prior to using your **scanf** statement. If not, the user will have no idea what they are supposed to enter, as the CPU will simply wait there for something to be typed in. So, 99 out of 100 times you will see a **printf** statement before a **scanf** statement in the code.

Let's take a look at an example using **scanf**, then we will explain the function in greater detail.

The following program will prompt the user for 2 numbers, add them together, and output the result.

```
#include <stdio.h>
void main(void)
{
    int value1, value2, sum;
    printf ("Please enter a number: ");
    scanf ("%i", &value1);
    printf ("Please enter a second number: ");
    scanf ("%i", &value2);

    sum = value1 + value2;
    printf ("\nThe sum of %i and %i is %i\n", value1, value2, sum);
    getchar( );
}
```

The output of the above program might look like the following (shown in **blue** print for clarity only). What the user might type in is shown in black.

```
Please enter a number: 100
Please enter a second number: 22
The sum of 100 and 22 is 122
```

Cool huh?!?! :-) (I love this stuff!!!)

In the above sample program execution the user typed in 100 for the first number, then 22 for the second number. The program then continued to calculate the sum and then output the results. As you can see, **scanf** is a very useful function, but not too difficult to use. Notice that I provided a **printf** statement before *each* of the **scanf** statements, otherwise the user would have not idea what to enter.

Before discussing the **scanf** function call, I want to bring up a few *user interface* related issues regarding the "prompt" above. Notice the **printf** statement:

```
printf ("Please enter a number: ");
```

This printf statement did not contain a "**\n**" within its string. Did you notice that? Do you think I accidently forgot it? Never! Each and every character I type in has important meaning! :-) The reason I did not provide a "**\n**" (newline) in the **printf** prompt is because I wanted the user to enter the number *to the immediate right* of the prompt. If I put newline character in the **printf** statement, the prompts to the user would look like:

```
Please enter a number:
100
Please enter a second number:
22
```

Another user-interface issue that I want to point out here is that I put a space between the colon (**:**) and the end quote (**"**) in the prompt. This too was purposely done. If I placed the end quote right up-against the colon, as in the following code:

```
printf ("Please enter a number:");
scanf ("%i", &value1);          ^ notice no space here (not good)
printf ("Please enter a second number:");
scanf ("%i", &value2);                  ^ notice no space here (not
good)
```

The prompts resulting from the above statements of code would look like:

```
Please enter a number:100
Please enter a second number:22
```

Notice no space between the prompt and the data that the user types in. This is not very readable.

Now, we can finally move on to the **scanf** function. (Whew!)

scanf - 2 arguments

As you can see from the above examples, **scanf** requires 2 arguments in its function call. The *first argument i*s the ***format specifier*** (sound familiar?) of the variable where **scanf** will *store* the result typed in by the user; and the *second argument* is the ***name*** of the variable where **scanf** will store the result typed in by the user.

In our example, variable **value1** is of type **int**, so the **%i** format specifier was used in the **scanf** statement. The same is true for variable **value2**.

Note: the special symbol "**&**" is required before the variable name in the second argument in the function call to **scanf**. This is the "**address operator**". It tells **scanf** the "address" of the location of the variable in memory, so that **scanf** knows where to place the data that the user types in. Addresses of variables are also known as "**pointers**". Pointers will be covered during Week 8. All you need to know at this point is that you need to put the address operator before a variable name when calling **scanf**.

I will provide 1 additional example here. The following program prompts the user for an integer (student ID) a floating point number (student grade point average), and a character (middle initial). It then outputs the information back to the user. See how **scanf** processes each one of these.

```
#include <stdio.h>
void main(void)
{
        /* Declare variables */

        int     student_id;
        float   student_gpa;
        char    middle_init, c;

        /* Prompt user for infomation */

        printf ("Please enter your Student ID: ");
        scanf ("%i", &student_id);
        while ( (c = getchar() != '\n') && c != EOF);   /* something
        new, described below. */

        printf ("Please enter your Grade Point Average (GPA):
        ");
        scanf ("%f", &student_gpa);
        while ( (c = getchar() != '\n') && c != EOF);   /* something
        new, described below. */
```

```
        printf ("Please enter your Middle Initial: ");
        scanf ("%c", &middle_init);
        while ( (c = getchar() != '\n') && c != EOF);   /* something
        new, described below. */

        /* Display info back to user. */

        printf ("\n\nYour Student ID is %i\n", student_id);
        printf ("Your GPA is %.1f\n", student_gpa);
        printf ("Your Middle Initial is %c\n", middle_init);

        getchar( );
    } /* end main */
```

A sample program output of the above code might be:

```
Please enter your Student ID: 200
Please enter your Grade Point Average: 3.5
Please enter your Middle Initial: S


Your Student ID is 200
Your  GPA is 3.5
Your Middle Initial is S
```

First, you notice a new line of code:

```
while ( (c = getchar() != '\n') && c != EOF);
```

This is added to ensure 100% portability. With some systems, the carriage return (\n) is left over in the *input buffer* from the previous scanf (in this case, the GPA). The *input buffer* is a memory location Data is temporarily placed in this input buffer as soon as the user presses the "Enter" key. The **scanf** function then reads the data from the input buffer. With character data, **scanf** will read only a **single character** from the buffer. So, for our example, if you didn't "flush the input buffer", the carriage return left over from the GPA input will remain in the input buffer, and **scanf** will load it (the carriage return) into variable **middle_init**, instead of waiting for you to enter something for the middle initial. (They system thinks you already entered something!)

Also, you want to be sure to get rid of the **\n** that is stored in the input buffer **after** you enter the middle initial. As mentioned, when you scan in a single character, only a single character is read in from the input buffer, leaving your carriage return there. The above line of code "*flushes*" the input buffer (i.e., pulls any remaining characters out of the input buffer), so that there is no data in the input

buffer before, and after the **scanf** for the single character. This ensures 100% portability.

Also notice that there is no "error checking" on user input here. This program would easily "crash" if the user attempted to enter a very large number as the student ID (more than 32,767), or the user tried to enter a number instead of a letter for the middle initial. Normally, we would add code to the program to ensure that the user is entering valid data. This is a very tedious, but necessary process. We will cover some of the basics of "data error checking" throughout this course.

The other point I want to make here is regarding floating point data and **scanf**. Notice that when I prompted for the GPA, I used only "%f" in the call to function **scanf**. I did not specify a number of decimal places in the scanf (e.g., "**%.2f**" or "**%.1f**") ***because we cannot do this in a* scanf**. We cannot control how many decimal places the user will enter during the **scanf** statement. What is *really* important is how many decimal places are displayed during ***output.*** So, we specify the appropriate number of decimal places when we output the information, using the **printf** statement.

If the user entered 3.5 as the GPA value, and the **printf** statement for the GPA was as follows:

```
printf ("Your GPA is %f\n", student_gpa);
```

Then, printf would display all 6 decimal places, since we did not put a "point" and a number between the **%** and the **f** above. So, if the user indeed entered 3.5, the output would be:

```
Your GPA is 3.500000
```

Notice in the full program above that I used "**%.1f**" as the format specifier for the GPA. This ensures that I will see only 1 decimal place in the output even if the user entered more than 1 in the input!

The statement: printf ("Your GPA is %10.2f\n", student_gpa);

Would produce:

```
Your  GPA is      3.50
```

Notice the spacing after the word ‘is’. The 10.2f gives you two decimals in a total field size of 10 spaces. **That type of formatting helps you line up decimal places** in lines of output that may have many different sizes of numbers.

That's about all there is to discuss regarding **scanf** at this time. The next topic discusses another important aspect of C, namely the Standard C Library.

The "Standard C Library" - A programmer's toolbox!

The Standard C Library: A collection of pre-existing functions (blocks of code) for performing common programming tasks.

These functions have been written for you, and are included when you purchase a C compiler. We use these pre-existing functions to avoid re-inventing the wheel. This is known as *software reusability*.

We have already had some exposure to the notion of program modularity! See that? You have already done all of this stuff! In the programs we have reviewed/created thus far, we have written the code for function **main** for each program. Then, we have made calls to other functions in the Standard C Library, namely **printf** and **scanf**! Functions **printf** and **scanf** are simply chunks of code that perform the input and output processing that we need. They are located somewhere on your hard drive (placed there when you installed your compiler), and linked with the code in function **main** during the link phase. These functions in the Standard C Library have already been compiled, and only the "object code" resides on your hard drive.

Because C provides a wealth of wonderful, useful code in its library, the first step to becoming a good C programmer is learning *which* functions are available in the Standard C Library. What good is having a library if you don't know what it contains?!

C categorizes related functions together and places their "function definitions or prototypes" in the files with the .h extension. These files are known as "header files" or "include files" and are discussed in more detail during Weeks 4 and 11. For example, the function definitions for all of the functions which perform some sort of input/output (I/O) operations are grouped together in the **stdio.h** header file (stdio is short for standard I/O). We have "included" this header file in our programs so far because we wanted to use the input and output functions defined in that particular header file, namely **scanf** and **printf**. There are other input/output functions in C, as you will see in the summary below.

Appendix B introduces the Standard C Library. Please be sure to reference it throughout this course as you will perhaps need to use some of the functions provided by the library.

A Summary of Appendix B - The Standard C Library.

Note that header files may contain "constants" instead of, or in addition to functions. Constants provide vital information to our programs. We discuss constants in greater detail in Week 7.

Header file (group)	Type of information included in this group.	Pages in Text
stddef.h	Standard constant definitions	471
limits.h	Constants defining the limits for integer data types.	472
float.h	Constants defining the limits for floating point data types.	473
string.h	Functions for processing string data.	475
	Functions for more efficient processing of string data in memory	
ctype.h	Functions for processing character data.	476
stdio.h	Functions for processing input/output data.	477
	Functions for processing in-memory format conversions	
stdlib.h	Functions for converting string data to numeric data.	483
	Functions for processing dynamic memory allocation.	
	General Utility Functions	
math.h	Functions for processing complex math functions.	485

I am not requiring you to study or know the Standard C Library, but just to know of its existence.

Well. That's it for this week! Not too bad huh? I realize that this topic was a bit "dry", but it was one of those things that I needed to discuss at this point. It's like fresh spinach. It may not taste so good, but it's good for you!

Next week we will discuss the very important, fundamental concept of: Program Looping. (I'm starting to feel a bit "loopy" myself. Speaking of dry, I think I'll go make myself a martini! Did I say martini?!?!? I meant to say a nice dry ginger ale! Yah that's it!)

And, as a reminder, your **1st programming assignment** is **due by the end of this week**.

Structured Programming vs. Unstructured Programming.

Note: This information is *not* found in the textbook**.**

The techniques we use to program today have greatly improved since the days of "unstructured programming". In the *old* days, we would have a segment of code, and then "**goto**" another segment of code, and then perhaps "**goto**" back to the original segment in order to repeat the code. This form of executing and repeating code via **goto** statements is considered **unstructured programming**. The code itself is referred to as "spaghetti code" because the flow of the programming statements is quite sloppy (like a big plate of spaghetti!).

Structured programming, on the other hand, is very organized and... well... structured (more like lasagna, since we're talking about Italian food). Structured code ultimately leads to programs that are more easily readable and modifiable, as well as code that is more efficient regarding processing speed and memory utilization. We organize our code using all of the following techniques to ensure that we are following structured programming standards:

- Sequence Structure - code flows in sequence from beginning to end.
- Repetition Structure - **for**, **while**, and **do** loops.
- Selection Structure - if, else, else if, and switch statements.

Sequence Structure

Sequence Structure refers to code that is executed in sequence. That is, the CPU starts at the beginning of the program (function main) and executes each instruction, one at a time, in sequence. There is no such thing as jumping back and forth throughout the program (which is what happens when we use **goto** statements). Execution completes at the end of function main (at the end brace). So far, all of the code we have reviewed in this course has followed this "sequence structure" standard.

Repetition Structure

Repetition Structure refers to code which repeats itself efficiently. Types of structures which fall into this category include all of the looping structures such as: **for** loops, **while** loops and **do** loops. We cover program looping this week.

Selection Structure

Selection Structure refers to code which makes choices, or selections, based on certain conditions. Types of structures which fall into this category include: **if**, **else**, **else if**, and **switch** structures. We cover these structures in weeks 5 and 6.

The next topic discusses the second of the above three structured programming techniques, namely: the ***Repetition Structure***.

Program Looping with the "for" loop.

One fundamental and necessary property of a computer is its ability to repetitively execute a set of instructions. The challenge here is to repeat code WITHOUT using "goto" statements. (A no-no in today's programming!) As we continue with programming we will quickly find the need to have portions of our programs repeated several times. We do this even in "real life". For example, if I wanted my children to take 5 bites of their vegetables, I don't say to them:

Take 1 bite of your vegetables
Take 1 bite of your vegetables
Take 1 bite of your vegetables
Take 1 bite of your vegetables
Take 1 bite of your vegetables

Instead, I would say: Take 5 bites of your vegetables.

(Although I have been found to repeat myself quite often with my children. I'm sure you know what I mean: Come here. Come here. Come HERE. COME HERE!!!!) Okay. I'm alright. Back to business...

We want this same ability with our computer programs. That is, to repeat a single, simple instruction, or perhaps a group of instructions, without having to literally repeat the code in our program. How do we do this?

C provides 3 tools for repeating code in programs. The are the "Repetition Structures", more commonly referred to as "loops". The 3 looping structures in C are:

1. The **for** loop
2. The **while** loop
3. The **do** loop

Although any of the above 3 looping structures can be used to repeat code in C, it is a good idea to get to know the slight differences amongst them because programming ***efficiency*** can be greatly enhanced if you select the loop that is best suited for solving you problem. This section discusses the first of the 3 looping structures: the for loop.

The for Loop

The for loop allows the programmer to specify that an action (1 or more statements of code) is to be repeated a specified number of times. So, if you know that the loop is to be repeated exactly 5 times, for example, then the for loop is a good one to choose. Using the previous analogy with the vegetables, I

would pick a ***for loop*** to solve the problem, because I know that I want my children to eat exactly **5** bites of their vegetables.

Let's look at a program which uses a for loop to output the numbers from 1 to 5 along with their squares. The output of the program will look like:

Hello, and welcome to the squares program!

1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25

Goodbye!

It would be easy enough to simply create 7 **printf** statements in our program to output these 7 lines, however, what if I wanted to output 100 squares? How about 1000? You will see how useful using repetition structures are when you realize that most computer programs will loop hundreds, thousands, even millions of times to process data!

Let's get back to the above problem. The code, using a for loop, would look as follows.

Note that the numbers shown to the left of some of the code are not actually part of the program; they are there to assist me in describing the code to you.

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

    /* Variable Declarations. */
    /*  ----------------------------- */

    int     x,  result;

    /* Output initial greeting. */
    /*  ----------------------------- */

    printf ("\nHello, and welcome to the squares program!\n\n");

    /* Calculate and output the squares for numbers between 1 and 5. */
    /* ------------------------------------------------------------------------------------------ */
```

```
1.    for (x = 1; x <= 5;  x = x + 1)
2.    {
3.         result = x * x;
4.         printf ("%i squared is %i\n",  x, result);
5.    } /*  end for loop  */

      /* Output the final greeting. */
      /* ------------------------------- */

      printf ("\nGoodbye!\n");

} /*end main   */
```

Let's take this one line at a time. The lines that are not numbered are lines that you have seen before. Variable declarations, as well as the initial and final greetings are nothing new. What is new here is the code at lines **1** - **5**.

Line 1: `for (x = 1;  x <= 5;  x = x + 1)`

```
                ^       ^          ^
          1st part  2nd part  3rd part
```

This is the beginning statement in a **for** loop. It is composed of the reserved word: for, then 3 segments (parts) of information, separated by semicolons, inside parenthesis. Notice NO SEMICOLON at the end of this initial **for** statement. More on that later. I will break down the 3 parts of the **for** loop statement and discuss each one separately.

1st part: *The initial value* (**x = 1**)

This segment of the for loop *sets the loop control variable to some initial value*. That is, in this example, I am using variable **x** as my **loop control variable**, and I am initializing it to **1**. The loop control variable (also known as a **loop counter)** controls the flow of the loop. (Note: Variable names **for** loop control variables do not have to be descriptive, and are commonly single character variable names.)

Since I want to process the numbers from 1 to 5, it makes sense to set this to this initial value 1. This portion of the **for** loop is executed only *once*, the first time through the loop. Depending on our problem to solve, we can set this initial value to any integer. For example, if I wanted to calculate and display the squares from 5 to 1 (decreasing order), it would make sense to set this initial value to 5. Remember, this initial value can be any value, as long as it is a whole number.

2nd part: *The loop condition (or final value)* (**x <= 5**)

This segment of the **for** loop *tests the loop control variable against some final value*. That is, in this example, we test that the loop control value, **x,** is "**less than**

or equal to 5". The **<=** operator is one of the 6 **Relational Operators** discussed in the next topic. As of now, you simply have to know that the test being performed in our example, using **<=** is as follows: is the loop control variable less-than or equal to the number 5? The loop will continue as long as this condition is true. However, as soon as this condition is no longer true, once variable x contains the value 5 in our example, the for loop will no longer execute. This portion (loop condition) of the for loop statement is executed each time through the loop.

3rd part: *The loop increment/decrement* (x = x + 1)

This final segment of the **for** loop *increments (or decrements) the loop control variable*. It is how the loop control variable gets from its initial value (**1**, in our example) to its final value (**5**, in our example). Here we add **1** to variable **x** each time through the loop. Without this portion, the loop control variable would never reach its final value, and the loop would execute forever (infinite loop). We could increment by 2, if we so choose (**x = x + 2**), or by whatever integer increment you choose. You can also decrement here, if for example, you need to process values in descending order. For example, if I wanted to display the squares for values between **5** and **1**, I would use the following for loop statement:

for (x = 5; x >= 1; x = x - 1)

In the above statement, we initialize our loop control variable **x** to be the number **5**, we test that it is *greater or equal* to **1**, and then we **decrement** the loop control variable each time through the loop.

There are a variety of ways to represent the incrementing and decrementing of variables in C. Since incrementing and decrementing is so often performed, C has abbreviations for these sort of operations. These abbreviations will be discussed in a separate topic this week.

Code sequence plays a very important role when discussing loops. At this point it is difficult to understand when part 1, part 2, and part 3 are executed. Once we conclude our discussion of **lines 2 - 5** from the above program, I will discuss code sequence in more detail.

Lines 2 and 5: { and }

As you probably guessed, these braces mark the **begin** and **end** of the block of code that make up the *body* of the **for** loop. Between these braces are the lines of code that will be repeated, through each iteration of the loop. Notice following the begin brace, I have indented the statements of code that make up the body of the loop. This is a good programming standard to follow, as it greatly improves readability. It is very consistent with the indentation that takes place between the begin and end braces of function main.

C allows you to leave the braces out if the body of the for loop consists of just a single line of code. For example, if you had a for loop that simply output: **Programming is fun!** 10 times, you could code it in either of the 2 ways below. The first method uses braces around the body of the loop. The second method does not use the braces. Either method is acceptable. Again, this is not a very practical program, but it helps to illustrate the concept.

First method (using braces):

```
void main (void)
{
    int   x;

    for (x = 1;  x <= 10;  x = x + 1)
    {
        printf ("Programming is fun!\n");
    }

} /*end main   */
```

Second method (no braces):

```
void main (void)
{
    int   x;

    for (x = 1;  x <= 10;  x = x + 1)
        printf ("Programming is fun!\n");

} /*end main   */
```

Both of these programs will do the exact same thing. One thing that I want to point out here is that even if the body is just 1 line of code, you should still **indent** that single line of code, as illustrated above.

Lines 3 and 4:
```
result = x * x;
printf ("%i squared is %i\n", x, result);
```

These 2 lines make up the **body** of the **for** loop. The body of the **for** loop are the statements of code that will be repeated each iteration through the loop. These statements are also known as the **block** of code that makes up the **for** loop. In this particular example, **Line 3** uses the loop control variable **x**, and multiplies it by itself to get the squared value! Keep in mind that the value of variable **x** is **1** the first time through the loop. It is then incremented, and becomes **2** the second time through the loop. Then it is **3** the third time through the loop, and so on. This loop control variable **x** becomes a perfect variable for using within the loop for

calculating the squares of the numbers **1** to **5**, since this loop control variable contains the desired number each time through the loop.

It is not uncommon for loops to use the loop control variable within the body of the loop as part of a calculation. The only thing you should avoid is modifying the value of the loop control variable within the body of a **for** loop. Recall, the value of the loop control variable is automatically updated each time through the loop in the 3rd segment of the **for** loop statement. You should not mess with it otherwise.

So, now we have the calculated value we want stored in variable **result**. **Line 4** is a simple **printf** statement that outputs the required information. Notice the loop control variable is used once again in this statement as part of the output!

At this time I will mention **code sequence**. The sequence of events is so important where loops are concerned. Another term used when referring to the sequence that the statements are executed in a loop is called: **Flow Control**.

Flow Control in a **for** loop:

1. The loop control variable (counter) is initialized (First segment of **for** loop statement).
2. The loop condition is evaluated (Second segment of **for** loop statement).
3. If the condition is **true**, the body of the loop is executed; the loop control variable is then incremented or decremented (3rd segment of **for** loop statement). Go back to **step 2**.
4. If the condition is **false**, the loop is terminated. Go to **step 5**.
5. Program execution continues with next executable statement following the loop.

Additional comments regarding the **for** loop:

- Initialization is only done once (step 1 above).
- The test is done before the body is executed (step 2 above).
- If the test is false initially (step 1 above), the body is not executed, thus *the body will not be executed at all*.

Additional Programming Examples:

I will provide a few additional programming examples here to be sure you understand loops fully. I am not too pleased with the programming example 5.2 on page 44 in the textbook, as I feel the triangular number algorithm on is a bit complex for explaining **for** loops, *although some people thoroughly enjoy it*. I think it is best to keep the algorithm simple while learning these concepts.

The program below is similar to the one we've worked on above (the square program), however it has been modified to prompt the user for the desired start and end numbers to be squared! This requires just a simple change, yet makes the program much more flexible.

The output of the program looks like:

Hello! This program will square a sequence of numbers for you.

Please enter a starting number: 3
Please enter an ending number: 10

3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100

Goodbye!

The code for the above program might look like:

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

      /* Variable Declarations. */
      /*  ------------------------------- */

      int     x,  result, initial_value, final_value; /* notice 2 new vars here */
      char  c;                                        /* for clearing input buffer */

      /* Output initial greeting. */
      /*  --------------------------- */

      printf ("\nHello! This program will square a sequence of number for
you\n\n");

     /* Prompt user for starting and ending numbers. */
     /* ------------------------------------------------------------------ */
```

```
    printf ("Please enter a starting number: ");
    scanf ("%i", &initial_value);
    while ( (c = getchar() != '\n') && c != EOF);      /* clear input buffer  */

    printf ("Please enter an ending number: ");
    scanf ("%i", &final_value);
    while ( (c = getchar() != '\n') && c != EOF);      /* clear input buffer  */

    /* Calculate and output the squares for numbers entered by user.*/
    /* ------------------------------------------------------------------------------------- */

    for (x = initial_value;  x <= final_value;  x = x + 1)
    {
        result = x * x;
        printf ("%i squared is %i\n",  x, result);
    } /*  end for loop  */

    /* Output the final greeting. */
    /* ---------------------------------- */

    printf ("\nGoodbye!\n");

} /*end main   */
```

Notice that the only real change that needed to be made to the **for** loop was in the **for** statement itself. Instead of setting the *initial value of* ***x*** to **1**, we set it to the *starting number* entered by the user. Also, instead of setting the *final value of* ***x*** (or loop condition) to **5**, we set it to the *ending number* entered by the user. This provides for a much more flexible program, as the user can enter any numbers (1 to 100, for example), and the program would process those numbers without any problem. The only thing you need to be concerned with here, is that if the user enters a number whose square would results in a value greater than 32,767 (the largest number that an integer can contain), it would not be able to be stored in variable **result**. So, you may want to make the data type for variable result to be of type **long** to be able to store the larger values. The other alternative is to test the numbers entered by the user to be sure they do not exceed a certain range, but we have not learned how to do that yet! :-)

One last comment to be made at this time. I know what your thinking: that "he always has **one last comment**". (Did I say I was psychic?!?!) Anyway, I wouldn't be able to sleep tonight if I neglected to tell you the following. And besides, I think you'll like it! :-)

The body of the for loop above contains 2 statements. I want you to be aware that you could *combine* the 2 statements into 1 statement! This has nothing to do with loops, but everything to do with **printf**. Recall that within a **printf** statement,

when you have a format specifier, the **printf** function will look for a value to stuff into the spot in the **printf** string. The value that the **printf** outputs is usually located in a variable following the end of the **printf** string. Well, that variable *does not have to be a variable*. It could instead be a calculation! In other words, the **printf** statement would obtain the value needed, not from the contents of a variable, but instead from the results of a calculation.

So, in our example, where we have:

```
result = x * x;
printf ("%i squared is %i\n",  x, result);
```

We could have instead used

```
printf ("%i squared is %i\n", x,  x*x);
                                  ^
```

calculation performed inside the **printf** statement

Notice the calculation for the square (**x * x**) is performed directly *inside* the **printf** statement. This is legal, valid, and often done, ***even though I do not like this in a program***. In this example, if you did the calculation directly in the **printf** statement, then there would be no need for variable **result** at all! The body of the **for** loop would be whittled down to a single statement. Some people prefer this. Others, do not. C provides so many "shortcuts", and I believe that if too many are used, they can actually reduce the readability of the program. In this case, I would ***definitely*** keep the calculation separate from the **printf** statement.

The next section of notes discusses the 6 relational operators.

Relational Operators.

Relational operators are used in C for testing conditions of certain variables. For example, we might want to test if a variable equals some number, such as zero. Tests are often made within repetition structures (**for**, **while** and **do** loops), as well as within selection structures (**if**, **else**, **else if**, **switch**). We need a way to represent to the computer that we want to test the contents of a variable. For this reason, we have the relational operators.

When a test is made using a relational operator, C responds with either a *true* or *false* response. In C, the value *true* is represented by a **1**, and the value *false* is represented by a **0**. This concept of *true* being equal to **1** and *false* being equal to **0** is not important at this time. In fact, you can use relational operators without ever having to consider the "values" of true and false. More on this later.

There are 6 relational operators. They are shown in the table below, and later described in more detail.

Relational Operator Symbol	Test being made
==	is equal to
!=	is not equal to
<	is less than
>	is greater than
<=	is less than or equal
>=	is greater than or equal

1. "Is Equal To" Relational Operator: ==

This relational operator tests for equality, and is represented by 2 equal signs together with no spaces between them. This relational operator is not commonly used as the loop condition in a **for** loop, because it is not very practical. It is however commonly used in **while** and **do** loops, as well as in the **if**, **else**, **else if** and **switch** statements. Below is a segment of code which uses the equality relational operator in a **for** loop (since it is the only loop covered so far). You can see how the body of this particular loop will not execute at all because the loop condition (**x == 5**) is false to begin with:

```
for (x = 1;  x == 5; x = x + 1)
{
    printf ("I am in the loop!\n")
```

```
}
printf ("Goodbye.\n");
```

In this example the following sequence takes place:

1. variable **x** is assigned the value **1** (**x = 1**).
2. variable **x** is tested against the value **5** (**x == 5**).
3. this test is false, so the body of the **for** loop (the **printf** statement) is not executed.
4. the program continues with the "goodbye" **printf** statement.

The output of the above program would be:

Goodbye!

*Please be sure that you do not confuse the equality relational operator (**==**) with the assignment operator (**=**).* This is a very common programming error, and can lead to many hours spent debugging your program. This type of error is difficult to detect visually. (Particularly at 2:00 a.m. when most programmers are debugging their code! :-))

2. "Is Not Equal To" Relational Operator: !=

This relational operator tests for in-equality, and is represented by an exclamation point followed by an equal sign, with no spaces between them. This relational operator is not usually used as the loop condition in a **for** loop because for loops usually run from some initial value, until a final value. It is however commonly used in **while** and **do** loops, as well as in the **if, else** and **else if** statements. Below is a segment of code which uses the in-equality relational operator in a **for** loop (again, since it is the only loop covered so far). This particular loop will execute 4 times.

```
for (x = 1;  x != 5; x = x + 1)
{
    printf ("I am in the loop!\n")
}
printf ("Goodbye.\n");
```

The first time through the loop, **x** is assigned **1**, and the body of the loop is executed (because the loop condition (**x != 5**) is true.

The second iteration, **x** is incremented to **2** and again the loop condition (**x != 5**) is true, so the body of the loop is executed.

The third iteration, **x** is incremented to **3** and again the loop condition (**x != 5**) is true, so the body of the loop is executed.

The fourth iteration, **x** is incremented to **4** and again the loop condition (**x != 5**) is true, so the body of the loop is executed.

The fifth iteration **x** is incremented to **5** and again the loop condition (**x != 5**) is false, so the body of the loop is NOT executed.

The output of the above program is as follows:

I am in the loop!
I am in the loop!
I am in the loop!
I am in the loop!
Goodbye!

3. "Is Less Than" Relational Operator: <

This relational operator tests that a value is *less than* another value, and is represented by a *less than* sign. This relational operator is commonly used in all the looping structures (**for**, **while**, **do**), as well as in the **if**, **else**, and **else if** selection structures. Below is a segment of code which uses the less-than relational operator in a **for** loop. Try to figure out how many times this the body of this loop will execute.

```
for (x = 1;  x < 5; x = x + 1)
{
    printf ("I am in the loop!\n")
}
printf ("Goodbye.\n");
```

As you might have guessed, the body of this loop will execute 4 times, and the output of the above program is as follows:

I am in the loop!
I am in the loop!
I am in the loop!
I am in the loop!
Goodbye!

4. "Is Less Than Or Equal To" Relational Operator: <=

This relational operator tests that a value is *less than or equal to* another value, and is represented by a less than sign followed by an equal sign, with no spaces between them. This relational operator is commonly used in all the looping structures (**for**,

while, **do**), as well as in the **if**, **else**, and **else if** selection structures. Below is a segment of code which uses the less-than relational operator in a **for** loop. Try to figure out how many times the body of this loop will execute.

```
for (x = 1;  x <= 5; x = x + 1)
{
    printf ("I am in the loop!\n")
}
printf ("Goodbye.\n");
```

The body of this loop will execute 5 times. The output of the above program is as follows:

```
I am in the loop!
I am in the loop!
I am in the loop!
I am in the loop!
I am in the loop!
Goodbye!
```

5. "Is Greater Than" Relational Operator: >

This relational operator tests that a value is *greater than* another value, and is represented by a *greater than* sign. This relational operator is commonly used in all the looping structures (**for**, **while**, **do**), as well as in the **if**, **else**, and **else if** selection structures. Below is a segment of code which uses the greater-than relational operator in a **for** loop. Try to figure out how many times this the body of this loop will execute.

```
for (x = 1;  x > 5; x = x + 1)
{
    printf ("I am in the loop!\n")
}
printf ("Goodbye.\n");
```

As you might have guessed, the body of this loop will not execute at all since the initial condition (**x > 5**) is false to begin with. The output of the above program is as follows:

```
Goodbye!
```

6. "Is Greater Than Or Equal To" Relational Operator: >=

This relational operator tests that a value is *greater than or equal to* another value, and is represented by a greater than sign followed by an equal sign, with no spaces

between them. This relational operator is commonly used in all the looping structures (**for**, **while**, **do**), as well as in the **if**, **else**, and **else if** selection structures. Below is a segment of code which uses the greater-than relational operator in a **for** loop. Try to figure out how many times the body of this loop will execute.

```
for (x = 1;  x >= 5; x = x + 1)
{
    printf ("I am in the loop!\n")
}
printf ("Goodbye.\n");
```

Again, the body of this loop will not execute at all since the initial condition (**x >= 5**) is false to begin with. The output of the above program is as follows:

```
Goodbye!
```

I guess the above output is fitting at this time because this ends the discussion of relational operators. (I see those smiles!) We will be using relational operators throughout the remainder of this course, so you are sure to understand them fully soon enough!

The next topic discusses the increment and decrement operators. These operators are very much used in looping, as well as other C statements.

Incrementing and decrementing - it has its ups and downs ☺

C provides the programmer with many abbreviations for common programming statements. Sometimes these abbreviations are useful, and sometimes they are simply too complex to understand. When they are too complex, readability is reduced, simply because the programmer didn't want to do much typing. Most of these abbreviations do not cause the code to be any more efficient, they simply make the programmer's job easier.

The reason I carry on about this is twofold. First of all, I love to type, and just thought I can get lots of practice with this class. ;-) Actually, first of all, when I introduce an abbreviation, it is because I have found it to be commonly used in industry. Second of all, I want to point out that if you choose never to use them, that is perfectly fine. However, you need to have some exposure to them, as you will see them along the way.

One of the most common abbreviations used is the one to increment or decrement a variable's value by 1. I use this one myself, as it is really nice, and I don't believe that it reduces the readability of the code.

Incrementing a variable's value by 1

To increment a variable's value by 1, you have 3 options. The first option below is the original "long hand" method. The second and third options are the abbreviated versions. Assume the following variable declaration, which declares variable **num** to be of type **int**, and initializes it to the value **25** during declaration. All of the options below increment variable **num** by **1**. So, assuming **num** starts off at **25** for each option, after the statements are executed, variable **num** is equal to **26**.

```
int  num = 25;
```

Option 1: **num = num + 1;** /* increment variable num1 by 1. */

Option 1 above shows the long-hand version of an increment.
We have seen this in the **for** loops covered so far.

Option 2: **num++; /*** increment variable num by 1. ***/***

Option 2 above shows an abbreviated increment statement. You put the variable name followed by the increment operator **++** (2 plus signs immediate following one another, with no spaces between). This causes variable **num** to be incremented by 1. You can instead place the **++** sign *before* the variable as follows: **++num;** When the increment operator is placed *before* the variable, it is known as a **pre increment**. When the increment operator is placed *after* the variable, it is known as a **post increment**. The difference between pre and post increments will be discussed later. The increment operator can only be used to increment

variables by 1. If you want to increment variables by anything other than 1 (2 for example), you need to either use the long version (option 1), or use option 3 below.

Option 3: **num += 1;** /* increment variable num by 1.*/

Option 3 above is another abbreviated form of the increment. You put the variable name followed by a space (not required), followed by a plus-sign then and equal sign, with no spaces between the plus and equal signs. Then you put the value that you want to add to the variable. In this case, we used the number **1**. So, we will be incrementing variable **num** by **1**. If we wanted to use this option to increment variable **num** by **2**, we could, simply by typing: **num += 2;**

So, in summary, regarding the increment, if you need to increment your variable's value by 1, you can pick any one of the three options above. If you want to increment your variable's value by anything other than 1, you can choose only options 1 and 3. Option 2 is solely reserved for incrementing by 1, and it is the most commonly used increment in for loops. (Recall, the 3rd segment of a for loop, "the increment or decrement".) Using the increment in option 2 a sample **for** loop statement would look like:

for (x = 1; x <= 5; x++)
^ notice the increment operator here.

<u>Decrementing a variable's value by 1</u>

Like incrementing, decrementing a variable's value by 1 can also be abbreviated. Again, you have 3 options. They follow the exact same format as the increment, with a slight modification. I think you will get the picture, even if I summarize below. All of the options below decrement variable **num** by **1**. So, assuming **num** starts off at **25** for each option, after the statements are executed, variable **num** is equal to **24.**

int num = 25;

Option 1: **num = num - 1;** /* long hand decrement of variable num1 by 1.*/

Option 2: **num--;** **/*** abbreviated decrement of variable num by 1. ***/**
/* could have been: **--num;** instead */

Option 3: **num -= 1;** /* abbreviated decrement of variable num by 1 */

Again, in summary, regarding the decrement, if you need to decrement your variable's value by 1, you can pick any one of the three options above. If you want to decrement your variable's value by anything other than 1, you can choose only options 1 and 3. Option 2 is solely reserved for decrementing by 1, and it is the most commonly used decrement in **for** loops. (Recall, the 3rd segment of a **for** loop, "the increment or

decrement".) Using the decrement in option 2 a sample for loop statement would look like:

for (x = 5; x >=1 5; x--)
^ notice the decrement operator here.

One final note regarding Option 3 above. You can use the abbreviation for other mathematical operations.

For example, if you wanted to *multiply* variable num by 3, you could use:

num = num * 3; /* Option 1.*/

Or, you could use:

num *= 3; /* Option 3 */

If you want to *divide* variable num by 5, you could use:

num = num / 5; /* Option 1*/

Or you could use:

num /= 5; /* Option 3.*/

That's about all there is regarding incrementing and decrementing. Next we learn to count. ☺

Step right up to the counter...and accumulate!

In programming, especially in loops, we take advantage of the loop index for counting (this is too obvious because the loop index counts for us!) and accumulating(adding up numeric data, or subtracting, multiplying etc.).

This is illustrated in several programs in the textbook in chapters 4 and 5.
The design for the addition (or subtraction) accumulator is as follows:

1. Set up a variable to accumulate the data, and set it *initially* to the value zero.

2. Within the loop add or subtract from the accumulator.

A facsimile (with my style of naming variables) of program 5.2:

```
int grade_total = 0;
....

 for ( i = 1; i <= number_of_grades; ++i)
 {
      printf ("Enter Grade #%i: ",i);
      scanf ("%i", &grade);

      grade_total = grade_total + grade;

 }
```

The variable **i** is the loop index.

Notice ***grade_total*** is on **both** sides of the equal sign. This takes the *current* value of ***grade_total*** (on the right), adds *grade* to it, then stores the result in ***grade_total***. That's it!

When the program exits the loop, the ***grade_total*** will be the total of all the grades entered during the loop process. It may seem a little fuzzy now, but once you understand the concept you will use it all the time.

Notice also the counter will be displayed as we put in the grades:

Enter Grade #**1**: (here the value of variable i is **1**)

Enter Grade #**2**: (here the value of variable i is **2** etc.)

At the end of the loop (i.e. outside the loop) you have the accumulated grades in ***grade_total***.

So, you could output them:

printf ("The total of grades is %i\n", grade_total);

then, to calculate an average:

ave_grade = (float) grade_total / number_of_grades;

I used the *typecast* **(float)** so that the computation would be computed as a decimal value, and if **ave_grade** is **float** type you can then print out as many decimals as you like. *Notice only one typecast is needed to make the entire calculation a float.*

For example for one decimal place:

printf ("The average of the grades is %.1f\n", ave_grade);

Simple as that!

The next topic this week deals with the concept of having a **for** loop within a **for** loop. This is known as: **insanity**. Oh, I mean: **nested for loops**. ☺

Here we go loop-dee-loo. Here we go loop-dee-light...

I think I might be going a bit loopy. Oh well, here we go.....

The While Loop

The **while** loop is a variation of the **for** loop. Like the **for** loop, the **while** loop is a looping structure which allows the programmer to specify that an action is to be repeated while some condition remains true (or, in other words, until the condition becomes false).

Please be aware that any problem that can be solved using a **for** loop, can also be solved using a **while** loop, and vice-versa. The reason we have the different loops is because, based on the problem, one loop may be better suited to solving the problem than the other. There are some very subtle differences between the **for** and the **while** loops, which we will discuss here. I also want to point out that first-time programmers tend to pick one loop and stick with it for all of their programming problems. Only when new programmers have some experience do they feel confident enough to choose different loops for solving different problems. I think I used the **for** loop for at least 3 months before I ever attempted to use any other loop!

The **while** loop is often chosen when the programmer ***does not know how many times the loop will be executed***. That is, instead of a *counter* controlling the loop, some other condition will control the loop. For example, if we want to write a program which prompts the user for 10 positive numbers and we keep a running total, and then output the sum, we could easily do this with a **for** loop. However, if we want to prompt the user for as many numbers as the user wants, and only stop when the user enters something like: -999, then we are not sure how many times the loop will execute. It will stop only when the user enters -999. This is an ideal application for using a **while** loop, and I will code it below.

However, before I get to that, I'd like to go back to the vegetable example from last week. (Have you noticed my food fetish??!?) Recall, if I told my children to eat 5 bites of vegetables, I could use a **for** loop to represent that. However, if I told my children to eat their vegetables until they were full, then I would have no idea how many bites they would be taking. They would stop only when they were full. This is an ideal situation for a **while** loop, rather than a for loop.

Now, back to the problem mentioned above. The following program will prompt the user to enter some numbers to add. It will keep prompting, and adding until the user enters **-999**. The program will then display the total of the numbers entered by the user. I will use a **while** loop here, since I do not know how many times to run the loop. Please pay particular attention to the format of the while loop.

A sample output of the program below might look like:

Enter a positive number (-999 to end): 10
Enter a positive number (-999 to end): 25
Enter a positive number (-999 to end): 923
Enter a positive number (-999 to end): 45
Enter a positive number (-999 to end): 87
Enter a positive number (-999 to end): -999

The total of your numbers entered is 1090

```
#include < stdio.h >
void main (void)
{
      /* Declare variables. */

      int  num, total = 0;

     /* Prompt user for first number */

     printf ("Enter a positive number (-999 to end): ");
     scanf ("%i", &num);

     /* If user did not enter -999 as first number, continue
         prompting for number until user enters -999,
         keeping track of the total, otherwise, skip loop.  */

      while (num != -999)
      {
            total = total + num;
            printf ("Enter a positive number (-999 to end): ");
            scanf ("%i", &num);

       } /*     end while loop */

       /* Display total to user. */

       printf ("\nThe total of your numbers entered is %i\n", total);

} /*   end main    */
```

Notice the ***format*** of the **while** loop:

```
while (condition)
{
```

```
    statement(s);
}
```

As mentioned previously, the **while** loop will continue repeating until the condition becomes false. Also, like the **for** loop, the body of the **while** loop can contain 1 or more statements. If the body is just a single statement, then the begin and end braces are not needed, however the code should still be indented for readability.

Note: There must be some code, inside the body of the **while** loop, which modifies the loop control variable, so that it will eventually terminate the loop. In our example, variable **num** is the loop control variable, and it is modified each time the user enters a value. If the loop control variable is not modified within the body of the **while** loop, you will have an infinite loop. This is not an issue with the **for** loop, as the loop control variable is incremented or decremented in the for statement itself (the 3rd segment).

The "**sequence**" or "**flow control**" of the while loop is as follows:

1. The loop condition is evaluated.
2. If true, the body of the loop is executed, including the modification of the loop control variable. Go to step 1.
3. If false, the loop is terminated. Go to step 4.
4. Program execution continues with next executable statement following the while loop block.

Note: If the loop condition is false initially, then the body of the **while** loop will never be executed. This was also the case with **for** loops.

I will illustrate below how the same problem can be solved using a **while** loop or a **for** loop. I will not use the above example, as we have not yet covered the "**break**" statement which would be needed if I converted the above **while** loop to a **for** loop. Instead, I will convert the program we reviewed last week, which squared the numbers from 1 to 5.

SQUARING NUMBERS FROM 1 TO 5 USING A WHILE LOOP

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

    /* Variable Declarations. */
    /*  ----------------------------- */

    int    x,  result;
```

```
    /* Calculate and output the squares for numbers between 1 and 5. */
    /* ------------------------------------------------------------------------- */

    x = 1;
    while ( x <= 5 )
    {
        result = x * x;
        printf ("%i squared is %i\n",  x, result);
        x++;                /*  increment variable x     */

    } /*      end while loop */

    /* Output the final greeting. */
    /* ------------------------------- */

    printf ("\nGoodbye!\n");>

} /*   end main   */
```

The output of the above program is the same as the program we reviewed last week using a for loop. That is:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25

Goodbye!
```

Notice that the 3 components of the **for** loop (initial value, final value, increment/decrement) are also present in this **while** loop. Setting the initial value is performed in the statement right before the while loop (**x = 1**), the final value (or test) is performed right in the **while** statement itself (**x <= 5**), and the increment is performed within the body of the **while** loop (**x++**).

Program 4.6 on page 56 is a straightforward program which illustrates the **while** loop. Please review it on, and post questions in the discussion are if you are unclear about any portion of that program.

Programs 4.7 and 4.8 on pages 58 - 60 also contain **while** loops, but their algorithms are somewhat complex. I am not requiring that you review these programs, as I feel you have seen enough examples which illustrate the basic fundamentals of a **while** loop. However, if you are looking for additional examples, both of those programs are interesting and certainly worth looking at.

Next, we discuss the final repetition structure: The "**do**" loop.

Do-bee-do-bee-do...

Sinatra was one of my favorites.....

The Do Loop

The third, and final repetition structure is known as the "**do**" loop. It is similar to the **while** loop in that it physically resembles the **while**, and it is a good loop to choose if you do not know how many iterations of the loop there will be. The main difference however, between the **do** loop and the **while** loop is that the **do** loop performs its test on the loop condition AFTER the body of the loop has been executed at least once. ***So, the do loop will always execute at least once***, the **while** loop (and **for** loop) may not (if the loop condition is false to begin with).

The *format* of the **do** loop is as follows:

```
do
{
    statement(s);

} while (condition);
```

Notice I put the "while" portion of the **do** loop on the same line as the end brace, rather than on the next line, as the textbook illustrates. This is allowed, and preferred. The reason that this is preferred is because, if we use the syntax shown in the textbook, the while portion of the **do** loop actually looks like the *start* of a new **while** loop, with a semicolon at the end. This actually looks quite confusing at first glance, as shown below:

```
do
{
    statement(s);

}
while (condition);
```

The **flow control** for the do loop is as follows:

1. The body of the loop (including modification of the control value) is executed.
2. The loop condition is evaluated.
3. If true, go to step 1.
4. If false, loop is terminated. Go to step 5.
5. Program execution continues with next executable statement following the do loop.

I think it's time for some examples. First of all, before we look at some code, let's go back to my children and their vegetables. (Do you think I need counseling???) In the prior example, they were to take a bite of their vegetables until they were full. Well, what if they were full right away? Then they wouldn't even bother to take a single bite. (Hey, this example is sounding quite familiar!) However, if I used a do loop, then they would have to take a bite first, then test to see if they were full before they decided to stop. Now, that's a loop I can use! :-)

Back to code. Let's take a look at our example from the previous topic where we prompted the user for numbers until the user entered **-999**.

The program then went on to total up the numbers entered. The difference between this program and the program using the while loop, is that the body of the do loop will be entered at least once.

Below is the program from the previous topic, with a simple modification from the while loop to the do loop:

```
#include < stdio.h > void main

(void)
{
     /* Declare variables. */

     int    num, total = 0;
     char c;

    /* Prompt user for first number */

    printf ("Enter a positive number (-999 to end): ");
    scanf ("%i", &num);
    while ( (c = getchar() != '\n') && c != EOF); /* clear input buffer */

    /* Prompt user for numbers until user enters -999,
       keeping track of the total.  */

     do
     {
          total = total + num;
          printf ("Enter a positive number (-999 to end): ");
          scanf ("%i", &num);
          while ( (c = getchar() != '\n') && c != EOF); /* clear input buffer */

      } while ( num != -999 );

      /* Display total to user. */
```

```
    printf ("\nThe total of your numbers entered is %i\n", total);

} /*end main */
```

Now, the problem with the above program is that if the user enters **-999** initially (when first prompted), the **while** loop would have handled it well, as the loop would not have been entered. However, in this case, if the user enters **-999** in the first prompt, the program continues to prompt for at least 1 more value. This is not good. So, we either go back to the **while** loop, or we modify this **do** loop to make it do what we need it to do. I have come up with the following solution:

```
#include < stdio.h >

void main (void)
{
    /* Declare variables. */

    int  num,  total = 0;
    char c;

   /* Prompt user for numbers until user enters -999,
      keeping track of the total. */

    num = 0;
    do
    {
        total = total + num;
        printf ("Enter a positive number (-999 to end): ");
        scanf ("%i", &num);
        while ( (c = getchar() != '\n') && c != EOF); /* clear input buffer */
     } while ( num != -999 );

     /* Display total to user. */

     printf (" \nThe total of your numbers entered is %i \n", total);

} /* end main */
```

In this case, the prompt exists only in the loop, and the loop will exit if user enters **-999** the first time. This is actually a better way to handle this problem, as we do not need to repeat the prompt both before, and inside the loop.

This next example is the "square" example, however it is processed using a **do** loop.

SQUARING NUMBERS FROM 1 TO 5 USING A DO LOOP

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

      /* Variable Declarations. */
      /*  ---------------------------- */

      int     x,  result;

      /* Output initial greeting. */
      /*  ---------------------------- */

      printf ("\nHello, and welcome to the squares program!\n\n);

      /* Calculate and output the squares for numbers between 1 and 5. */
      /* ------------------------------------------------------------------------------- */

      x = 1;
      do
      {
          result = x * x;
          printf ("%i squared is %i\n",  x, result);
          x++;                  /* increment variable x */

      } while ( x <= 5);

      /* Output the final greeting. */
      /* ------------------------------- */

      printf ("\nGoodbye!\n");

} /* end main */
```

The output of the above program is the same as the output of the **for** loop and **while** loop versions of this program. That is:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25

Goodbye!
```

If you want additional practice with do loops, please review program 4.9 on page 61 in your textbook. It is the same program as 4.8. The only difference is that

program 4.8 uses a **while** loop to solve the problem, and program 4.9 uses a **do** loop.

Notice the similarities between the 3 loops discussed, but also be aware of the differences. Traditionally, if we know exactly the number of times a loop is to be performed, use the *'for loop',* if the loop must be performed at least once for an indeterminate number of times, use '*do*', and if the loop is contingent upon a starting condition to be performed at all, use '*while*'.

This ends the discussion of the 3 different repetition structures in C. Is this really the end, or just the end of the beginning, or is there a sequel? Who knows what lurks ahead of us, only the shadow knows....

Note: We will be skipping the section in the textbook that discusses the "**break**" and "**continue**" statements for now. I feel that we are better off discussing these 2 statements next week, where we cover the Selection Structures (if, else, else if, and switch). Although **break** and **continue** statements are used in loops, they cannot be effectively used unless combined with loops *and* selection structures. This will be clearer as we actually start discussing these topics.

That's all for this week's lecture notes. Whew!

No rest for the weary though. It's time to get started on your 2nd programming assignment.

The "if" selection structure.

Recall from Week 5, that we organize our code using the following 3 techniques to ensure that we are following structured programming standards:

- Sequence Structure - code flows in sequence from beginning to end.
- Repetition Structure - **for**, **while**, and **do** loops.
- Selection Structure - **if**, **else**, **else if**, and **switch** statements.

During Weeks 2 and 3 we reviewed programs which used "sequence structures". Weeks 5 and 6 we introduced programs which incorporated "repetition structures", and this week we introduce "selection structures".

As mentioned in Week 5, **selection structure** refers to code which makes choices, or selections, based on certain conditions. A program's ability to "make decisions" based on certain conditions is vital in programming. Types of structures which fall into this category include: **if**, **else**, **else if**, and **switch** structures. We will now introduce the **if** selection structure.

We will be covering two of these selection structures this week (Week 5), and two next week (Week 6) as shown here:

Week 5:

- The "if" selection structure
- The "else" selection structure

Week 6:

- The "else if" selection structure
- The "switch" selection structure

The "if" selection structure.

The logic involved in this type of structure is not unfamiliar to us. We use this type of decision-making logic in our everyday lives. For example: "*If* you eat all your vegetables, you can have dessert." Here we are making a decision (you can have dessert) based on some condition (if you eat all your vegetables). You can infer from this statement, that if you do NOT eat all your vegetables, then you cannot have dessert. :-((I'm really not that mean!)

There are times in our program that we want to execute some statement, or statements, only if a condition is true. In this case, we use the **if** selection structure. The if selection structure has the following format:

```
if (condition)
{
     statement(s);
}
```

Notice the *condition* above in parentheses. This condition will normally contain an comparison between 2 values using one of the 6 relational operators (shown in example below). The parentheses are required. The *statements* should be *indented*, and enclosed in begin and end braces. However, if the body of the **if** structure is just a single statement, the braces are not required. The statements that follows the if are all indented and are executed when the condition is true.

The flow of the above structure is as follows:

1. The condition is tested.
2. If *true*, the statement(s) following the **if** is(are) executed. (Otherwise, they are not.)
3. Program continues with the next executable statement following the **if** structure.

Below is a sample program which uses an **if** selection structure. It prompts the user for a grade, and if the grade is greater than, or equal to 60, then the message "you passed" is output. The program then displays a "goodbye" greeting.

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

     /* Declare variables. */

     int    grade;
     char c;

     /* Prompt user for grade. */

     printf ("Enter grade: ");
     scanf ("%i", &grade);
     while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */


     /* If grade is greater than or equal to 60, display message. */

     if ( grade >= 60 )
         printf ("You passed!\n");
```

```
    /* Display goodbye greeting. */

    printf ("\nGoodbye.\n");

} /*   end main    */
```

If I ran the above program, and the user entered: 88 as the grade, the following would be output:

Enter grade: 88
You passed!

Goodbye.

If I ran the above program, and the user entered 52 as the grade, the following would be output:

Enter grade: 52

Goodbye.

Notice, the body of the **if** statement did not get executed in the second run of the program.

Programming example 6.1 illustrates another **if** statement. This program displays the "absolute value" of a number entered by the user. (In case you've forgotten, the absolute value is a positive version of a number. That is, the absolute value of -50 is 50. The absolute value of 50 is 50.) I have repeated the code below, so to explain the use of the "minus sign" in one of the statements.

PROGRAM 6.1 (with comments added)

```
#inlcude < stdio.h >
void main (void)
{

   /* declare variables. */

   int    number;
   char c;

  /* Prompt user for numbers. */

   printf ("Type in your number: ");
   scanf ("%i", &number);
   while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */
```

```
    /* If number is negative, make it positive. */

    if ( number < 0 )
        number = -number;

    /* Display the absolute value of the number entered. */

    printf ("The absolute value is %i\n", number);

} /*     end main    */
```

I am assuming that you understand the concept of the **if** statement, however you might be confused about the following code:

number = -number;

Well, it is not all that confusing. Basically, what this statement is doing is assigning to variable **number** the value that was in variable **number** but a *negated* version of it. C works from the right side of the assignment operator to the left, and the right side is: **-number**. So, if variable **number** contained the value **-50**, then **-number** would be equal to **--50**, which equals **+50**. Then we take this **+50** and assign it to the variable on the left of the assignment operator, which happens to be variable **number**! Thus, variable **number** ends up containing the negative of what it had, which ultimately makes its value positive. Whoa. What the heck did I just say?!?!? Is it my imagination, or did I confuse things more????

If the value in variable number is already positive (i.e., the user enters a positive number), then the body of the **if** statement is not executed, and the original, positive value is output as the absolute value.

You can see sample output of this program in the gray boxes on pages 72 and 73.

I will not review **program 6.2**, as it is similar to your programming assignment! You can use it as a guide for your assignment. However, I do want to point out one important concept that is discussed in this program, namely: typecasting. **Typecasting** is discussed in the next set of notes.

Type casting -- a very important feature of C.

In order to maintain data integrity, we have to adhere to the strict rule that the data on the left side of the assignment operator should have the same data type as the data on the right side. Sometimes this is not possible. However, we have the ability in C to resolve this problem, using **type casting**.

Let's first make sure we understand the problem. Let's say we want to calculate the average of a 10 salaries which are entered by the user. And we want to display the average with 2 decimal places of accuracy. Well, from this description of the problem, it appears as though we need a variable of type "**float**" to hold the calculated average. Let's pretend that the salaries were entered by the user, one at a time, and a running "**sum**" was being accumulated as each grade was entered. (Refer to program 5.2 on for this type of processing.)

Once all the salaries are entered, and the **sum** is calculated, you we might have a statement such as:

average = sum / 10;

Well, this is not good because let's assume variable **sum** is of type **int**, and the number **10** is also a whole number. The division on the right side of the assignment statement will yield an *integer* result. However, variable **average** is of type **float**, as you recall. Bad, bad programming practice.

What are our choices. Well, we have a few. One solution is, instead of using the number **10** in our calculation, we could use the number **10.0**. This would make the number 10 a floating point number, which will make the result of the calculation of type **float**. This is the best solution, but not good for explaining type casting! So, I will move along to the second solution:

We can **type cast** variable **sum**. With type casting, you *temporarily* modify the data type of a variable to be whatever data type you want. In this case, if we wanted to keep the number **10** as **10,** and not **10.0** (I'm not sure why you would ever want to do that), then our only other solution is to type cast variable **sum** to *temporarily* be type **float**. This would look like:

average = (float) sum / 10;

We now have type cast variable **sum** to be of type **float**. The data type located in parentheses before the variable is the data type we want. In this case, **float**. So, in this statement, variable **sum** is temporarily converted to be type **float**. This causes the data type on the right to be type **float**, and can now be *safely* assigned to the **float** variable on the left side of the assignment operator.

This would be *very* important if the number of salaries were a variable, like **how_many.**

average = (float) sum / how_many;

You can type cast any variable to be any data type you choose. For example, if I had the following variable declarations:

int total;
float num1, num2;

Assume I prompted the user to enter 2 numbers, and stored them in ***float*** variables **num1** and **num2.** If I wanted to add the two numbers together and store them in variable total, I would have to type cast *both* variables num1 and num2, because the only way to get an integer result is to have all variable involved in the calculation to be of type int. So, the expression would look like:

total = (int)num1 + (int)num2;

If the user entered 10.2 and 20.6 as their numbers, the total would be: 30. (10.2 would be come 10 in the type cast, and 20.6 would become 20 in the second type cast. 10 + 20 = 30.) Notice when a float is assigned to an integer variable, or is typecast to an integer, the decimal points are "***truncated"*** (cut off). To effect a numeric round off, you would need to first add .5 to the float variable, then assign to (or typecast to) an integer, as below:

total = (int) (num1 + num2 + .5);

Warning: In C printing a float with 0 decimals using %.0f would *display* the value rounded off, but the actual number is still not rounded!

That's all I have for type casting. The textbook gives some additional information regarding this concept.

We now move on to the next selection structure: "**if else**".

The "if/else" selection structure. Learn it, or ELSE!

The second type of selection structure is discussed here. This is very similar to the **if** selection structure. Like the **if** selection structure, the **if/else** selection structure will perform an action if a condition is true; however, it will go ahead and perform a *different action* if the condition is false.

Again, we do this sort of thing in our every day lives. For instance, I might say: If it is a nice day, I will go for a walk, otherwise (else) I will clean out the garage.

The if/else selection structure has the following format:

```
if (condition)
{
    statement(s);
}
else
{
    statement(s);
}
```

Remember to indent the bodies of both the if, and the else portions. Also, if the body of the **if** is a single statement, then the braces are not needed in that portion. If the body of the **else** is a single statement, then the braces are not needed in that portion.

The flow of the above structure is as follows:

1. The condition is tested.
2. If *true*, the statement(s) following the **if** is(are) executed. Go to step 4.
3. If *false,* the statement(s) following the **else** is(are) executed. Go to step 4.
4. Program continues with the next executable statement following the **if/else** structure.

Below is a modification of our "grade" program which I introduced in the discussion of the **if** selection structure. This program prompts the user for a grade, and if the grade is greater than, or equal to 60, then the message "you passed" is output; *otherwise the message "your failed" is output*. The program then displays a "goodbye" greeting.

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

    /* Declare variables. */
```

```
    int     grade;
    char c;

    /* Prompt user for grade. */

    printf ("Enter grade: ");
    scanf ("%i", &grade);
    while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

    /* Output appropriate message based on grade entered. */

    if ( grade >= 60 )
        printf ("You passed!\n");
    else
        printf ("You failed.\n");

    /* Display goodbye greeting. */

    printf ("\nGoodbye.\n");

} /* end main */
```

If I ran the above program, and the user entered: 88 as the grade, the following would be output:

Enter grade: **88**
You passed!

Goodbye.

If I ran the above program, and the user entered 52 as the grade, the following would be output:

Enter grade: **52**
You failed.

Goodbye.

Notice, the body of the **else** did not get executed in the first run of the program, and the body of the **if** statement did not get executed in the second run. Only the body of the **if** *or* the body of the **else** will be executed, not both.

Program 6.4 illustrates the if/else selection structure. It is a program that determines if a number entered by the user is odd or even. I will repeat it below, as I think it is a good example. I have added some comments for clarity. (Readability folks!)

Think for a moment (you can still think, can't you?!?!). The way we determine if a number is odd or even is if it is divisible by 2, with no remainder. If so, the number is even. Otherwise it is odd. With that in mind, we can continue with some code. (And, lo and behold, we just created a quick algorithm!) :-)

Program 6.4 (with some comments added)

```
#include < stdio.h >
void main (void)
{
    /* Declare variables. */

    int     number_to_test,  remainder;
    char c;

    /* Prompt the user for a number */

    printf ("Enter your number to be tested: ");
    scanf ("%i", &number_to_test);
    while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

    /* Determine the remainder by using the "mod" function. */

    remainder = number_to_test % 2;

    /* If the number "mod" 2 is zero, then number is even. Otherwise it is
odd. */

    if ( remainder == 0 )
        printf ("The number is even.\n");
    else
        printf ("The number is odd.\n);

} /* end main */
```

For some sample output to the above program, please refer to the textbook.

That is about all I have for the **if/else** selection structure. The next topic to be discussed this week is something called: Compound Relational Tests.

Different, yet related topics.

Compound Relational Tests

We have used *single* relational tests in our **loops** and in our **if** statements. They are the conditions that are tested using one of the 6 relational operators discussed during Week 5.

For example (using an **if** statement): **if (remainder == 0)**

Another example (using a **for** loop): **for (x = 1; x <= 5; x++)**

These are examples of single relational tests, because we are using a single relational operator in each test (**==** in the first example, and **<=** in the second example). Sometimes, we need to test more than 1 condition at a time. We do this using "**compound relational tests**". That is, more than 1 relational operator on a line. An example of this in "human language" might be something like:

I might tell my children: "If you eat all of your vegetables, AND if you drink all of your milk then you can have dessert." In that case, the child cannot have dessert unless *both* of the conditions are met. (Pretty strict huh?!?!). Okay, another option I have is to lighten up a bit and tell my children: If you eat all of your vegetables, OR if you drink all of your milk, then you can have dessert." In this case, the child can have dessert if *either* the veggies are eaten, *or* the milk is drunk. Notice here I needed to put some "connector" between the two conditions in each example. This connector (AND / OR) are known as "Logic Operators" in C, and they are discussed below.

Logic Operators: && and ||

Logic operators are used to "connect" compound relational tests together to make a single condition. The 2 logic operators discussed here are: AND and OR. (There is a third logic operator, known as NOT, but I will defer that until a later topic.)

The **AND** logic operator is represented by 2 ampersand signs together, with no spaces between them. It looks like: **&&**.

The **OR** logic operator is represented by 2 vertical lines together (located above the "Enter" key, it's *shift*-'\'), with no spaces between them. It looks like: **||**.

When you combine 2 relational tests using the AND logic operator, what you are saying is that BOTH conditions must be true for the statement to be true. That is, if the following table represents 2 conditions, you can see how the resulting combined condition is only true if *both* condition 1 and condition 2 are true. This

is known as the **logical AND truth table**. "T" represents "TRUE" and "F" represents "FALSE".

Condition 1 (Ex: Eat all veggies)	**Condition 2** (Ex: Drink all milk)	**Cond1 && Cond2**
F	F	F (no dessert)
F	T	F (no dessert)
T	F	F (no dessert)
T	T	T (can have dessert!)

When you combine 2 relational tests using the OR logic operator, what you are saying is that EITHER condition can be true for the statement to be true. That is, if the following table represents 2 conditions, you can see how the resulting combined condition is true if *either* condition 1 *or* condition 2 is true. This is known as the **logical OR truth table**. Again, "T" represents "TRUE" and "F" represents "FALSE".

Condition 1 (Ex: Eat all veggies)	**Condition 2** (Ex: Drink all milk)	**Cond1 || Cond2**
F	F	F (no dessert)
F	T	T (can have dessert!)
T	F	T (can have dessert!)
T	T	T (can have dessert!)

Now, let's apply this wealth of knowledge using some C code. Assume you want to prompt the user for an amount between 10 and 50. We want to test to be sure that the number entered is in fact within this range. (Finally, some user input validation!) The code might look something like:

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

    /* Declare variables. */

    int     grade;
    char c;

   /* Prompt user for grade between 10 and 50. */

    printf ("Enter grade (10 - 50): ");
    scanf ("%i", &grade);
    while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

     /* If grade is out of range (less than 10, OR greater than 50),
        output error message, otherwise, output grade entered. */
```

```
    if (grade < 10 || grade > 50)
        printf ("Invalid grade entered.\n");
    else
        printf ("You entered %i.\n", grade);

} /* end main */
```

The above code would produce the following results:

Enter grade (10 - 50): **48**
You entered 48.

Another sample run might look like:

Enter grade (10 - 50): **125**
Invalid grade entered.

Notice in the **if** statement above that if the grade entered is less than 10 *OR* greater than 50, then we would have an error. If either of these conditions are true, then the entire test is true, and the user would see an error. For example, when the user typed in 48 above, 48 is NOT less than 10, *and* it is NOT greater than 50, so the test fails. Both are false, so the entire test is false. In the second example, when the user typed in 125, 125 is NOT less than 10, but it IS greater than 50. So, since the second condition is TRUE , the entire test is TRUE (when using the logical OR operator).

One **very important point** I want to make here is that there are many ways to perform the same test, using different logic and relational operators. For example, the above code would have worked if my **if** statement contained the "AND" logic operator as follows:

```
if (grade >=  10 && grade <= 50)
    printf ("You entered %i.\n", grade);
else
    printf ("Invalid grade entered.\n");
```

Notice, in this code segment, the test for "valid grade" is performed first, whereas in the previous program, we were testing for "invalid grade" first. Here, we are checking that the grade entered is greater than or equal to 10 AND the grade is less than or equal to 50. Only when BOTH of these conditions are true, is the grade valid. So, the number 48, for example, is valid because it is greater than 10 AND it is less than 50. The number 125 however, is invalid because it is NOT less than or equal to 50. So, since with the logic AND, BOTH conditions must be true for the statement to be true, we fail this test when the user enters 125.

Program 6.5 determines if a "year" entered by the user is a "leap year". It seems somewhat complex, but if you step through it, it is pretty straight-forward. Don't stress over the algorithm too much, I want your focus to be the logic operators used in the if/else statement towards the end of the program. Also, please remember to read through the "QUICK TIPS" in this chapter. They are quite useful.

We will see a lot more of this sort of processing as the course progresses. (I knew you'd be psyched when I told you that!) I'm looking forward to it too!!! :-)

In the next topic, we discuss Flags and Traps. Say what?!

Flags & Traps

You thought you knew what a ***flag*** was, didn't you? Well, we are talking computers here, not Patriotism. A flag in C is typically a variable, that we declare as an integer (or maybe a character), but will contain only the values 0 and 1. In C, 0 (zero) represents FALSE , and 1 (one) represents TRUE. Some programming languages (such as Pascal) have a data type reserved for this type of variable. It is normally referred to as a "Boolean" variable. Boolean variables in Pascal can *only* contain the values 0 and 1. Since C ***does not**** have a data type of Boolean, we simply use a data type of int, and go from there. *In our book there is a type Boolean, but that is only ANSI 99 version of C, so most compilers we use will not recognize it. (Please do no use it!)

Flags are used to indicate when it is time to stop processing a loop. Normally we use flags in **while** and **do** loops only. We initialize the flag variable to false (0), and the loop continues until the flag variable becomes true (1). In the body of the loop, some condition takes place which will eventually cause that Boolean variable to become true (1). Some programmers prefer to use flags instead of the "break" or "continue" statement, as they can sometimes be used interchangeably.

For example, the following program will prompt the user to enter a number to be squared (a number multiplied by itself), and then display the squared value of the number entered. The loop continues up to *until the user enters the number 999 to terminate data entry*. If the user enters the number 999, the loop terminates. Notice the flag "get_nums" being used to control the while loop. It will remain TRUE (1) until the user decides to terminate, at which point it will become FALSE (0).

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

    /* declare variables. */

    int     num, squared_num, i, get_nums;
    char c;

    /* Get as many values as user chooses.*/

    get_nums = 1;
    while (get_nums == 1)
    {
        printf ("Enter a number to square (999 to quit): ");
        scanf ("%i", &num);
        while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */
```

```
        /* If user enters 999, then reset flag.*/

        if (num == 999)
            get_nums = 0;

        /* Otherwise, calculate and output squared value of number entered.*/

       else
       {
            squared_num = num * num;
            printf ("%i squared is %i\n", num, squared_num);

       }  /* end if-else    */

    }  /* end while loop*/

    /* Display greeting.*/

    printf ("\nGoodbye!\n");

    getchar();

} /* end main*/
```

A sample program execution is shown below:

```
Enter a number to square (999 to quit): 5
5 squared is 25
Enter a number to square (999 to quit): 8
8 squared is 64
Enter a number to square (999 to quit): 999

Goodbye!
```

In the above example, the user terminated data entry by entering 999. The loop terminated as a result of the program setting the "flag" (get_nums) to zero. It was no longer 1, so the while loop test failed.

Some programmers do not like to use the data input line to elicit the response from the user, but may use another method as illustrated below:

```
#include <stdio.h>
void main (void)
{
  /*declare variables.*/
```

```
    int    num, squared_num;
    char   get_nums, c;

    /*  Get as many values as user chooses.*/

    get_nums = 'y';
    while (get_nums == 'y')
    {
        printf ("\n\nEnter a number to square: ");
        scanf ("%i", &num);
        while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

        /*   calculate and output squared value of number entered.*/

        squared_num = num * num;
        printf ("%i squared is %i\n", num, squared_num);

        /*  If user enters n, then  flag is false.  This is another 'trap'
example*/
        do
        {
            printf("\n Try again? ( enter y or n): ");
            scanf("%c", &get_nums);
            while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */
        } while (get_nums != 'y' && get_nums != 'n');

    }  /*  end while loop*/

    /*  Display greeting.*/

    printf ("\nGoodbye!\n");

   getchar();

} /*  end main*/
```

A sample program execution is shown below:

Enter a number to square : 5
5 squared is 25

Try again? (enter y or n): y

Enter a number to square: 8
8 squared is 64

Try again? (enter y or n): **0**

Try again? (enter y or n): **n**

Goodbye!

In the above example, the user terminated data entry by entering **n**. The loop terminated as a result of the program "flag" (get_nums) being zero. It was no longer **y**, so the while loop test failed. It re-prompted when the user responded with a 0 which did not meet the trap requiremets for a response.

Another example of a *'trap' loop*:

```
do
{
        printf("Please enter an integer from 10 to 20: ");

        scanf("%i", &some_number);
        while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

        if ( some_number < 10 || some_number > 20)
                printf("Sorry invalid entry, try again...\n\n");

  } while (some_number < 10 || some_number > 20);
```

Notice that the user is 'trapped' into responding properly to the data entry. The do loop above is the best representation of a simple trapping loop.

You should ***always*** use a trap loop for user entry to validate the data as much as you can. Note that the 'if' condition and the 'while' condition are exactly the same – copy and paste could be used so that they are the same.

After the trap the program will continue and you are more confident that the data is correct.

FYI: This next part is a bit advanced, and might seem confusing at first, but I feel compelled to mention it at this time. If you do not understand it, don't worry, it is discussed again in the C Programming Part II course.

In C, variables can be tested for TRUE or FALSE (0 or NOT 0) without using the relational operators == or !=. For example, in the above example, we tested variable **get_nums**in the while loop as follows:

while (get_nums == 1).

There is an abbreviation for this type of test. That is, if you are testing for any integer value for a non-zero value, you can simply test as follows:

while (get_nums)

If you wanted to test if the value of a variable was *false,* then your test would be as follows:

while (variable == 0)

and your abbreviation could be:

while (! variable)

The "**!**"is the logical NOT operator, and it negates the value of a variable for testing. So, if the variable was actually 0 (false), and I tested for ! 0 (NOT 0) or NOT false, then I am testing for true, and the test would pass if the variable was in fact false. (NOT false is the same as true). Yikes!

The following 3 tests all do the same thing. That is, they test if the value of a variable is "TRUE".

- if (variable == 1)
- if (variable != 0)
- if (variable)

The following 2 tests do the same thing. That is, they test if the value of a variable is "FALSE".

- if (variable == 0)
- if (! variable)

Now, if that didn't cause a brain cramp, I don't know what will!! :-)

Chapter 5 has more information regarding flags. Also, for another sample program using flags, please refer to program 5.10 , however it contains _Bool type which we cannot use. Here is another version of 5.10 without _Bool type:

/* Program 5.10B to generate a table of prime numbers */

```
#include <stdio.h>

main()
{

   int p, is_prime, d;
```

```
   for ( p = 2; p<=50; p++)
    {

        is_prime = 1;

        for ( d = 2; d < p; d++)

                if ( p % d == 0 )
                    is_prime = 0;

                if (is_prime != 0)
                    printf("%i  ",p);

     }/*   end for loop  */

 printf("\n");

 getchar();

}/*    end program */
```

Program 5.8 in our text book uses a character variable named ***operator.*** This is **not** a good idea, and is probably an error in most compilers, because that word is a *reserved* word in C++ and in *some* versions of C. (For example you cannot use *printf* as a variable name!)

That's all for this week.

If you haven't already done so, be sure to finish your 2nd programming assignment!

if ... "else if" ... "else if" ... "else if" ... "else if" ... else

Recall from Week 5, we covered two of the four selection structures:

- The "if" selection structure
- The "else" selection structure

This week, we will cover the other two selection structures:

- The "else if" selection structure
- The "switch" selection structure

The "else if" selection structure.

The third type of selection structure is discussed here. Like the **if** selection structure, the **if/else if** selection structure will perform an action if a condition is true; however, if the condition is false, a *different condition* is then tested. This logic can keep going, until as many conditions as necessary are tested.

Again, we do this sort of thing in our every day lives. For instance, I might say: If it's really hot today, I will go to the beach. Otherwise if (else if) it is warm today, I will take a walk. Otherwise (else), I will clean out the garage.

The above example follows the following format:

```
if (it is really hot today)
    I will go to beach
else if (it is warm today)
    I will take a walk
else
    I will clean out the garage.
```

Note that you do not have to have an "**else**" to terminate the **if / else if** statement. For example, a valid **if / else if** structure can have the following format:

```
if (condition)
{
    statement(s);
}
else if (condition 2)
{
    statement(s);
}
else if (condition 3)
{
```

```
        statement(s);
    }
    etc. (You could have more conditions here)
```

Remember to ***indent*** the bodies of the **if**, and the **else if** portions. Also, if the body of the **if** is a single statement, then the braces are not needed in that portion. If the body of the **else if** is a single statement, then the braces are not needed in that portion.

The flow of the above structure is as follows:

1. The 1st condition is tested.
2. If *true*, the statement(s) following the **if** is(are) executed. Go to step 6.
3. If *false,* the next condition is tested.
4. If *true,* the statement(s) following the **else if** is(are) executed. Go to step 6.
5. Go to step 3.
6. Program continues with the next executable statement following the **if/else if** structure.

Below is a modification of our "grade" program which I introduced in the discussion of the **if** and **if/else** selection structures (last week). This program prompts the user for a grade, and if the grade is greater than, or equal to 90, then the message "Excellent" is output. Otherwise, the grade is then tested to see if it is greater than or equal to 80. If so, the message "Good job" is output. If not, the grade is tested to see if it is greater than or equal to 70. If so, the message "Satisfactory" is output. If not, the grade is tested to see if it is greater than or equal to 60. If so, the message "Poor" is output. If not, the message "You failed" is output. The program then displays a "goodbye" greeting.

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

    /* Declare variables. */

    int grade;

    /* Prompt user for grade. */

    printf ("Enter grade: ");
    scanf ("%i", &grade);

    /* Output appropriate message based on grade entered. */

    if ( grade >= 90 )
        printf ("Excellent!\n");
```

```
    else if ( grade >= 80 )
        printf ("Good job!\n");
    else if ( grade >= 70)
        printf ("Satisfactory.\n");
    else if ( grade >= 60 )
        printf ("Poor job.\n");
    else
        printf ("You failed.\n");


    /* Display goodbye greeting. */

    printf (" \nGoodbye. \n");

    getchar();

} /* end main*/
```

If I ran the above program, and the user entered: 88 as the grade, the following would be output:

Enter grade: 88
Good job!

Goodbye.

If I ran the above program, and the user entered 52 as the grade, the following would be output:

Enter grade: 52
You failed.

Goodbye.

Notice, once the condition is met, the appropriate statements are executed and program execution continues with the next executable statement after the selection structure.

An important factor which must be considered when using the **else if** structure is *order.* Order plays a very important role when setting up these sometimes complex structures. For example, using the above program, if I reversed the order of the tests on the grade, I would clearly have a logic error. Look at the program segment below (extracted from the above program), and try to figure out what is logically wrong with it. (I realize that some of you are having difficulty concentrating right now, but try not scroll down to the answer until you have figured it out!!!!)

```
if ( grade >= 60 )
   printf ("Poor job.\n");
else if ( grade >= 70 )
   printf ("Satisfactory.\n");
else if ( grade >= 80 )
   printf ("Good job!\n");
else if ( grade >= 90 )
   printf ("Excellent!\n");
else
   printf ("You failed.\n");
```

Now, if I ran the program, and the user entered: 88 as the grade, the following would be output:

Enter grade: **88**
Poor job.

It is because any grade greater than or equal to 60 passes the first test. Clearly a problem.

With the following variation of the same program segment, *order is not important*, as we are actually testing for a specific *range* of values. The program segment below performs a compound relational test for each condition:

```
 if ( grade >= 60 &&  grade < 70 )
   printf ("Poor job.\n");
else if ( grade >= 70 && grade < 80 )
   printf ("Satisfactory.\n");
else if ( grade >= 80 && grade < 90 )
   printf ("Good job!\n");
else if ( grade >= 90 )
   printf ("Excellent!\n");
else
   printf ("You failed.\n");
```

Now, if I ran the program, and the user entered: 88 as the grade, the following (desired results) would be output:

Enter grade: **88**
Good job!

The grade entered by the user (88) is greater than or equal to 80 AND it is less than 90, so it passes the third condition above.

Please review Programs 5.6, 5.7, 5.8 and 5.8a in text.

If you have any questions regarding programs 5.6, 5.7 and 5.8, please post them to this week's discussion board.

if ... "else if" ... "else if" ... "else if" ... "else if" ... else

Recall from Week 5, we covered two of the four selection structures:

- The "if" selection structure
- The "else" selection structure

This week, we will cover the other two selection structures:

- The "else if" selection structure
- The "switch" selection structure

The "else if" selection structure.

The third type of selection structure is discussed here. Like the **if** selection structure, the **if/else if** selection structure will perform an action if a condition is true; however, if the condition is false, a *different condition* is then tested. This logic can keep going, until as many conditions as necessary are tested.

Again, we do this sort of thing in our every day lives. For instance, I might say: If it's really hot today, I will go to the beach. Otherwise if (else if) it is warm today, I will take a walk. Otherwise (else), I will clean out the garage.

The above example follows the following format:

```
if (it is really hot today)
    I will go to beach
else if (it is warm today)
    I will take a walk
else
    I will clean out the garage.
```

Note that you do not have to have an "**else**" to terminate the **if / else if** statement. For example, a valid **if / else if** structure can have the following format:

```
if (condition)
{
    statement(s);
}
else if (condition 2)
{
    statement(s);
}
else if (condition 3)
{
```

```
        statement(s);
    }
    etc. (You could have more conditions here)
```

Remember to ***indent*** the bodies of the **if**, and the **else if** portions. Also, if the body of the **if** is a single statement, then the braces are not needed in that portion. If the body of the **else if** is a single statement, then the braces are not needed in that portion.

The flow of the above structure is as follows:

1. The 1st condition is tested.
2. If *true*, the statement(s) following the **if** is(are) executed. Go to step 6.
3. If *false,* the next condition is tested.
4. If *true,* the statement(s) following the **else if** is(are) executed. Go to step 6.
5. Go to step 3.
6. Program continues with the next executable statement following the **if/else if** structure.

Below is a modification of our "grade" program which I introduced in the discussion of the **if** and **if/else** selection structures (last week). This program prompts the user for a grade, and if the grade is greater than, or equal to 90, then the message "Excellent" is output. Otherwise, the grade is then tested to see if it is greater than or equal to 80. If so, the message "Good job" is output. If not, the grade is tested to see if it is greater than or equal to 70. If so, the message "Satisfactory" is output. If not, the grade is tested to see if it is greater than or equal to 60. If so, the message "Poor" is output. If not, the message "You failed" is output. The program then displays a "goodbye" greeting.

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

    /* Declare variables. */

    int grade;

    /* Prompt user for grade. */

    printf ("Enter grade: ");
    scanf ("%i", &grade);

    /* Output appropriate message based on grade entered. */

    if ( grade >= 90 )
        printf ("Excellent!\n");
```

```
    else if ( grade >= 80 )
        printf ("Good job!\n");
    else if ( grade >= 70)
        printf ("Satisfactory.\n");
    else if ( grade >= 60 )
        printf ("Poor job.\n");
    else
        printf ("You failed.\n");


    /* Display goodbye greeting. */

    printf (" \nGoodbye. \n");

    getchar();

} /* end main*/
```

If I ran the above program, and the user entered: 88 as the grade, the following would be output:

Enter grade: 88
Good job!

Goodbye.

If I ran the above program, and the user entered 52 as the grade, the following would be output:

Enter grade: 52
You failed.

Goodbye.

Notice, once the condition is met, the appropriate statements are executed and program execution continues with the next executable statement after the selection structure.

An important factor which must be considered when using the **else if** structure is *order.* Order plays a very important role when setting up these sometimes complex structures. For example, using the above program, if I reversed the order of the tests on the grade, I would clearly have a logic error. Look at the program segment below (extracted from the above program), and try to figure out what is logically wrong with it. (I realize that some of you are having difficulty concentrating right now, but try not scroll down to the answer until you have figured it out!!!!)

```
if ( grade >= 60 )
   printf ("Poor job.\n");
else if ( grade >= 70 )
   printf ("Satisfactory.\n");
else if ( grade >= 80 )
   printf ("Good job!\n");
else if ( grade >= 90 )
   printf ("Excellent!\n");
else
   printf ("You failed.\n");
```

Now, if I ran the program, and the user entered: 88 as the grade, the following would be output:

Enter grade: **88**
Poor job.

It is because any grade greater than or equal to 60 passes the first test. Clearly a problem.

With the following variation of the same program segment, *order is not important*, as we are actually testing for a specific *range* of values. The program segment below performs a compound relational test for each condition:

```
 if ( grade >= 60 &&  grade < 70 )
   printf ("Poor job.\n");
else if ( grade >= 70 && grade < 80 )
   printf ("Satisfactory.\n");
else if ( grade >= 80 && grade < 90 )
   printf ("Good job!\n");
else if ( grade >= 90 )
   printf ("Excellent!\n");
else
   printf ("You failed.\n");
```

Now, if I ran the program, and the user entered: 88 as the grade, the following (desired results) would be output:

Enter grade: **88**
Good job!

The grade entered by the user (88) is greater than or equal to 80 AND it is less than 90, so it passes the third condition above.

Please review Programs 5.6, 5.7, 5.8 and 5.8a in text.

If you have any questions regarding programs 5.6, 5.7 and 5.8, please post them to this week's discussion board.

"break" and "continue": Two very useful commands in C.

Note: These two statements are actually introduced in the textbook in chapter 5. I feel however, that since these statements are only used along with a selection structure (typically within the body of a loop) that it didn't make sense to discuss them before we finished discussing some of the selection structures. So... here we are. We are ready to introduce them, and we will discuss them one at a time.

The "break" statement

Sometimes, while executing the body of a loop (**for**, **do**, **while** -- remember those?!?!?), we need to exit the loop as soon as a certain condition occurs (before the originally planned termination). This typically occurs when some error is detected within the body of the loop. The **break** statement causes the program to *immediately exit* from within the body of the loop that it is executing. If the **break** statement is executed, subsequent statements in the loop are skipped. Execution continues with whatever statement follows the loop. If a **break** is used within a nested loop, only the innermost loop in which the **break** is executed, is terminated.

Some common uses for a break statement:

1. To escape early from a loop because of some error condition. (Discussed below).
2. To skip the remainder of a switch structure. (Discussed in next topic).

Below is a simple example illustrating the **break** statement. It is not a very practical program, but it is a good one for explaining the concept. Once we master this example, we will move on to a more practical example.

The program below uses a **for** loop, which would normally run from 1 to 5, and output the numbers 1 to 5. However, I have a condition within the loop checking if the loop control variable is 3. If so, the loop "**break**s", and program execution of that loop terminates right then and there. Program execution continues with the next executable statement *after* the loop, which is the goodbye greeting.

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

    /* declare variable. */
    int count;

   /* Loop from 1 to 5, but break once counter reaches 3 (don't ask why!!!).
*/
```

```
    for (count = 1;  count <= 5;  count++)
    {
        if (count == 3)
            break;

        /* Display counter value. */

        printf ("Counter = %i\n", count);

    } /* end for loop*/


    /* Output greeting. */

    printf ("Goodbye!\n");

    getchar();

} /* end main*/
```

The output of the above program is:

```
Counter = 1
Counter = 2
Goodbye!
```

Note: There is no need for an **else** statement following the **if** condition in the above program. Basically, the rule of thumb is this: There is no need for an **else** statement following a **break** or a **continue** statement. If you are curious as to why this is, just think about the logic for a while, and it should come to you. For example, I could have used the following for loop in the above program:

```
    for (count = 1;  count <= 5;  count++)
    {
        if (count == 3)
            break;
        else
            printf ("Counter = %i\n", count);

    } /* end for*/
```

However, the **else** was not necessary. The standard is to NOT use an **else** following a **break** or **continue** statement. So, this example violates that particular standard.

The following program is a more practical example of the use of a **break** statement. It will prompt the user to enter a number to be squared (a number multiplied by itself), and then display the squared value of the number entered. The loop continues up to 10 iterations, *OR until the user enters the number 999 to terminate data entry*. If the user enters the number 999 OR if the user has entered 10 values, the loop terminates.

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

    /* declare variables. */
    int   num, squared_num, i;

    /* Get up to 10 values. */

    for (i = 1;  i <= 10;  i++)
    {
        printf ("Enter a number to square (999 to quit): ");
        scanf ("%i", &num);

        /* If user enters 999, then terminate loop. */

        if (num == 999)
            break;

       /* Calculate and output sqared value of number entered. */

        squared_num = num * num;
        printf ("%i squared is %i\n", num, squared_num);
    }  /* end for loop*/

    /* Display greeting. */

    printf ("\nGoodbye!\n");

    getchar();

} /* end main*/
```

A sample program execution is shown below:

```
Enter a number to square (999 to quit): 5
5 squared is 25
Enter a number to square (999 to quit): 8
```

8 squared is 64
Enter a number to square (999 to quit): 999

Goodbye!

You see in the above example, the user terminated early from the loop by entering 999.

That is all I have for the **break** statement. Next we will discuss the **continue** statement.

The "continue" statement

The continue statement is similar to the break statement, in that it causes a modification to the original plan of the loop, however, it does not cause the loop to terminate. Instead, it causes any statements following the word continue to be skipped, but allows the loop to continue with the next iteration.

A common use for the continue statement is to discard an invalid data value that has been entered by the user, and re-prompt for additional, valid data.

The best way to explain the continue statement is by example. Below is a simple example illustrating the **continue** statement. Again, it is not a very practical program, but it is a good one for explaining the concept.

The program below uses a **for** loop, which would normally run from 1 to 5, and output the numbers 1 to 5. However, I have a condition within the loop checking if the loop control variable is 3. If so, the loop "**continue**s", and program execution of that loop will continue with the next iteration of the loop.

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

    /* declare variable. */
    int count;

   /* Loop from 1 to 5, but break once counter reaches 3 (don't ask why!!!).
*/

    for (count = 1;  count <= 5;  count++)
    {
        if (count == 3)
            continue;
```

```
        /* Display counter value. */

        printf ("Counter = %i\n", count);

    } /* end for loop*/


    /* Output greeting. */

    printf ("Goodbye!\n");

    getchar();


} /* end main*/
```

The output of the above program is:

```
Counter = 1
Counter = 2
Counter = 4
Counter = 5
Goodbye!
```

Notice that only value "3" was skipped. The remaining values were output, as the loop continued, allowing it to process values 4 and 5.

The next example is a more practical one. It uses a **while** loop to prompt the user to enter 3 grades between 0 and 100. It then determines the highest of the 3 grades. It tests, however that the user in facts enters a number between 0 and 100. If not, the loop will "continue" to prompt for a valid grade.

```
#include <stdio.h>

void main (void)

{

    /* Declare variables. */

    int     grade,  highest = 0,  grade_counter = 1;
    char c;

    /* Prompt user for 3 grades. */
```

```
    while ( grade_counter <= 3 )
    {
        printf ("Enter grade (0 - 100): ");
        scanf ("%i", &grade);
        while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

        /* Ensure grade is between 0 and 100. If not, re-prompt. */

        if (grade < 0 || grade > 100)
       {
            printf ("*** Invalid Grade Entered ***.\n");
            continue;
       }

       /* Determine if this grade is in fact the highest grade. */

       if ( grade > highest )
            highest = grade;

       /* Increment grade counter. */

       grade_counter++;

    } /* end while loop*/

   /* Output the highest grade entered. */

    printf ("\nThe highest grade entered is %i.\n", highest);

    getchar();

} /* end main*/
```

Notice that the **grade_counter** (loop control variable) is only incremented if the grade entered is valid. This ensures us that we will in fact obtain the 3 valid grades that we are looking for.

A sample execution of the above program might be:

```
Enter grade (0 - 100): 80
Enter grade (0 - 100): 102
*** Invalid Grade Entered. ***
Enter grade (0 - 100): -20
*** Invalid Grade Entered. ***
Enter grade (0 - 100): 95
Enter grade (0 - 100): 78
```

The highest grade entered is 95.

That will do it for the **break** and **continue** statements. If you have any questions regarding these 2 statements, please post them in the discussion are for this week.

Hey. Maybe this is an omen. Perhaps we should take a "break" and then "continue" after a nice cup of coffee! (and a bag of chips!!!) :-)

"break" and "continue": Two very useful commands in C.

Note: These two statements are actually introduced in the textbook in chapter 5. I feel however, that since these statements are only used along with a selection structure (typically within the body of a loop) that it didn't make sense to discuss them before we finished discussing some of the selection structures. So... here we are. We are ready to introduce them, and we will discuss them one at a time.

The "break" statement

Sometimes, while executing the body of a loop (**for**, **do**, **while** -- remember those?!?!?), we need to exit the loop as soon as a certain condition occurs (before the originally planned termination). This typically occurs when some error is detected within the body of the loop. The **break** statement causes the program to *immediately exit* from within the body of the loop that it is executing. If the **break** statement is executed, subsequent statements in the loop are skipped. Execution continues with whatever statement follows the loop. If a **break** is used within a nested loop, only the innermost loop in which the **break** is executed, is terminated.

Some common uses for a break statement:

1. To escape early from a loop because of some error condition. (Discussed below).
2. To skip the remainder of a switch structure. (Discussed in next topic).

Below is a simple example illustrating the **break** statement. It is not a very practical program, but it is a good one for explaining the concept. Once we master this example, we will move on to a more practical example.

The program below uses a **for** loop, which would normally run from 1 to 5, and output the numbers 1 to 5. However, I have a condition within the loop checking if the loop control variable is 3. If so, the loop "**break**s", and program execution of that loop terminates right then and there. Program execution continues with the next executable statement *after* the loop, which is the goodbye greeting.

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

    /* declare variable. */
    int count;

   /* Loop from 1 to 5, but break once counter reaches 3 (don't ask why!!!).
*/
```

```
    for (count = 1;  count <= 5;  count++)
    {
        if (count == 3)
            break;

        /* Display counter value. */

        printf ("Counter = %i\n", count);

    } /* end for loop*/


    /* Output greeting. */

    printf ("Goodbye!\n");

    getchar();

} /* end main*/
```

The output of the above program is:

```
Counter = 1
Counter = 2
Goodbye!
```

Note: There is no need for an **else** statement following the **if** condition in the above program. Basically, the rule of thumb is this: There is no need for an **else** statement following a **break** or a **continue** statement. If you are curious as to why this is, just think about the logic for a while, and it should come to you. For example, I could have used the following for loop in the above program:

```
    for (count = 1;  count <= 5;  count++)
    {
        if (count == 3)
            break;
        else
            printf ("Counter = %i\n", count);

    } /* end for*/
```

However, the **else** was not necessary. The standard is to NOT use an **else** following a **break** or **continue** statement. So, this example violates that particular standard.

The following program is a more practical example of the use of a **break** statement. It will prompt the user to enter a number to be squared (a number multiplied by itself), and then display the squared value of the number entered. The loop continues up to 10 iterations, *OR until the user enters the number 999 to terminate data entry*. If the user enters the number 999 OR if the user has entered 10 values, the loop terminates.

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

    /* declare variables. */
    int   num, squared_num, i;

    /* Get up to 10 values. */

    for (i = 1;  i <= 10;  i++)
    {
        printf ("Enter a number to square (999 to quit): ");
        scanf ("%i", &num);

        /* If user enters 999, then terminate loop. */

        if (num == 999)
            break;

       /* Calculate and output sqared value of number entered. */

        squared_num = num * num;
        printf ("%i squared is %i\n", num, squared_num);
    }  /* end for loop*/

    /* Display greeting. */

    printf ("\nGoodbye!\n");

    getchar();

} /* end main*/
```

A sample program execution is shown below:

```
Enter a number to square (999 to quit): 5
5 squared is 25
Enter a number to square (999 to quit): 8
```

8 squared is 64
Enter a number to square (999 to quit): 999

Goodbye!

You see in the above example, the user terminated early from the loop by entering 999.

That is all I have for the **break** statement. Next we will discuss the **continue** statement.

The "continue" statement

The continue statement is similar to the break statement, in that it causes a modification to the original plan of the loop, however, it does not cause the loop to terminate. Instead, it causes any statements following the word continue to be skipped, but allows the loop to continue with the next iteration.

A common use for the continue statement is to discard an invalid data value that has been entered by the user, and re-prompt for additional, valid data.

The best way to explain the continue statement is by example. Below is a simple example illustrating the **continue** statement. Again, it is not a very practical program, but it is a good one for explaining the concept.

The program below uses a **for** loop, which would normally run from 1 to 5, and output the numbers 1 to 5. However, I have a condition within the loop checking if the loop control variable is 3. If so, the loop "**continue**s", and program execution of that loop will continue with the next iteration of the loop.

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

    /* declare variable. */
    int count;

   /* Loop from 1 to 5, but break once counter reaches 3 (don't ask why!!!).
*/

    for (count = 1;  count <= 5;  count++)
    {
        if (count == 3)
            continue;
```

```
        /* Display counter value. */

        printf ("Counter = %i\n", count);

    } /* end for loop*/


    /* Output greeting. */

    printf ("Goodbye!\n");

    getchar();


} /* end main*/
```

The output of the above program is:

```
Counter = 1
Counter = 2
Counter = 4
Counter = 5
Goodbye!
```

Notice that only value "3" was skipped. The remaining values were output, as the loop continued, allowing it to process values 4 and 5.

The next example is a more practical one. It uses a **while** loop to prompt the user to enter 3 grades between 0 and 100. It then determines the highest of the 3 grades. It tests, however that the user in facts enters a number between 0 and 100. If not, the loop will "continue" to prompt for a valid grade.

```
#include <stdio.h>

void main (void)

{

    /* Declare variables. */

    int     grade,  highest = 0,  grade_counter = 1;
    char c;

    /* Prompt user for 3 grades. */
```

```
    while ( grade_counter <= 3 )
    {
        printf ("Enter grade (0 - 100): ");
        scanf ("%i", &grade);
        while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

        /* Ensure grade is between 0 and 100. If not, re-prompt. */

        if (grade < 0 || grade > 100)
       {
            printf ("*** Invalid Grade Entered ***.\n");
            continue;
       }

       /* Determine if this grade is in fact the highest grade. */

       if ( grade > highest )
            highest = grade;

       /* Increment grade counter. */

       grade_counter++;

    } /* end while loop*/

   /* Output the highest grade entered. */

    printf ("\nThe highest grade entered is %i.\n", highest);

    getchar();

} /* end main*/
```

Notice that the **grade_counter** (loop control variable) is only incremented if the grade entered is valid. This ensures us that we will in fact obtain the 3 valid grades that we are looking for.

A sample execution of the above program might be:

```
Enter grade (0 - 100): 80
Enter grade (0 - 100): 102
*** Invalid Grade Entered. ***
Enter grade (0 - 100): -20
*** Invalid Grade Entered. ***
Enter grade (0 - 100): 95
Enter grade (0 - 100): 78
```

The highest grade entered is 95.

That will do it for the **break** and **continue** statements. If you have any questions regarding these 2 statements, please post them in the discussion are for this week.

Hey. Maybe this is an omen. Perhaps we should take a "break" and then "continue" after a nice cup of coffee! (and a bag of chips!!!) :-)

The "switch" statement. A.K.A. the "case" statement.

The switch statement is the 4th (and final!) selection structure, and is simply another form of the **if/else if** statement. That is, when you have a lot of conditions to check a value against, you can use the if/else if/else if/ statement, or you can use the switch statement.

For example, if I prompted the user to enter the numbers from 1 to 5, and if I wanted to output the "text" versions of those numbers (one, two, three, four, or five) , I could use an **if/else if** statement as follows:

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

  /* Declare variable. */
    int   num;

  /* Prompt user for number. */

    printf ("Enter a number (1 - 5): ");
    scanf ("%i", &num);

  /* Based on number entered, output appropriate text. */

    if ( num == 1 )
        printf ("You entered the number one. \n");
    else if ( num == 2 )
        printf ("You entered the number two. \n");
    else if ( num == 3 )
        printf ("You entered the number three. \n");
    else if ( num == 4 )
        printf ("You entered the number four. \n");
    else if ( num == 5 )
        printf ("You entered the number five. \n");
    else
        printf ("You entered an incorrect number \n");

  /* Output farewell message. */

    printf (" \nGoodbye! \n");

    getchar();

} /* end main*/
```

A sample run of the above program might look like:

Enter a number (1 - 5): 3
You entered the number three.

Goodbye!

Now, if I wanted to use a *switch* statement to represent this code, I would be able to easily convert this if/else if to a **switch**. However this is not true for all **if/else if** statements. There is a restriction on the *types* of data you can test using the **switch** statement. The **switch** statement can only be used for testing for *equality*. That is, each and every case that is being tested must be for equality. Notice that every condition in the above **if/else if** structure contains a "==" relational operator? Only in those cases can we convert to a **switch** statement. So, with that said, and without further ado, (???) I will convert the above **if/else if** statement to a **switch** statement. Drum roll please.....

```
/* Based on number entered, output appropriate text. */

switch (num)
{
    case 1:
        printf ("You entered the number one. \n");
        break;
    case 2:
        printf ("You entered the number two. \n");
        break;
    case 3:
        printf ("You entered the number three. \n");
        break;
    case 4:
        printf ("You entered the number four. \n");
        break;
    case 5:
        printf ("You entered the number five. \n");
        break;
    default:
       printf ("You entered an incorrect number \n");
       break;
} /* end switch*/
```

This statement works exactly as the above if/else if statement. I want to point out a few important issues regarding the switch statement:

1. Notice there are no *begin* and *end* braces within each the body of each **case**. This is the only place in C where a multiple statement body does not require braces. (However feel free to insert them if you'd like.)

2. Notice the **break** statement at the end of each **case**. It is via this **break** statement that the **switch** is exited once a **case** is met. These **break** statements are required in a **switch** statement.

3. No two case values can be the same. That is, I could not have 2 different cases with the label. For example, you *could not* have the following in your **switch** statement:

```
case 1:
    printf ("You entered the number one. \n");
    break;
 case 1:
    printf ("Have a nice day! \n");
    break;
```

4. The **default** statement acts as the **else** statement. It will catch all values that do not match one of the cases. The **default** statement is not required.

5. You can have as many statements in the body of each **case** as desired. I have only 1 statement plus the **break** statement in the above example.

Note: I could not setup a **switch** statement to handle the sample **if/else if** program I illustrated in the previous topic. (Where we tested for grades >= 90, then grades >= 80, etc.) The reason for this is because we cannot test a "range of data" using a **switch** statement. As I stated earlier, each case must be a test for *equality*.

The program below is similar to the one in the previous topic, except we are prompting the user to enter a **letter** grade (A - F). Based on that letter grade, we display a message. I will use a switch statement to handle this:

```
#include <stdio.h>

void main (void)

{

  /* Declare variable. */

    char   letter_grade;

  /* Prompt user for letter grade. */
```

```
    printf ("Enter your letter grade (A - F): ");

    scanf ("%c", &letter_grade);

  /* Based on the letter grade entered, output appropriate text. */

    switch (letter_grade)
    {
        case 'A':
            printf ("Excellent! \n");
            break;
        case 'B':
            printf ("Very Good! \n");
            break;
        case 'C':
            printf ("Satisfactory. \n");
            break;
        case 'D':
            printf ("Poor. \n");
            break;
        case 'F':
            printf ("You failed. \n");
            break;
        default:
            printf ("You entered an incorrect letter grade \n");
            break;
    } /*end switch*/

  /* Output farewell message. */

    printf (" \nGoodbye! \n");

    getchar();

} /* end main*/
```

A sample run of the above program might look like:

Enter a letter grade (A - F): **B**
Very Good!

Goodbye!

Note that if the user enters a *lower case* letter grade, the **switch** will consider it an *incorrect* letter grade. So, instead of having to list 5 additional cases to handle the lowercase letters, the **switch** statement provides for 2 (or more) **case** values

having the same statements. You do this by listing the 2 different values together. It acts similar to the logical *OR* operator. For example, the above switch statement would be better as follows (to handle both upper and lower case letters):

```
switch (letter_grade)
{
    case ‘A’:
    case ‘a’:
        printf ("Excellent! \n");
        break;
    case ‘B’:
    case ‘b’:
        printf ("Very Good! \n");
        break;
    case ‘C’:
    case ‘c’:
        printf ("Satisfactory. \n");
        break;
    case ‘D’:
    case ‘d’:
        printf ("Poor. \n");
        break;
    case ‘F’:
    case ‘f’:
        printf ("You failed. \n");
        break;
    default:
        printf ("You entered an incorrect letter grade \n");
        break;
} /* end switch*/
```

Please read pages 83 - 86 for more information regarding the **switch** statement.

The last topic this week covers a miscellaneous feature of C: the "conditional operator".

The "switch" statement. A.K.A. the "case" statement.

The switch statement is the 4th (and final!) selection structure, and is simply another form of the **if/else if** statement. That is, when you have a lot of conditions to check a value against, you can use the if/else if/else if/ statement, or you can use the switch statement.

For example, if I prompted the user to enter the numbers from 1 to 5, and if I wanted to output the "text" versions of those numbers (one, two, three, four, or five) , I could use an **if/else if** statement as follows:

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

  /* Declare variable. */
    int   num;

  /* Prompt user for number. */

    printf ("Enter a number (1 - 5): ");
    scanf ("%i", &num);

  /* Based on number entered, output appropriate text. */

    if ( num == 1 )
        printf ("You entered the number one. \n");
    else if ( num == 2 )
        printf ("You entered the number two. \n");
    else if ( num == 3 )
        printf ("You entered the number three. \n");
    else if ( num == 4 )
        printf ("You entered the number four. \n");
    else if ( num == 5 )
        printf ("You entered the number five. \n");
    else
        printf ("You entered an incorrect number \n");

  /* Output farewell message. */

    printf (" \nGoodbye! \n");

    getchar();

} /* end main*/
```

A sample run of the above program might look like:

Enter a number (1 - 5): 3
You entered the number three.

Goodbye!

Now, if I wanted to use a *switch* statement to represent this code, I would be able to easily convert this if/else if to a **switch**. However this is not true for all **if/else if** statements. There is a restriction on the *types* of data you can test using the **switch** statement. The **switch** statement can only be used for testing for *equality*. That is, each and every case that is being tested must be for equality. Notice that every condition in the above **if/else if** structure contains a "==" relational operator? Only in those cases can we convert to a **switch** statement. So, with that said, and without further ado, (???) I will convert the above **if/else if** statement to a **switch** statement. Drum roll please.....

```
/* Based on number entered, output appropriate text. */

switch (num)
{
    case 1:
        printf ("You entered the number one. \n");
        break;
    case 2:
        printf ("You entered the number two. \n");
        break;
    case 3:
        printf ("You entered the number three. \n");
        break;
    case 4:
        printf ("You entered the number four. \n");
        break;
    case 5:
        printf ("You entered the number five. \n");
        break;
    default:
       printf ("You entered an incorrect number \n");
       break;
} /* end switch*/
```

This statement works exactly as the above if/else if statement. I want to point out a few important issues regarding the switch statement:

1. Notice there are no *begin* and *end* braces within each the body of each **case**. This is the only place in C where a multiple statement body does not require braces. (However feel free to insert them if you'd like.)

2. Notice the **break** statement at the end of each **case**. It is via this **break** statement that the **switch** is exited once a **case** is met. These **break** statements are required in a **switch** statement.

3. No two case values can be the same. That is, I could not have 2 different cases with the label. For example, you *could not* have the following in your **switch** statement:

```
case 1:
    printf ("You entered the number one. \n");
    break;
 case 1:
    printf ("Have a nice day! \n");
    break;
```

4. The **default** statement acts as the **else** statement. It will catch all values that do not match one of the cases. The **default** statement is not required.

5. You can have as many statements in the body of each **case** as desired. I have only 1 statement plus the **break** statement in the above example.

Note: I could not setup a **switch** statement to handle the sample **if/else if** program I illustrated in the previous topic. (Where we tested for grades >= 90, then grades >= 80, etc.) The reason for this is because we cannot test a "range of data" using a **switch** statement. As I stated earlier, each case must be a test for *equality*.

The program below is similar to the one in the previous topic, except we are prompting the user to enter a **letter** grade (A - F). Based on that letter grade, we display a message. I will use a switch statement to handle this:

```
#include <stdio.h>

void main (void)

{

  /* Declare variable. */

    char   letter_grade;

  /* Prompt user for letter grade. */
```

```
    printf ("Enter your letter grade (A - F): ");

    scanf ("%c", &letter_grade);

  /* Based on the letter grade entered, output appropriate text. */

    switch (letter_grade)
    {
        case 'A':
            printf ("Excellent! \n");
            break;
        case 'B':
            printf ("Very Good! \n");
            break;
        case 'C':
            printf ("Satisfactory. \n");
            break;
        case 'D':
            printf ("Poor. \n");
            break;
        case 'F':
            printf ("You failed. \n");
            break;
        default:
            printf ("You entered an incorrect letter grade \n");
            break;
    } /*end switch*/

  /* Output farewell message. */

    printf (" \nGoodbye! \n");

    getchar();

} /* end main*/
```

A sample run of the above program might look like:

Enter a letter grade (A - F): **B**
Very Good!

Goodbye!

Note that if the user enters a *lower case* letter grade, the **switch** will consider it an *incorrect* letter grade. So, instead of having to list 5 additional cases to handle the lowercase letters, the **switch** statement provides for 2 (or more) **case** values

having the same statements. You do this by listing the 2 different values together. It acts similar to the logical *OR* operator. For example, the above switch statement would be better as follows (to handle both upper and lower case letters):

```
switch (letter_grade)
{
    case ‘A’:
    case ‘a’:
        printf ("Excellent! \n");
        break;
    case ‘B’:
    case ‘b’:
        printf ("Very Good! \n");
        break;
    case ‘C’:
    case ‘c’:
        printf ("Satisfactory. \n");
        break;
    case ‘D’:
    case ‘d’:
        printf ("Poor. \n");
        break;
    case ‘F’:
    case ‘f’:
        printf ("You failed. \n");
        break;
    default:
        printf ("You entered an incorrect letter grade \n");
        break;
} /* end switch*/
```

Please read pages 83 - 86 for more information regarding the **switch** statement.

The last topic this week covers a miscellaneous feature of C: the "conditional operator".

The Conditional Operator: Unusual, yet commonly used.

There are many abbreviations in C that I would not normally recommend, as they make our programs quite cryptic, and difficult to read and understand. Although C provides for many of these abbreviations, I don't recommend them. Here is some good advice in C programming: "just because you ***can***, doesn't mean you ***should***!"

However, the conditional operator is one of those funky abbreviations (I said funky), that for some reason, has caught on, and is quite popular. So, with that in mind, and knowing that you will be *seeing* it "*out there*" in the "real world", I feel once again compelled to explain it to you at this time.

The conditional operator is used to abbreviate an **if / else** statement. Only, however, if the body of the **if** and the body of the **else** is a *single* statement. As an example, if I had program that prompted the user for 2 numbers and wanted to determine the *maximum* of the 2 numbers, using an **if / else** statement, my program might look like:

```
#include <stdio.h>

void main (void)
{

   /* declare variables. */

    int     maximum, num1, num2;
    char c;

   /* Prompt for 2 numbers. */

    printf ("Enter 2 numbers: ");
    scanf ("%d%d", &num1, &num2);
    while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

    /* Determine maximum of the 2 numbers. */

    if (num1 > num2)
        maximum = num1;
    else
        maximum = num2;

    /* Output maximum number. */

    printf ("The maximum of %i and %i is %i \n", num1, num2, maximum);
```

```
    getchar();

} /* end main*/
```

A sample run of the above program might be:

```
Enter 2 numbers: 23 45
The maximum of 23 and 45 is 45
```

Now, I will replace the above if/else statement with a statement containing a conditional operator:

```
maximum = num1 > num2 ? num1 : num2;
```

Pretty cool huh? I will color code the above statement, and describe it below:

This is what the statement looks like:

```
num1 > num2 ? maximum = num1 : maximum = num2;
```

This is what the statement means:

Is **num1 > num2** ?
Yes - Assign **num1** to variable **maximum**
No - Assign **num2** to variable **maximum**

The question mark (**?**) and the colon (**:**) are the 2 symbols which are used to denote the conditional operator.

The new program would be as follows:

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

   /* declare variables. */

    int     maximum, num1, num2;
    char c;

   /* Prompt for 2 numbers. */

    printf ("Enter 2 numbers: ");
    scanf ("%d%d", &num1, &num2);
    while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */
```

```
    /* Determine maximum of the 2 numbers. */

    maximum = num1 > num2 ? num1 : num2;

    /* Output maximum number. */

    printf ("The maximum of %i and %i is %i \n", num1, num2, maximum);

    getchar();

} /* end main*/
```

One final example of the conditional operator is shown below. First, I give an example of an **if/else** segment of code, and then I show the same segment below using a conditional operator. The segment of code should look familiar to you, as we used it in our discussion of the **if/else** selection structure (last week).

```
if (grade >= 60)
    printf ("You passed! \n");
else
    printf ("You failed. \n");
```

Now, using a conditional operator:

```
grade >= 60  ?  printf ("You passed! \n") :  printf ("You failed. \n");
```

Do you think you've got it? Great! That's all I have for this week. Please read the textbook for more information regarding the conditional operator. You don't have to use it - but if you see it you need to know what it does.

Next week we will be introducing Arrays.

Remember to complete Programming Assignment #3 during Week 6 of this course!

Hurray! Arrays!

Before we can even begin to discuss how to declare an array, I think it would be a good idea to explain what an array is! Below is a definition.

Array: *A group of contiguous memory locations related by the fact that they all have the same name and the same data type.*

Well. That should do it for you. It's crystal clear now, I'm sure. What? It makes no sense at all to you? I guess I should elaborate.

The reason we have arrays is so that we can begin to better *structure our data*. Yes. Just as we have techniques for structuring our code (loops, ifs, etc.), we have techniques to ensure that our data is also well structured. This improves the efficiency and readability of our code. Some of you might have heard of "Data Structures in C". It is one of the more advanced C Programming courses offered here at UML. Well, you'll be surprised to know, that arrays, are in fact data structures in C! It is the first data structure that you will cover.

Now, for a more specific explanation of what an array is, if you refer back to the definition above, we can reserve a "group" of memory locations (variables) instead of just 1 at a time. Each of the memory locations in the group will be contiguous (that is, they will follow one after the other... i.e., consecutive memory locations). We do this using arrays. If we do decide to use an array, each one of these memory locations will all have the same name, and will all be of the same data type. The fact that the locations are contiguous is what makes the data structure very efficient as far as processing time is concerned. This will be discussed in more detail at a later time.

You might be asking yourself: why would we ever use an array? We've gotten along quite well so far without them. Well, imagine if you had a program that needed to manage 5 grades. That is, the program prompted the user for 5 grades, calculated the average of the grades, and then output the 5 grades back to the user along with the average.

The dialog with the user would look something like:

Please enter grade #1: **100**
Please enter grade #2: **80**
Please enter grade #3: **97**
Please enter grade #4: **82**
Please enter grade #5: **73**

The grades entered are:

100
80
97
82
73

The average is 86

The code would look something like:

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

    /* Declare variables. */

    int grade1, grade2, grade3, grade4, grade5, sum, average;

    /* Prompt user for the grades. */

    printf ("Please enter grade #1: ");
    scanf ("%i", &grade1);
    sum = grade1;
    printf ("Please enter grade #2: ");
    scanf ("%i", &grade2);
    sum = sum + grade2;         /* could have been: sum += grade2;
    printf ("Please enter grade #3: ");
    scanf ("%i", &grade3);
    sum = sum + grade3;         /* could have been: sum += grade3;
    printf ("Please enter grade #4: ");
    scanf ("%i", &grade4);
    sum = sum + grade4;         /* could have been: sum += grade4;
    printf ("Please enter grade #5: ");
    scanf ("%i", &grade5);
    sum = sum + grade5;          /* could have been: sum += grade5;

    /* Calculate the average of the grades entered. */

    average = sum / 5;

    /* Output the grades entered along with the average. */

    printf ("\nThe grades entered are:\n\n");
    printf ("\t%i\n", grade1);      /* \t for a tab
    printf ("\t%i\n", grade2);
    printf ("\t%i\n", grade3);
```

```
    printf ("\t%i\n", grade4);
    printf ("\t%i\n", grade5);
    printf ("\nThe average is %i\n", average);

} /* end main*/
```

Already, this seems cumbersome, with only 5 grades. What if your program needed to manage 30 grades? What about 100 grades???!?!?!?

Well, it would be a nightmare to have to declare 100 integer variables, one for each grade. Not to mention the amount of code that would be required to be written to work with this information. This is why we have arrays. We can structure this related data in the form of an array.

Using the example above, where we have 5 grades, I will declare an array to hold 5 grades. In other words, I will declare a variable (called **grades**) which will not contain a single value, but instead will contain and entire set of values (in this case, 5 values). Below is the declaration of the array **grades**:

```
int  grades[5];
```

The declaration above tells the C compiler to allocate 5 memory locations, all of type **int**. Each memory location is referred to as an "element". The array above has 5 elements. We reference individual elements in an array by using a *subscript* number. The subscript of the first element is 0, the second element is 1, etc. (In the C language, the first element of an array is always subscript 0).

So, in other words:

The **first element** in array grades[5] is referenced as:	**grades[0]**
The **second element** in array grades[5] is referenced as:	**grades[1]**
The **third element** in array grades[5] is referenced as:	**grades[2]**
The **fourth element** in array grades[5] is referenced as:	**grades[3]**
The **fifth element** in array grades[5] is referenced as:	**grades[4]**

An individual element can be used anywhere in the program that a simple variable can be used. For example:

```
grades[0] = 80;
grades[1] = 100;
grades[2] = grades[0];   /* in this case, grades[2] would be 80
```

Note:

- When using the subscript (or index) of an array in the code (the number between the brackets), that value must be an integer value. That is, you

could not set element **grades[1.5] = 80;**

- An array declared as having 100 elements, **nums[100]** for example, is referenced in the code as elements **nums[0]** through **nums[99]**.

- Be careful. C would allow you to reference beyond the end of the array bounds! That is, if you declare an array to be 100 elements, **nums[100]** for example, and you had the following line of code in your program: **nums[200] = 50;** The compiler and linker would allow this. Only during runtime you may get an error, because you are now writing into a memory location that you did not reserve.

- Array names follow the same conventions as variable names.

- Arrays can contain data of any type. For example, if you need to store a bunch of related floating point numbers, you can declare an array to be of type **float** as shown:

  float values[10];

  This would reserve 10 floating point locations in memory.

  If I wanted to set the first element to be equal to 22.12, I would use: **values[0] = 22.12;**

- Arrays are normally processed within a loop (**for**, **do**, **while**), with the loop control variable being used as an array subscript. (See example that follows.)

Now, I will jump ahead just a bit here and convert the above (grades) program to a program which uses arrays. You will quickly see how this type of data structuring makes a lot of sense!

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

    /* Declare variables. */

    int    grades[5], average, sum = 0, x;
    char c;

    /* Prompt user for the grades. */

    for (x = 0; x < 5; x++)
    {
```

```
        printf ("Please enter grade #%i: ", x + 1);
        scanf ("%i", &grades[x]);
        while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

        sum = sum + grades[x];

    }/* end for loop*/

    /* Calculate the average of the grades entered. */

    average = sum / 5;

    /* Output the grades entered along with the average. */

    printf ("\nThe grades entered are:\n\n");

    for (x = 0; x < 5; x++)
        printf ("\t%i\n", grades[x]);

    printf ("\nThe average is %i\n", average);

} /* end main*/
```

Notice a few things here, when working with arrays, we usually start the loop control variable at 0 instead of 1. We do this because we normally use the loop control variable as the subscript in the array. Since the array subscript starts at 0, we begin our loop variable at 0. Also, notice the loop condition is: **x < 5**, instead of **x <= 4**. We do this for readability. It is clear, at a quick glance, that we are processing 5 elements in the array (not 4). However, if we said: **x <=5**, then we would process 1 element too many! So we say: **x < 5**.

One very important observation to be made here, is that the above program would be easily modified if the number of grades to process was 100, instead of 5. We would simply make a change to the array size, and then to the loop condition test (x < 5). Also, in the average calculation, we would divide by 100, instead of 5. Actually, to make the above program easily "changeable", we could declare a *symbolic constant* which contains the size of the array, and we would use this constant instead of the hard coded number 5. Constants are similar to variables, except that their values cannot be changed in the program. We define a symbolic constant using the preprocessor command:

```
#define  NUM_GRADES  5
```

Notice the symbolic constant name is uppercase. We do this as a standard. Also, notice that there is no assignment operator for setting the value of the constant

NUM_GRADES to **5**. You simply put a space after the constant name and give it a value.

So, now, with this new tool in hand, I will rewrite the above program using a constant for the amount of grades:

```
#include <stdio.h>
#define NUM_GRADES  5
void main (void)
{

    /* Declare variables. */

    int     grades[NUM_GRADES], average, sum = 0, x;
    char c;

    /* Prompt user for the grades. */

    for (x = 0; x < NUM_GRADES; x++)
    {
        printf ("Please enter grade #%i: ", x + 1);
        scanf ("%i", &grades[x]);
        while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

        sum = sum + grades[x];

    }/* end for loop*/


    /* Calculate the average of the grades entered. */

    average = sum / NUM_GRADES;

    /* Output the grades entered along with the average. */

    printf ("\nThe grades entered are:\n\n");

    for (x = 0; x < NUM_GRADES; x++)
        printf ("\t%i\n", grade[x]);

    printf ("\nThe average is %i\n", average);

} /* end main*/
```

Now, here is the real strength of arrays. If we decided that we had 100 grades, instead of 5, we would have to make just a simple change to the above program.

That is, we would change the value of the symbolic constant **NUM_GRADES** to be 100 instead of 5. Then recompile and re-link. That's all! The program would work for 100 grades, just as well as it worked for 5. We could make it 1000 grades! Amazing huh?!? Try doing *that* with *simple variables*!

Important Distinction

The book shows on page 115 Variable-Length Arrays. This is **not** supported by most compilers, will definitely not run in C-Free, and the example given violates the ANSI C rule that the variables in a block be declared before all other statements. So, do not use a variable length array.

We have covered quite a bit in this section. (I think arrays are so terrific, that I couldn't wait to share it all with you!) Don't fret though. Have I let you down yet? We will now slow down, and even back up a bit, and take the time to process this stuff. The next section discusses the simple concept of "initializing arrays". In the above examples I did not clear (flush) the input buffer; as you should after each scanf. Other editing for proper grade ranges may be helpful also. See program #4.

This topic is relatively simple. :)

Sometimes we need our arrays to contain values before we use them in the body of the program. Arrays, like simple variables, are not automatically initialized to zero for you. Like simple variables, we might want to initialize the array elements ahead of time, before they are used. There are 2 useful techniques for initializing an array(other than just assigning each element a value):

1. Initializing Arrays During Declaration.
2. Initializing Arrays in a for Loop.

Each of these techniques are discussed below.

1. Initializing Arrays During Declaration.

Just as we can assign initial values to simple variables during declaration, we can also assign initial values to elements of an array during declaration. We initialize elements of an array by listing the initial values of the array, starting from the first element, in a comma-separated list, which is enclosed in braces.

For example, if I needed to initialize the integer array **nums[5]** during declaration, so that the first element had the value 10, the second element had the value 20, the third element the value 30, the fourth element the value 40, and the first element the value 50, it would look like:

int nums[5] = {10, 20, 30, 40, 50};

Note that you do not have to initialize every element in the array during declaration. For example, if I declared the **nums[5]** array as follows:

int nums[5] = {10, 20, 30};

Then **nums[0]** would equal 10, **nums[1]** would equal 20, **nums[2]** would equal 30, **nums[3]** would equal 0, and **nums[4]** would equal 0. The rule is: *if there are fewer initializers than elements, the remaining elements are initialized to zero.*

So, with that tidbit of information in mind, if I wanted to initialize the entire **nums** array to all zeros, I simply have to type:

int nums[5] = {0};

Now I ask you: What if I wanted to initialize the entire **nums** array to all 1's. Could I simply put:

int nums[5] = {1};

That's right, the answer is ***no.*** (I can read your mind, and I can tell that you answered this one correctly.) Oh. Were you saying no *more*?! I thought you were answering my question. Well, just a bit more to discuss, then it's time for a break.

If I initialized arrays **nums** as shown above, the first element would be initialized to 1, but, using the above rule, the remaining 4 elements would be initialized to zero.

So, to initialize array **nums[5]** to all 1's, I could do the following:

```
int  nums[5] = {1,1,1,1,1};
```

This seems just dandy, however, what if array nums had 100 elements? What if it had 1000 elements. What about a million? In the "real world", array sizes can get quite large, so it would not be practical to initialize an array this way. In reality, unless you are initializing your array to all zeros, we normally do not initialize the array during declaration. If an array needs to be initialized before it is used, then we use the second technique for initializing it. We initialize the array using a **for** loop.

2. Initializing Arrays in a for Loop

Assume the **nums** array above had 100 elements. If I wanted to initialize every element in that array to all 1's, I would do that initialization in a **for** loop, as shown below:

```
int  i, nums[100];

for (i = 0,  i < 100; i++)
    nums[i] = 1;
```

That's it! That's all there is to it. You can quickly set every element to whatever you want, using this technique. This can also be done for arrays of type **float**. For example, if I needed to declare and initialize a **float** array called **values[5]** to be 100.1, 200.2, 300.3, 400.4, and 500.5 respectively, I could do this during declaration (since it is a small array) such as:

```
float  values[5] = {100.1, 200.2, 300.3, 400.4, 500.5};
```

Or, I could do this in a **for** loop:

```
int    i;
float  values[5];

for (i = 0;  i < 5;  i++)
    values[i] = (i+1) * 100.1;
```

That's all there is to this! Simple huh?!?!

Please review programs 6.1 and 6.5 for additional examples of declaring and initializing arrays. I am not too thrilled with programs 6.2, 6.3 and 6.4 at this time. I think they are too complex for students who are learning the basic fundamental concepts of arrays. If you are up for the challenge, go right ahead and try to understand them. If not, please feel free to skip right over them. We will cover additional examples in the next topic titled: *Working with Numeric Arrays*.

Array Techniques & an additional example using an Integer array.

Are you still with me? Go ahead and make yourself a cup of coffee. Make it decaf though, because this won't take long, and you can get right to bed!

I want to give some short 'snippets' of some array techniques that can come in handy.

1. Multiplying all elements of an array by a number:

```
    for (x = 0;  x < how_many;  x++)
        My_array[x] = My_array[x] * 5;  /* mult by 5, (or add or subtract, just
change the operation) */
```

that's it!

2. Copy an array to another array:

```
    for (x = 0;  x < how_many;  x++)
        My_array[x] = Other_array[x]; /* copies other_array to my_array*/
```

that's it!

3. Sum up all the elements in an array:

```
    total = 0;  /* initialize accumulator */
    for (x = 0;  x < how_many;  x++)
        total = total + My_array[x];  /* total will contain all the data added
together! */
```

that's it!

Pretty cool, huh?

Another example:

This topic uses an additional example to firm-up what we have discussed so far regarding arrays. The following program simulates someone tossing a die. (singular form of dice!) The program asks the user what their roll was. The user must enter 1 to 6. Anything other than 1 to 6 will not be accepted by the program, and the user must roll again. The program asks the user to roll 10 times. Once the user successfully rolls 10 times, the program then outputs a log of how many 1's, how many 2's, how many 3's, how many 4's how many 5's and how many 6's were rolled. The power of arrays is that it holds all the data in an organized way

so that you can retrieve the data later in the program. A sample program execution is as follows:

```
What is your roll #1?  6
What is your roll #2?  8
*** Invalid roll. Please roll again. ***
What is your roll #2?  4
What is your roll #3?  5
What is your roll #4?  4
What is your roll #5?  6
What is your roll #6?  1
What is your roll #7?  4
What is your roll #8?  2
What is your roll #9?  1
What is your roll #10?  6

You rolled:

2  1's
1  2's
0  3's
3  4's
1  5's
3  6's

Thank you for playing!
```

The program to solve the above problem will use an integer array to keep track of the number of 1's, 2's, etc. It will basically be a 6 element "counter" array. The first element represents the number of 1's. The second element represents the number of 2's etc. In this particular example, it makes sense to initialize the array elements (counters) to all zeros. Once the user rolls a certain number, we will increment the element which represents that particular counter.

Note that since the element numbers start at 0, which represents the number of 1's rolled, we will have to do a bit of math to increment the correct counter. That is, we can't simply increment element 1 if we roll a 1, because we really need to increment element 0 when we roll a 1. For this reason, instead of using the rolled value as an index into the array, we need to use the rolled value *minus 1*. This is a common programming consideration when working with arrays. The program is as follows:

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

```
    /* Declare and initialize variables. */

    int   x = 0, counter[6] = {0}, roll;

    /* For each of 10 rolls, get value that use rolled. */

    while ( x < 10 )
    {
        printf ("What is your roll #%i? ", x+1);
        scanf ("%i", &roll);

        /* Ensure that the roll was between 1 and 6.
           If not, issue error message and re-prompt. */

        if ( roll < 1 || roll > 6 )
            printf ("*** Invalid roll. Please roll again. ***\n");

        /* If roll is valid, increment appropriate roll counter and
           loop control variable. */

        else
        {
            counter[roll-1] = counter[roll-1] + 1;   /* could be: counter[roll-1]++
*/
            x++;
        }/* end else */

    } /* end while loop*/

    /* Output the counters for each number from 1 to 6. */

    printf ("\n\nYou rolled:\n\n");
    for (x = 0;  x < 6;  x++)
        printf ("%i   %i's\n", counter[x], x+1);

    /* Output a farewell greeting.  */

    printf ("\n\nThank you for playing!\n");

} /* end main*/
```

Again, note that the loop starts at ***zero*** and goes to less than the size of the array. It could have been written for (x = 0 x <=5; x++), for example.

It is a common programming mistake (uh oh, more points deducted?) to begin the array at 1 and go up to the size of the array - which is out of bounds (and

may cause runtime errors). I don't like the idea of making an array one bigger and ignoring the zero position, although I have seen it done.

I realize that this program is a bit tough to digest the first time looking at it. May I suggest that you print it out, and try to step through it. Put on paper what values are where, and take it 1 line at a time. You will be amazed how simple this program really is, once you step through it and "play computer". That is, pretend that you are the CPU, executing this program one line at a time, keeping track of each and every variable along the way. I know you can do it!!!

If you have any questions regarding arrays so far, please post them in the discussion board for Week 7. We will be discussing character arrays, as well as multi-dimensional arrays next week. Yippie!

Hip Hip Hurray! More and more arrays!!! :-)

And you thought you already knew all there was to know about arrays! Well, you do. Almost. *Numeric* arrays are wonderful for processing numeric data, however, sometimes we need to process *character* data. What exactly do I mean by that? Well, first of all, ***the C language does not have a data type called string.*** That is, if we wanted to store the name: "Mary" in a variable, we obviously could not store it in a variable of type **int** or **float**, and, not as obvious, we could not store it in a variable of type **char**, because a variable of type **char** can hold only a *single* character. So, we are stuck.

But, not really. If you think of what an array is (a group of consecutive memory locations, storing related data of the same type), and if you think of a what a text string is (your name, for example): a set of consecutive characters, all related by the fact that they belong to the same word; then you can perhaps come to the conclusion that C handles string data by processing them as *arrays of characters*! Yes! I knew you would come to that conclusion!

Huh? You didn't come to that conclusion? You were thinking about what you were going to eat next? Well, let's see if I can explain things in a way that you can understand! Unfortunately, the textbook does not discuss character arrays too much. In fact, in the few pages that they are discussed (pages 117 and 121), I don't think that the examples are very good. So, I will provide additional information and examples here.

A string in C is stored as an array of characters. If we did in fact have a string data type (let's for a moment imagine that C did in fact provide one, and it was called: string), then the following segment of code could be used to prompt a user for their first name, and store it in the *string* variable called **first_name**. Note, the **%s** format specifier used for strings in ***the "pretend" example below:***

A sample output of the program below might look like:

Please enter your first name: **Mary**
Hi Mary, nice to

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

    /* declare variables.*/

    char     c;
    string   first_name;       /* notice variable first_name declared as type
string. */
```

```
                                        /* ...and as we know, there is no such data type!
*/

    /* Prompt user for first name. */

    printf ("Please enter your first name: ");
    scanf ("%s", &first_name);
    while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

    /* Display first name back to user. */

    printf ("Hi %s, nice to meet you!\n, first_name);

} /* end main*/
```

This seems almost digestible doesn't it?!? Unfortunately, ***there is no such data type as string*.** But, the good news is that if we declare variable **first_name** as an **array of characters**, the rest of the program hardly changes at all! C even allows us to prompt for data into an array of characters, and use the **%s** format specifier in the **scanf** statement when we do! Also, when we output the contents of an array of characters, C allows us to use the **%s** format specifier in our **printf** statement as well!

So, without further ado, I will convert the above code to the correct C code, using an **array of characters** instead of a variable of type **string**:

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

    /* declare variables.
    char    first_name[20];          /* Notice "array of character" data type
here. */
    char    c;

    /* Prompt user for first name. */

    printf ("Please enter your first name: ");
    scanf ("%s", first_name);       /* Notice no "&" symbol in scanf statement.
*/
   while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

    /* Display first name back to user. */

    printf ("Hi %s, nice to meet you!\n", first_name);
```

} /* end main*/

The only lines of code that actually changed above are the 2 lines highlighted in **blue** text.

The first change is simply to use an **array of characters** instead of the pretend data type of **string**.

One comment to be made regarding arrays of characters, be sure to declare enough elements to hold the data that you will be storing. In this example, I declared the size of the array **first_name** to be 20 elements. This tells C that I have allocated space for *at most,* 20 characters in the first name. If I had declared this array to be size 5, (char **first_name[5]** for example), and the user entered: Kristina as the name, then **scanf** would try to store all 8 characters in the array (plus something called the "null terminator", discussed in chapter 10 -- not in this course.) The program might continue just fine, even though I have written into memory that I did not reserve, however, it might crash. You want to be sure to make your character arrays larger than you need!

The second change is not so obvious. If you look closely, you will notice that the ampersand symbol (**&**) is not in its usual place in the **scanf** statement. In fact, it is not there at all. This has to do with pointers, which will not be discussed in this class. (We discuss pointers in Part 2 of this course.) All you have to know is that when you use the **scanf** statement, and if you use the **%s** as the format specifier (because you are reading data into a character array), then you do not use the **&** symbol.

Now that we have fast forwarded a bit, let's backtrack all the way to the beginning. Let's start at the very beginning. A very good place to start. When you read you begin with A-B-C, when you sing you begin with DO-RE-ME. :)

Seriously though, let's move beyond the Sound of Music for a bit, and discuss the first step in working with character arrays, which is *declaring them*.

Declaring Character Arrays.

Character arrays are declared just like other arrays. That is, you provide an array name (descriptive), as well as the number of elements you need. With character arrays, we typically allocate a number of elements above and beyond what we would normally use. That is, if you want to store someone's last name in a character array, you could make the array size 50. If you were prompting for someone's street address, perhaps 50 characters would be sufficient for that too. Below illustrates 2 character arrays declared to hold a person's first name and last name:

char first_name[30], last_name[50], c;

Now, if you wanted to prompt for someone's first and last names, you could use the following code segment:

```
printf ("Please enter your first and last name: ");

scanf ("%s%s", first_name, last_name);
while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

printf ("\nGolly %s, are you related to John %s?\n", first_name, last_name);
```

The output from the above segment of code might look like:

Please enter your first and last name: **Mary Jones**

Golly Mary, are you related to John Jones?

(I don't often use the word "golly", but it seemed to fit nicely in this example.) In this example, "Mary" is stored in variable **first_name**, and "Jones" is stored in variable **last_name**.

What if I wanted to store the *entire name* in a single character array. Say: full_name. This would be a problem with **scanf**, as **scanf** will read only up until a **space**, **tab**, or **carriage return** from the input buffer (stdin). Recall: once the user presses <ENTER> on data entry, whatever the user types will be placed in the input buffer (stdin). **scanf** will then grab everything it sees up until a space, tab, or carriage return! With that in mind, try to determine what the following segment of code will do:

```
char  full_name[80], c;

printf ("Please enter your name: ");
scanf ("%s", full_name);
while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */

printf ("Your name is %s\n", full_name);
```

The output of the above segment of code might look like:

Please enter your name: **Mary Jones**
Your name is Mary

Notice that "Jones" did not get output. That is because **scanf** only took the word "Mary" out of the input buffer. It stopped as soon as it reached the space character. So, the rest of the text ("Jones") is still in the input buffer! Then, after the buffer is cleared, the remaining text is expunged. Not exactly what we had in

mind! There is a way to read the entire *string of text* into an array, including any *white space*, if that is what you want. You need to use a different function, not discussed in this chapter however. I will briefly mention it here.

To read in an entire string of data from the input buffer, you would instead use function **gets**. It is the input function specifically for strings (character arrays). You use it as follows:

```
char  full_name[80];

printf ("Please enter your name: ");
gets (full_name);

printf ("Your name is %s\n", full_name);
```

Now, if I ran the above code using the same data, I would get:

Please enter your name: Mary Jones
Your name is Mary Jones

Viola!

Note that with **gets**, you do not need to use the **%s** format specifier, because gets is strictly for character arrays (strings. It actually stands for: "get string"). Unlike **scanf**, which is quite generic, for many data types, **gets** already knows that your are prompting for a string, so the **%s** is not necessary.

Because of this issue with the **scanf** function and string data, you always want to be sure to *flush the input buffer* after using a **scanf** for strings. In case the user enters more than a single word at the prompt, you want to clear out the remaining text in the input buffer. So, a typical segment of code, which prompts for first name only, would look like:

```
char first_name[30], c;

printf ("Please enter your first name: ");

scanf ("%s", first_name);
while ( (c = getchar() != '\n') && c != EOF);   /* clear input buffer  */
```

Wow. We sort of drifted off from "Declaring Character Arrays" to "Using Character Arrays". That's okay though. It seemed to flow nicely.

Now, we move on to the next important topic that we haven't yet discussed, which is "initializing character arrays".

Initializing Character Arrays.

Sometimes, you might want to assign an initial value to a character array. That is, you might want to initialize it to something during declaration. Character arrays can be initialized during declaration in 1 of 2 ways. You will quickly see however, that the method of choice is the second technique shown below.

First technique:

Recall: when we initialize arrays during declaration we create a comma-separated list of elements within 2 braces. In the case of a string, which is really an array of characters, the initialization might look like:

```
char  first_name[30] = { 'M', 'a', 'r', 'y', '\0' };
char  last_name[50] = { 'J', 'o', 'n', 'e', 's', '\0' };
```

The above segment of code would initialize variable **first_name** to be "Mary" and variable **last_name** to be "Jones". This is a very time-consuming technique for initializing a character array, so we tend to use the second technique, discussed below.

Note: The '**\0**' character at the end of each string is called the "null terminator". It is required so that **printf** (and other string handling functions) knows when to stop outputting characters. It marks the *end of the text*. **printf** cannot use the *size of the array* to determine when to stop printing, because typically we have fewer characters than the size of the array. This is discussed in greater detail in chapter 10 (in the "C Programming Part 2" course).

Second technique:

The more commonly used technique for initializing character arrays is shown below. It is the most commonly used because it is easier! That is the only reason. It is no more or less efficient than the first technique. Only easier. I'm all for that! Here goes:

```
char  first_name[30] = "Mary";
char  last_name[50] = "Jones";
```

Ahhh. Much better. Here we do not even have to worry about the null terminator! It is automatically put in the array for you. Simply surround the desired text in double quotes, and you are good to go!

This is all I had on string data (character arrays) at this time. This textbook devotes an entire chapter on more details of character and string data, chapter 10. This chapter is not discussed in the class, however it is discussed in the Part

2 section of this course (90.212). Feel free to take a look, and post questions to the discussion board as needed.

One final note that the textbook discusses in this particular section is that if you omit the *size* of the array during declaration, the compiler will count up the *number of initializers* that you specify, and use that as the size of the array. This goes for all data types. This is **not a good practice**.

You should always specify the size of your array. It greatly improves the readability of your code. There is no benefit to leaving the size out. *If you do not initialize your array during declaration*, *then YOU MUST indicate the size of the array in the brackets*. Remember, if a size is not specified, the compiler will simply count up the number of initializers that you have listed.

For example, if I had the following arrray declarations:

```
char   first_name[ ] = "Mary";
int    nums[ ] = {10, 20, 30, 40, 50, 60};
float  values[ ] = {10.2, 20.3};
```

The above lines of code would declare array **first_name** to be size 5 (4 letters in the word "Mary" plus the null terminator); It would declare array **nums** to be size 6, and array **values** to be size 2.

The following line of code *would* be invalid:

```
int  nums[ ];
```

The above line of code is invalid because the compiler has no idea how big to make that array. You did not provide a size between the braces, and you did not provide initializers. The compiler needs one or the other.

Okay. This is *really* all I have in this topic. See you in the next topic titled: Multi-Dimensional Arrays. Are you ready for the next dimension??? Now would be a good time to get that very large cup of coffee (and maybe some Bailey's added in!).

Doo Dee Doo Doo. Doo Dee Doo Doo. You have now entered the Twilight Zone.

Remember that one?!?! Am I dating myself or what?!?!

I think I'll just jump right in with this one. If I give you lots of introductions regarding multi-dimensional arrays, it might scare you. Luckily, you will *not* have to use multidimensional arrays in an assignment.

So, here goes. Sometimes we want to write a program that ultimately needs to display data in a *table* or *grid* form. Something with *rows* and *columns*. For example, think of a spreadsheet program. All of that data is displayed in tabular form. Well, if we are working on a program that displays data in tabular form, it makes sense to store that data, and work with that data as "rows and columns" of data. Rather than simply a *linear* set of data as we find in a single-dimensional array. (All of the arrays discussed so far has been single-dimensional arrays.) With arrays, each subscript is considered a dimension.

Arrays that represent data as rows and columns require a second subscript. We call these types of arrays "double-dimensional" arrays, "2-dimensional arrays" or "double-subscripted" arrays. Take your pick! I have heard them all. You might also hear "multi-dimensional" arrays, which simply means arrays with more than 1 dimension. (You can actually have even more than 2 dimensions, but we will not go there!)

At this stage of your programming life, you are not required to have too much experience with multi-dimensional arrays. You are simply required to have a basic knowledge and understanding of them.

So, I will briefly discuss the following topics:

1. Declaring 2-dimensional arrays.
2. Initializing 2-dimensional arrays.
3. Using 2-dimensional arrays.

Declaring 2-Dimensional Arrays.

To declare a 2 dimensional array, you follow the same rules as any other array. That is, you give it a descriptive name, and declare it to be the same type as the data it will contain. For example, if we wanted to declare an array that will hold 2 grades for 3 different students, we could declare an array as follows:

```
int  grades[3][2];
```

This would allocate 6 integer locations in memory (3 * 2). This array contains 3 rows and 2 columns of data. It is known as a "3 by 2" array. Let's assume that this is reserving enough space to hold up to 2 grades, for up to 3 students.

We reference each element as follows:

grades[0][0] = 100;

This would set the first grade for the first student to be 100. To set the first grade for the second student to be 90, it would look like:

grades[1][0] = 90;

This takes a while to process. But when the above declaration was made, the following memory locations were reserved:

grades[0][0] = first student, first grade
grades[0][1] = first student, second grade

grades[1][0] = second student, first grade
grades[1][1] = second student, second grade

grades[2][0] = third student, first grade
grades[2][1] = third student, second grade

Initializing 2-Dimensional Arrays.

This is a good time to discuss initializing 2-dimensional arrays. This can be done in one of the two following ways:

1. During declaration.
2. Within the processing code.

Initializing 2-dimensional arrays *during declaration*:

To initialize during declaration you simply put all the initializers in a comma-separated list. You can do this in 1 of 2 ways. You can list them linearly, or by rows and columns.

Initialization in a *linear* fashion during declaration:

int grades[3][2] = { 100, 90, 80, 95, 75, 89};

This would set:

grades[0][0] to 100
grades[0][1] to 90

grades[1][0] to 80
grades[1][1] to 95

grades[2][0] to 75
grades[2][1] to 89

Initializing using a *rows* and *columns* technique during declaration:

```
int   grades[3][2] = { {100, 90},
                       {80, 95},
                       {75, 89} };
```

The above initialization would yield the same values as the linear initalization technique above. In either of these 2 techniques, if their are fewer initializers than elements, the remaining elements would be set to zero. So, to initialize the entire array to zero, you simply use:

```
int   grades[3][2] = {0};
```

We rarely initialize multi-dimensional arrays during declaration (unless we are initializing it to zero), because the array sizes are normally quite large, and it would be very cumbersome. For this reason, we normally will stuff our array with data from within the processing of the code. The data normally comes from either the user, or a data file.

Initializing 2-dimensional arrays *within the processing code*:

Recall: when we work with single-subscripted arrays, we typically process the arrays within a loop (for, do, or while). Well, with 2-dimensional arrays, we process the arrays within a *nested* loop. A loop within a loop!

For example, if I wanted to initialize the above **grades** array to all 100s (for whatever reason, I'm not sure), the code would look like:

```
int   row,  col;
int   grades[3][2];

for  (row = 0;   row < 3;  row++)
{
    for (col = 0;  col < 2;  col++)
        grades[row][col] = 100;
}
```

This will run the inner loop ("col") from 0 to 1 for each "row" value. So, for row = 0, col will be equal to 0 first, then 1. For row = 1, col will be 0 first, then 1. For row = 2, col will be 0 then 1.

This sets all elements of the array to 100. If this array was 1000 rows by 10000 columns, this same loop could be used. Only your range of values would change as follows:

```
for  (row = 0;   row < 1000;  row++)
{
    for (col = 0;  col < 10000;  col++)
        grades[row][col] = 100;
}
```

So, you can see how easy it is to initialize this data, of any size.

Look at the following code segment. What do you think the values of array **nums** will be:

```
int   nums[3][4], row, col;

for (row = 0;  row < 3;  row++)
{
    for (col = 0;  col < 4;  col++)
        nums[row][col] = row + 1;
}
```

Think about it for a minute before you look at the solution that follows.

nums[0][0] is 1
nums[0][1] is 1
nums[0][2] is 1
nums[0][3] is 1

nums[1][0] is 2
nums[1][1] is 2
nums[1][2] is 2
nums[1][3] is 2

nums[2][0] is 3
nums[2][1] is 3
nums[2][2] is 3
nums[2][3] is 3

Did you guess correctly? It is tough to do without jotting things down on paper. I usually draw a grid, with rows and columns, and label each "cell" with the

appropriate subscript. Then I start to fill in the cells with the correct, calculated value. This helps quite a bit.

Using 2-Dimensional Arrays.

The following program illustrates the use of a 2-dimensional array. It starts off with an array containing some numbers (initialized during declaration). It then calculates the "square" of each of the numbers in the array, and replaces the original numbers with the "squared" value of the original numbers. The contents of the array are printed out both before, and after the squaring of the numbers.

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

    /* Declare variables and initialize array. */

    int   nums[3][5] = {2, 3, 5, 2, 4, 8, 9, 1, 0, 3, -5, 4, 6, 10, -12};
    int   row, col;

    /* Output contents of array before squaring. */

    printf ("\nThe array before squaring:\n\n");

    for  (row = 0;   row < 3;  row++)
    {
        for (col = 0;  col < 5;  col++)
            printf ("%8i ", nums[row][col]);
        printf ("\n");

    }/* end for loop*/

    /* Now, replace original values with their squares. */

    for  (row = 0;   row < 3;  row++)
    {
        for (col = 0;  col < 5;  col++)
            nums[row][col] = nums[row][col] * nums[row][col];

     }/* end for loop*/

     /* Output contents of array after squaring. */

    printf ("\nThe array after squaring:\n\n");
```

```
    for  (row = 0;   row < 3;  row++)
    {
        for (col = 0;  col < 5;  col++)
            printf ("%8i ", nums[row][col]);
        printf ("\n");

   }/* end for loop*/



} /* end main*/
```

The output of the above code is:

The array before squaring:

```
   2     3     5     2      4
   8     9     1     0      3
  -5     4     6    10    -12
```

The array after squaring:

```
   4     9    25     4     16
  64    81     1     0      9
  25    16    36   100    144
```

Another (more involved) example:

```
#include <stdio.h>
/*
This program uses two two-dimensional arrays as described here:

1. The "original student array" contains 6 columns of data for each student
(up to 25 students). These columns represent the student ID as well as 5
grades. It is initialized with the actual data during declaration.

2. A "student array" which contains 7 columns of data for each student.
These columns represent the student ID, as well as the 5 grades. The 7th
column will contain a calculated average. This entire array is initialized to
zero during declaration. It will ultimately contain a copy of the original
student array, and the calculated average.
*/
void main()
{
        int original_student_array[25][6]=
        {       12233,87,93,98,77,69,
```

```
                21033,35,91,89,57,77,
                13456,78,99,98,98,99,
                22020,55,96,77,88,99,
                10103,65,91,79,87,76,
                18456,79,89,98,88,91,
                29920,54,66,79,88,92,
                22223,39,81,69,87,73,
                16456,68,85,98,87,94,
                16020,89,69,77,68,96 };

        int student_array [25][7] = {0};   /* initializes array to zero. The 7th
column will hold the average of grades */

        int row;
        int col;
        int num_students = 10;
        int num_grades = 5;
        int average;

        /* string constants for report headers  */

        char header1[80] = "                University Student Report \n";
        char header2[80] = "                ---------------------------------- \n\n";
        char header3[80] = "Student ID   Exam 1    Exam 2    Exam 3    Exam
4   Exam 5    Avg\n";
        char header4[80] =
"=============================================\n";

        /* Example of copying elements of one two dimensional array to
another  */
        /* This will yield 2 copies of the array.
*/

        for (row = 0; row < num_students ; ++row)
                for (col = 0; col <= num_grades; ++col)
                        student_array[row][col] =
original_student_array[row][col];

        /* compute average for each row (student) - and assign to 7th
column  */

        for (row=0; row < num_students; ++row)
        {
                /* initialize average to zero for each student. */

                average = 0;
```

```
                /* For loop to add all grades for this student. */

                for(col = 1; col <= num_grades; ++col)
                        average = average + student_array[row][col];

                average = average / num_grades;     /* calculate average */

                student_array[row][6] = average;     /* store average in 7th
column */
        }

        /*  display report  */

        printf("\n\n");
        printf("%s", header1);
        printf("%s", header2);
        printf("%s", header3);
        printf("%s", header4);

        /* Nested for loop to output the contents of the entire array
(including the average) */

        for (row=0; row < num_students; ++row)
        {
              for(col = 0; col <= num_grades + 1; ++col)
                    printf("%8i  ",student_array[row][col]);
              printf("\n");
        }  /*  end for loop  */

        printf("%s", header4);
        getchar();

} /* end main     */
```

Output of the above program:

University Student Report

Student ID	Exam 1	Exam 2	Exam 3	Exam 4	Exam 5	Avg
12233	87	93	98	77	69	85
21033	35	91	89	57	77	70
13456	78	99	98	98	99	94
22020	55	96	77	88	99	83

```
10103    65    91    79    87    76    80
18456    79    89    98    88    91    89
29920    54    66    79    88    92    76
22223    39    81    69    87    73    70
16456    68    85    98    87    94    86
16020    89    69    77    68    96    80
=================================================
```

Pretty good?

So, there you have it. I have nothing else to add regarding multi-dimensional arrays. Keep in mind that these types of arrays are used much less frequently as single-subscripted arrays. (I know. I was relieved when I was told that too!)

We will be using arrays (single-subscript) in the our discussion of functions in the next 2 lessons. This will ensure that you have a good grasp of the subject! The program above is a very good example of how to copy, and compute row and column calculations with a two dimensional array.

Please post questions to the discussion board as needed.

Introducing: Functions!

Most computer programs that solve real-world problems are much larger than the programs presented so far in this course. The best way to develop and maintain large programs is to construct it from smaller pieces called "modules". Programs that are modular are more manageable, as they are developed piece by piece. This also results in code pieces that are more reusable.

One of the common programming standards in industry today specifies that each source file (filename.c) should have between 50 and 100 lines of code. Anything more than 100 lines is way too many! So, how do we write code that is hundreds-of-thousands of lines-of-code (such as word processor programs, games, etc.) Again, we do this by organizing our code in smaller pieces and putting them together when we compile and link.

Modules in C are called "functions". (In other languages, these modules are sometimes called subroutines, or subfunctions.) C Programs are typically written by combining new functions that the programmer writes with the pre-packaged functions available in the Standard C Library.

We are quite familiar with using (calling) some of the functions in the Standard C Library, such as **printf()** and **scanf()**. Now, we will become familiar with writing our own functions to support our programs. Actually, we have already written code for function **main()** in each program that we have developed so far. We will now learn how to create other functions, as well as the mechanics of using these other functions within our program.

3 Motivations for Using Functions

1. Program development is more manageable since you develop it function-by-function. (Building blocks.)
2. Software reusability is encouraged by using existing functions as building blocks for new programs. (Avoid reinventing the wheel.)
3. To avoid repeating code within a program.

We will now learn how to write our own customized functions.

A Simple Function

Let's use program 2.2 as an example for creating our own functions*. I have made some slight modifications*, as shown below:

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

```
        printf ("Programming is fun.\n ");
        printf ("And programming in C is more fun!\n");

        getchar();
}
```

Now, instead of using **printf** in function **main** to output our information, lets create our own function to output the message. This function will be called **display_message**.

Function main would now look like:

```
#include <stdio.h>
void main(void)
{
        display_message();

        getchar();
}
```

Voila! That's it! Well, not really. Function main is now calling a function (display_message) that does not exist in the Standard C Library. So, the compiler and/or linker will have a problem with this. So, we need to provide the code for function display_message, since it is not provided for us. The code for such a function would look like:

```
void display_message (void)
{
        printf ("Programming is fun.\n ");
        printf ("And programming in C is more fun!\n");
}
```

Voila! The new function! (Don't worry about the **void**s at this time. We will discuss them later in this lesson.) Now, we have all of the pieces that we need. We just need to *put them together* . You have 3 options when it comes to physically placing function code in your program:

1. You can put your function code *above* ***main()*** in your source file.
2. You can put your function code *below* ***main()*** in your source file.
3. You can put your function code in a *separate* source file.

These will all be discussed at a later time. For now, we will use option #1. That is, we will put our function code *above* function **main** in the same source file.

So, if for example, our source file was called: fun.c, it would contain the following:

```
#include <stdio.h>

void display_message (void)
{
        printf ("Programming is fun.\n ");
        printf ("And programming in C is more fun!\n");

} /* end display_message/*


void main(void)
{
        display_message();

        getchar();

}  /* end main/*
```

Voila! That's all there is to it. The output of the above program would look like:

```
Programming is fun.
And programming in C is more fun!
```

Now, what if our function main looked like:

```
void main(void)
{
        display_message();
        display_message();

        getchar();
}
```

What do you think the output would be? You guessed it:

```
Programming is fun.
And programming in C is more fun!
Programming is fun.
And programming in C is more fun!
```

Let's discuss some of the concepts introduced so far. In this example, within function **main()**, the line:

```
display_message();
```

is referred to as the *function call.*

In function **display_message**, the first line is called the *function header.* The function header has 3 components.

1. The return data type (**void** - in this example).
2. The name of the function (**display_message**- in this example)
3. The arguments or information that the function needs to do the job (**void** - in this example).

The 3 parts of our function header are shown below and are discussed in more detail later in this lesson:

```
 1              2               3
void    display_message  (void)
```

Notice no semicolon at the end of the function header. This should not be strange to you, as you have been typing in the function header for function **main** all semester! Our function header is always followed by a *begin brace.* Then the body of the function block is specified. Then, the function block ends with an *end brace*.

Now, what if function main looked like:

```
void main (void)
{
        int   x;

        for (x = 1;   x <= 5;   x++)
        {
                display_message();

        } /* end for loop/*

      getchar();

} /* end main/*
```

What do you think the output would be? You guessed it!

```
Programming is fun.
And programming in C is more fun!
Programming is fun.
And programming in C is more fun!
Programming is fun.
And programming in C is more fun!
Programming is fun.
And programming in C is more fun!
```

Function **display_message**was called within the **for**loop, which ran 5 times. So the body of function **main** was executed 5 times. Hence the output repeated 5 times.

Note: It is a common standard to design function **main** so that it contains no real processing at all. That is, that function **main** is simply a series of calls to other functions which will do all the work. So, for example, if we had a program which prompted the user for a set of numbers, then calculated the average, then output the results, our function **main** might look like:

```
void main (void)
{
        get_numbers();
        calc_average();
        display_results();

        getchar();
} /* end main/*
```

get_numbers, **calc_average**, and **display_results**would all be functions that you, the programmer would create. All of the processing would be done in these functions.

Now that we have visited some examples of functions, it is a good time to perhaps use an analogy as to what is really going on here. This should be helpful to those of you who are not quite sure yet. We actually do this sort of thing in real life. It is called "delegating". For example, if you worked in an office and needed to make 20 copies of a document, you could do that task yourself, *or* you could delegate that job to someone else. Once you delegate the job to someone else, you don't always know *how* that person did the job, only that he/she did it correctly for you. The same is true for functions. The calling function has no information of *how* the function will do the job, but this information is not necessary to the calling function.

The next topic discusses *function arguments*. Function arguments give functions the opportunity to be more useful, more generic, and more reusable.

Want to argue? You've come to the right place!

And you thought it was only you. Even functions have arguments! Another term for *function* ***arguments*** is *function* ***parameters***. Both terms are interchangeable. Arguments or parameters provide additional information to the function so that the function can do its job. Function arguments are not new to us. We use them every time we call function **printf** or function **scanf**. When we call function **printf**, we always supply 1 or more values to the function as listed below:

1. The information to be printed.
2. The results to be displayed (if any).

This makes functions more flexible (unlike our **display_message** function discussed in the previous topic). We will now learn how to define functions that accepts arguments.

First, let's use function display_message as an example, and spruce it up with an argument. We will modify the function so that it expects to receive "the number of times to output the message" as an argument. So, if we passed the number 4, for example, to function display_message, then the message would be displayed 4 times. If we passed the number 7, it would be displayed 7 times. Let's first work on the function call. The function call, from function main, must now "pass" the number of times to function display_message. Function main is shown below, passing the number 4 to function display_message.

```
#include <stdio.h>

void main (void)
{
    display_message (4);
}
```

When we pass information as arguments to functions, we enclose them in parentheses, following the function name. If there is more than 1 argument, then the information is listed between the parentheses, separated by commas.

Now, let's make the necessary change to function display_message to handle this information being passed to it. This is shown below:

```
void display_message (int num_times)
{
    int  x;

    for (x = 1;   x <= num_times;   x++)
    {
        printf ("Programming is fun. ");
```

```
        printf ("And programming in C is more fun!\n");
    }

} /* end display_message/*
```

Whoaaaa! What was that?!?! That, my dears, is what we call a function argument! It's not so bad. Let's think about it a bit.

First of all, the output of the above would be:

Programming is fun. And programming in C is more fun!
Programming is fun. And programming in C is more fun!
Programming is fun. And programming in C is more fun!
Programming is fun. And programming in C is more fun!

The value that is passed to the function from the calling function (in this case, function **main** is the calling function); this value must be "held" by the function when the function is called. Once function **display_message** is called, the value 4, in this example, would be stored in variable **num_times** while the function is executing. Notice that we are actually declaring variable **num_times** right in the function header! That is how we show that the function will be receiving an argument. We provide a **data type** and a **variable name** right in the function header, to hold the value being passed.

Now, if a function does not receive any arguments, then we use the word **void** in between the parenthesis, instead of arguments. (I knew you knew that one!)

Notice how we used variable **num_times** in the body of the function block. Pretty nifty huh? Now, function **main** could call function **display_message** a variety of times, outputting the information however many times it desires. So, if function main looked like:

```
void main (void)
{
    display_message (3);
    printf ("Hi There!\n");
    display_message (2);
}
```

The output would be:

Programming is fun. And programming in C is more fun!
Programming is fun. And programming in C is more fun!
Programming is fun. And programming in C is more fun!
Hi There!

Programming is fun. And programming in C is more fun!
Programming is fun. And programming in C is more fun!

Now, let's make this program a bit more flexible. How about if we allow the user to specify how many times that the info will be output? Yah! Let's try that. Function display_message will not change at all. It will still output the information as many times as is passed to it. The change would be made in the calling function, function main. Function main will now prompt the user for the number of times to output the info. And the value that the user types in will be passed to the function. The program would look like:

```
#include <stdio.h>

void display_message (int num_times)
{
    int x;

    for (x = 1;   x <= num_times;   x++)
    {
        printf ("Programming is fun. ");
        printf ("And programming in C is more fun!\n");
    }

} /* end display_message/*

void main (void)
{
    int users_choice;

    printf ("Enter the number of times to output message: ");
    scanf ("%i", &users_choice);

    display_message (users_choice);

   getchar();

} /* end main/*
```

Woaa. Check that out! How flexible is that! The output of the above program might look like:

Enter the number of times to output message: 2
Programming is fun. And programming in C is more fun!
Programming is fun. And programming in C is more fun!

Fantastic huh?!?! I know that you can hardly contain yourself, but hold on, we've only just begun! The next example also illustrates passing information to a function. This particular function will calculate and output the square of a number that is passed to it. (Remember, the square of a number is the number multiplied by itself.) So, if the calling function passes the number 5, the function will output the number 25. Function **main** will prompt the user for a number to be squared, and will then pass that number to function: **square.**

A sample output of such a program might look like.

Enter number to be squared: 6
The square of 6 is 36.

The code for the above would look like:

```
#include <stdio.h>

void square (int num)
{
     int  num_squared;

    num_squared = num * num;

    printf ("The square of %i is %i.\n", num, num_squared);

} /* end square/*

void main (void)
{
    int  num_to_square;

    printf ("Enter number to be squared: ");
    scanf ("%i", &num_to_square);

    square (num_to_square);

    getchar();

} /* end main */
```

Notice, in function **main**, the value that the user enters is stored in variable **num_to_square**. However, when function **square()** is called, only the *value* of the variable **num_to_square** is passed over to the function, and this *value* (6 in our example) is temporarily stored in variable **num**, in the function.

Programs 8.4 and 8.5 on pages 131 and 133 illustrate this concept of passing of arguments to functions. They are somewhat complex in that they are math related. However, don't let that get you down. Try to focus on the function call, and don't stress over the algorithm. If you have the time, please review both of those programs, so to firm up what we have covered so far. If you have any questions regarding arguments and functions, please post them in the discussion area for this week.

That's all I have for function arguments. We will be using function arguments throughout the remainder of this course. So, you will certainly get plenty of practice! The next section discusses "returning function results". That is, sometimes a function will return a calculated value to the calling function. That way, the calling function can do whatever it wants to do with that value. This is another technique for adding flexibility to functions.

I'd suggest that you get a cup of coffee at this point, but I know you are so wide-awake from all the excitement of functions, that you probably don't need one. How about a decaf? :)

"return" - The most common word used in a function.

Let's get right to it!

Functions are even more flexible if they can "return" a calculated value to the calling function. That is, if a function calculates some value (as in our previous example of function **square**), the calling function might want to use that calculated value too! So, somehow, the function has to get the calculated value back to the calling function. We do this using the *return data* portion of the function header.

Recall the format of a function header:

```
  1          2             3
void  function_name (void)
```

Part **1** is the "return data type". That is, it will indicate the data type of the data being returned. If there is no data being returned to the calling function, then this return data type is **void**. (I know that is confusing, but hopefully it will be cleared up soon!)

Let's use the example introduced in the previous section: function **square.** Let's assume that we do not want function **square** to output the squared value of the number, but we instead want the calling function (function **main**, in our example) to *receive* the calculated squared value, and do what it wants with it. In our example below, function **main** will simply output the squared value. Let's set this sort of thing up.

First of all, function main will now look something like:

```
#include <stdio.h>

void main (void)
{
    int squared_value, value;

    printf ("Enter number to square: ");
    scanf ("%i", &value);

    squared_value = square(value);  /* a variable in main is needed to store
the returned value from the function */

    printf ("%i squared is %i.\n", value, squared_value);

    getchar();
}
```

Notice above that there is an assignment operator to the left of the function call to function **square**. Also, variable **squared_value** is being assigned the *result* of function square. This is exactly how we store values that are returned from functions. We put the function call on the right side of an assignment operator, and a variable to hold the returned value on the left side of the assignment operator. Now, it is very important that we are aware of the data type of this variable that is being assigned the return value of the function. In this example, the data type of variable **squared_value** is **int**. So, the return data type of the function must be **int**!

Let's now put the code together for function **square**, which will return the squared value of a number:

```
int  square (int num)
{
    int  num_squared;

    num_squared = num * num;

    return   num_squared;   /* the number is being returned to main and
needs to be assigned to a variable in main */

}
```

Notice the new code introduced above. Instead of **void** as the return data type, it is now: **int.** That indicates that this function will return a value, and the data type of that value is **int.** In order for a function to actually return a value, it must contain the **return** statement. This statement consists of the reserved word **return**, followed by the value to be sent back to the calling function. This value must be of the same data type that is specified in the return data type area of the function header. In our case, variable **num_squared** is of type **int**, which is exactly what we need.

So, the return data type is of type **int**, which makes the data type of the entire function type **int**. For this reason, we can assign the function call in function **main**, to a variable of type **int** (variable **squared_value** in this example). The data types match! Halleluiah!

The output of the above program might look like:

```
Enter number to square: 5
5 squared is 25.
```

Like a **printf** statement, the calculations can be performed right in the **return** statement. That is, the above code could have looked like:

```
int  square (int num)
{
    return   num * num;
}
```

The above code would do the exact same thing as the previous code for function **square**. That is, it will return the calculated squared value of a number that is passed to it. With this particular method however, we no longer need variable **num_squared**. The programming style is up to you. Both techniques are acceptable.

Note that if a function *does not* return a value to the calling function, we use **void** as the return data type in the function header, and the function body *does not contain* a **return** statement.

Now, let's modify function **main** a bit. What if function **main** looked like:

```
#include <stdio.h>

void main (void)
{
    int squared_value, x;

    for (x = 1;  x <= 5; x++)
    {
        squared_value = x * x;
        printf ("%i squared is %i\n", x, square_value );
    }

   getchar();

} /* end main*/
```

Assume function square did not change at all. The output of the above program would look like:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

I knew you knew that one. Now, since you're so smart, let's abbreviate function **main** a bit more, so that the function call to function **square**, is performed right in the **printf** statement! The new abbreviated function **main** would look like:

```
#include <stdio.h>

void main (void)
{
    int x;

    for (x = 1;  x <= 5; x++)
        printf ("%i squared is %i\n", x, square(x) );

    getchar();

} /* end main*/
```

Notice, variable **squared_value** is no longer needed! This works because function **square** returns an integer value to the caller. In this case, **printf** is the caller, and the integer value will go right in the correct spot in the output. Again, this particular coding style is your choice.

Personally, I prefer NOT to put the calculation in the **printf** statement. I think it is clearer when it is a separate line of code. However, this is very common, and you will see it a lot.

Program 7.6 on page 127 is a modification of program 7.5. It illustrates the concept of returning data to the calling function. Please refer to that program as another example of this concept.

Function Prototypes - AKA Function Declarations.

Does any of that make sense to you? Prototypes. Declarations. Well, I know you've heard the word declaration before. Yes, we declare variables in C. Variables must be declared before they are used. Remember that one? Well, the same is true for functions! Also, the term prototype, relates to a sort of example. A model. Well, that too applies to functions.

How you ask? Well, hold on partner, and I'll fill you in!
(It's 3:50 in the morning right now, and I'm feeling quite numb. I'm not responsible for anything that I might say at this point.)

The function prototype is what the compiler uses to determine if you are setting up your functions properly. That is, it uses prototypes to ensure that you are defining your function correctly, and that you are calling your function correctly. The compiler refers to the function prototype to check the following:

1. Is the return data type correct (if any) in both the function call(s) and function header?
2. Is the number of arguments (if any) consistent in both the function call(s) and the function header?
3. Is the data type of the arguments (if any) consistent in both the function call(s) and the function header?
4. Is the order of the arguments (if any) consistent in both the function call(s) and the function header?
5. Is the name of the function consistent in both the function call(s) and the function header?

Now, pay close attention, if the function header is located above **main**, then you DO NOT NEED a function prototype. The compiler will create one based on the function header. However, it is a good habit to get into creating function prototypes, even if you put your functions above **main**.

What does a prototype look like? Well, it looks an awful lot like the function header. In fact, it is exactly the same as the function header, without any variable names listed, and with a semicolon at the end. For example, our function header for function square looks like:

```
int  square (int  num_squared)      /* function header/*
```

Our function prototype for function square would look like:

```
int  square (int);                  /* function prototype. /*
```

Now, the prototype above tells the C compiler that function square:

1. Returns something of type **int**.
2. Requires 1 argument.
3. The argument is of type **int**.
4. The function name is **square**.

So, now, if any function tries to call function **square**, and violates any of the above, the compiler will give you an error. For example, if you tried to call function **square** from within **main** as follows:

```
void main (void)
{
    float   result;
    int     num = 5;

    result = square (num);

    getchar();

}/* end main */
```

The compiler would not allow this because you are trying to store the return value of function **square**, which is of type **int**, into a variable of type **float**.

Also, the following function call would not work either:

```
num = square (10.2);
```

This wouldn't work because **square** is expecting data of type **int** as an argument, and we are sending data of type **float**.

Now, the big question is: where do these prototypes go? Well, the book puts them right in function **main**. In the variable declaration section.

This is NOT a good place. It is very confusing, and difficult to read. They actually belong right below your **#include** commands. At the top of your file. Actually, the *include files contain the prototypes for the functions in the Standard C Library*! Now you know what is in those files!

Also, the book erroneously puts variable names near the data types in the prototypes. This is completely unnecessary. In fact the compiler ignores those variable names. The compiler is only interested in the number, order and data types of the values coming over.

So, let's put together the entire file which will contain the program to calculate the square of 5 numbers. It would look like:

```
#include <stdio.h>

/* Function Prototypes/*

int square (int);

/* Begin function main/*

void main (void)
{
    int squared_value, value;

    printf ("Enter number to square: ");
    scanf ("%i", &value);

    squared_value = square(value);

    printf ("%i squared is %i. \n", value, squared_value);

    getchar();

} /* end main/*

/* Begin function square/*

int square (int num)
{
    int  num_squared;

    num_squared = num * num;

    return   num_squared;

} /* end square /*
```

Notice that I put function **main** first here. That is, before function **square**. Now that I declared my function prototype, I am allowed to do this. If I didn't declare a prototype here, the compiler would give me an error because it starts compilation at the top of the file, and if it doesn't come across either a function header, or a function prototype, BEFORE a function call, then the compiler does not know if the function call is correct, and it will give you an error.

That's about all I have for prototypes. Please read pages 134 - 135 in textbook for additional information.

A function calling a function, calling a function, etc.

...and so on and so on and so on... This can get ugly. I'm not sure you're up for it. Okay. NOW's a good time for that coffee. Make this one caffeinated!

Actually, I'm scaring you for no reason. This is pretty straightforward.

Let's say that we wanted to write a function which calculates the "Cube" of a number. (A number multiplied by itself 3 times.) And let's say that we didn't want any negative results. That is, if we prompted the user for a number to cube, and the user entered -3, then we don't want to see: -27, instead we want to see: 27.

For this problem, we will have function main, which will call function cube, which will call function abs_val (to get the absolute value of a number). The final cubed value will be returned to function main, where it will be output from within main.

Here goes:

```
#include <stdio.h>

/* function prototypes/*

int  cube (int);
int  absolute_value (int);

/* Begin main/*

void main (void)
{
    int  num, cubed_num;

    printf ("Enter a number to cube: ");
    scanf ("%i", &num);

    cubed_num = cube (num);

    printf ("%i cubed is %i\n", num, cubed_num);

} /* end main; /*

/* Begin function cube/*

int cube (int value)
{
    int  abs_cubed_value, cubed_value;
```

```
    cubed_value = value * value * value;

    abs_cubed_value = absolute_value (cubed_value);

    return abs_cubed_value;

} /* end cube/*

/* begin absolute_value/*

int absolute_value (int number)
{
    if (number < 0 )
        number = -number;

    return number;

} /* end absolute_value/*
```

A sample run of the above program would look like:

```
Enter a number to cube: -5
-5 cubed is 125
```

What is happening here? I'll try to summarize:

Function **main** prompts the user for a number to cube. The user enters the number **-5**, which is stored in variable num in main.

This value **-5** is passed to function **cube**. Function **cube** stores that value **-5** in its own variable called **value**. Function **cube** then multiplies value 3 times, and stores that result (**-125**) in variable **cubed_value**. Next, function **cube** needs to ensure that this value is positive, so it calls function **absolute_value**, passing it the result stored in variable **cubed_value**.

Function **absolute_value** stores the signed cubed result (**-125**) in variable **number**, and checks to see if that value is negative. If so, it makes it positive (**125**). Otherwise, it leaves it alone. It then returns the value stored in variable **number** (**125**) back to the caller, which happens to be function **cube**.

Function **cube** then stores the result of function **absolute_value** (**125**) in variable **abs_cubed_value**. Function **cube** then goes ahead and returns the value stored in **abs_cubed_value** (**125**) back to the calling function, which happens to be function **main**.

Function **main** then takes the return value of function **cube** (**125**) and stores it in variable **cubed_num**. It then goes ahead and outputs this calculated value. Whew!!!

Now that we've done all of this, I will say that all of this processing could have been done within function **main**. However, as mentioned in the first topic in this lesson, we do not normally put very much processing in function **main**. Also, each of these functions are now reusable components, which can be used by other functions within your *same program*; or by *other programs* you write, or even by programs that *other people write*. The **abs_value** function itself, as well as the **cube** function, for example, can be used for many different reasons, for a variety of different programs. Why re-invent the wheel?!?

That's about all I can do for you at this point. For more information about functions calling functions, calling functions, please see textbook.

Next we combine the 2 concepts recently covered: functions and arrays.

Passing array data to functions.

Let's get right to it. This topic focuses on a commonly used and very necessary programming technique of *passing array data to functions*.

Like simple variables, we can also pass *the value of an array element*, or even *the entire array* to a function. We will handle each of these 2 issues separately. Passing the value of an individual array element is easy, passing an entire array however, is a whole new ballgame! (I didn't say it was hard. I just said it was a new ballgame.)

First, let's discuss passing the value of an individual array element to a function.

Passing the value of an individual array element to a function.

Passing the value of an individual array element to a function is no different than passing the value of a simple variable to a function. In either case, the *value* of a variable is being passed to a function. The function does not know (or care) where the value came from. It does not need to know that the value was stored in an array, instead of in a simple variable. So, the function itself is the same, whether the value is coming from a simple variable, or from an array. It is the *function call* that will look different when the value comes form an array element.

All you need to do is pass the *array element* to the function when making the function call. For example, let's take a look at function **square** from last week's lesson. Function **square** is passed a value to be squared, and it returns the value's squared result to the calling function (function **main** is the calling function, in our case). Let's make a slight change, and instead of calling function **square,** and passing it the value of a *simple variable*, let's pass to it the value of an *array element*. In fact, let's pass to it all of the elements of an array, 1 element at a time.

So, function **square** remains the same as it did last week (because, as I noted, function square does not care where the value is coming from).

```
#include <stdio.h>

int  square (int num)
{
    int  num_squared;

    num_squared = num * num;

   return   num_squared;
}
```

Notice, that function **square** above is the same exact function from last week's notes. Now, let's modify function **main** slightly so that it will call function **square**, and it will pass to it the contents of an array, 1 element at a time. In function **main** below, the array (**values[5]**) is initialized during declaration, so that the values of the array are setup for us (just to make it easier for the example). So, the contents of the array after declaration are as follows:

Element	Value
values[0]	2
values[1]	5
values[2]	7
values[3]	8
values[4]	10

```
void main (void)
{
    int  squared_value, values[5] = {2, 5, 7, 8, 10};
    int  i;

    for ( i = 0; i < 5; i++)
    {
        squared_value = square( values[i] );
        printf ("%i squared is %i.\n", values[i], squared_value);
    } /* end for loop*/

} /* end main*/
```

Notice above that I am passing the value of each element of the array to the function (shown in **purple** above.)

The output of the above program would be:

```
2 squared is 4
5 squared is 25
7 squared is 49
8 squared is 64
10 squared is 100
```

FYI: The function **prototype** of function **square** would be:

```
int square (int);
```

In the above example, the original contents of the values array are unchanged. What would happen however, if the for loop in function main looked like:

```
    for ( i = 0; i < 5; i++)
    {
        printf ("%i squared is ", values[i] );

        values[i] = square( values[i] );

        printf ("%i.\n", values[i] );
    } /* end for loop*/
```

Now, the contents of the **values** array have been modified to contain the squared values. Also noticed, with this algorithm, I needed to *split* the **printf** statement into 2 parts, because after the function call to function **square**, I would have lost the original contents of the array, and I needed the original value as part of the output statement.

Now, we are ready to tackle the more interesting technique of passing an entire array to a function.

Passing an entire array to a function.

Let's modify the program above so that function **square** will square the *entire array*. Function **main** will pass the entire array to function **square**, and function **square** will loop through the array and calculate and output the squared values.

Function **main** will simply call function **square**, and pass to it the array values. The code in function **main** is now reduced to a few lines as follows:

```
#include <stdio.h>

void main (void)
{
    int  values[5]  {2, 5, 7, 8 20};

    square ( values );

} /* end main*/
```

Notice the function call above (shown in **blue**). It is different from the previous example in 2 ways. First, function **square** does not return a value in this example, because it will calculate and output the squared value from within the function itself. In this example, function **main** does not need any information returned from function **square**, so no need to return any. Second, you see the function call to function **square** contains just the name of the array (**values**). This is because when we want to pass the entire array to a function, we simply pass the array name.

Note: Passing the name of the array is actually passing the *starting memory address* of the array, so that the function can access the data. When we pass an address to a function (as we are doing when we pass the array name), this is known as "Pass by Reference". When we pass a value (as in all prior examples), this is known as "Pass by Value". When we pass by reference, the function will have direct access to the original data, and can change the original contents from within the function. When pass by value, the function does not have access to the original data, and cannot change it. The details of the information mentioned in this paragraph is not necessary for this C Programming Part 1 course, however I thought your might want a head's up, as it is discussed in Part 2. This is your first introduction to "Pointers"!

Now, as you can imagine, we need to modify function square to handle this array. Function square looks like:

```
void square (int n[ ])
{
    int i, result;

    for (i = 0;  i < 5;  i++)
    {
        result = n[i] * n[i];
        printf ("%i squared is %i\n.", n[i], result);
    } /* end for loop*/

} /* end function square*/
```

Wow. Check that out! I have introduced a new concept here. That is, how a function is defined to receive an array as an argument. In the function header you see:

void square (int n[ ])

The return data type is **void,** the function name is **square**, and the argument is an "integer array". That is how we specify an integer array: **int n[].** It tells the C compiler that the function is receiving a *starting address of a set of integers*. Wow! If we left the brackets out, the compiler would assume that the function was receiving a simple value (pass by value). Also notice that there is *no number* inside the brackets in the function header. If we did put a number in there (which the textbook tends to do by mistake), the compiler simply ignores it! It is useless to the program to stick a number between the brackets in the function header.

Also, function **square** will use variable name **n** to access the contents of the array. Variable **n** takes on the exact same memory location as variable **values**. It is as though a single memory address now has 2 variable names! So, within

function **square**, we can directly access the data using array name **n**. It is the exact same data as if we were accessing array **values** from within function **main**.

With that in mind, if function **square** made a change to any part of array **n**, then when the function terminates, and control is brought back to function **main**, the contents of the array values would be modified.

If, for example, function **square** looked like:

```
void square (int n[ ])
{
    int i;

    for (i = 0;  i < 5;  i++)
    {
        printf ("%i squared is ", n[i] );
        n[i] = n[i] * n[i];
        printf ("%i\n", n[i]);
    } /* end for loop*/

} /* end function square*/
```

... then the entire contents of array **values** would be modified from within function **square**!

Also, the *function prototype* for function **square** would look like:

void square (int [ ]);

The prototype is always exactly like the function header, but without any variable names.

One final observation to be made here is the issue of *function reusability*. The above function **square** is not very generic, and so not very reusable. It only works on any integer array that is 5 elements long. What if our program had 2 integer arrays, one that had 5 elements, and another that had 10 elements, and we wanted to square them both. Would we need to create 2 different functions to do this? No! Of course not! What we do, is make function **square** a bit more generic by passing not only the starting address of the array to the function, but also the array *size.* The size of the array would be passed as the second argument, by value! Will wonders never cease!

So, for example, say our following program called function square 2 times, with 2 different arrays. Try to step through the following code:

```
#include <stdio.h>
void  square (int n[ ], int);     /* function prototype*/

void main (void)
{
    int   values[5] = {2, 5, 7, 8, 10};
    int   nums[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

    square (values, 5);
    printf ("\n");
    square (nums, 10);

} /* end main*/

void square (int n[ ], int size)
{
    int i;

    for (i = 0;  i < size;  i++)
    {
        printf ("%i squared is ", n[i] );
        n[i] = n[i] * n[i];
        printf ("%i\n", n[i]);
    } /* end for loop*/

} /* end function square*/
```

The output of the above program would be:

```
2 squared is 4
5 squared is 25
7 squared is 49
8 squared is 64
10 squared is 100

10 squared is 100
9 squared is 81
8 squared is 64
7 squared is 49
6 squared is 36
5 squared is 25
4 squared is 16
3 squared is 9
2 squared is 3
1 squared is 1
```

Notice how much more generic and reusable function **square** is now.

Take a look at program 7.12 on pages 144 & 145. Function **main** calls function **sort** to sort an array in ascending order. Please do not spend too much time on the details of the **sort** function. It is a bit complex. What I want you to focus on is how function **main** calls function **sort**. Also note that the function prototype for function sort is located *within* function **main**, and is not necessary at all since function **sort** is located *above* function **main**.

TYPO ALERT: Finally, before you look at program 7.13, I want to point out a typo. You are not required to review this program, as it passes a 2-dimensional array to a function, which is pretty tough to grasp in Part 1 of this course. However, I wanted to mention that the function prototypes *should* be located *above* **main** (below the **#include** statement).

That's about all I have for introducing you to the concept of passing arrays to functions. This topic is discussed at length again in Part 2 of this course, with additional examples. This topic was simply to introduce the concept to you. Hopefully it was somewhat "graspable".

Next, we will discuss the various properties of variables.

Variable Properties

This section introduces some of the different properties of variables, which include:

- Global vs. Local variables
- Automatic vs. Static variables
- Register variables

It is important that you are aware of these concepts, but do not worry if you don't understand them fully at this time.

Global vs. Local Variables:

I have 2 words for you: **Go local**. Using global variables is not accepted as a proper programming practice in C.

With that said, I guess I should at least explain what global variables are, so you know how to spot them, and to avoid using them!

First of all, even before we get going, I just want to point out that if acceptable, the use of global variables are meaningless unless you are working with functions. With that said, let's start with local variables.

Local variables are declared within a particular function (eg., **main(), square(),** etc.). We have been using local variables throughout this semester. The value of a local variable can only be accessed from within the function that it is declared. The "availability" of a variable is known as its *scope.*

With a program that uses local variables, if a function other than the function that declares a particular variable needs that variable, then we must *pass* that value to the function as an parameter. This is the preferred way to design and develop our code. It keeps the program modular, and less likely to lose track of variables.

For example, the following program calls function **max** to determine the maximum of 2 numbers entered by the user. The function receives the 2 values as arguments (by value), and returns the maximum of the 2 values entered. Function **main** then outputs the maximum of the 2 numbers entered. After looking at the program, I will point out a few things regarding local variables.

```
#include <stdio.h>
int  max (int, int);   /* function prototype */

main()
{
    int  num1, num2, maximum;
```

```
    printf ("Please enter 2 numbers: ");
    scanf ("%i%i", &num1, &num2);

    maximum = max (num1, num2);

    printf ("\nThe maximum of %i and %i is %i.\n", num1, num2, maximum);

    return 0;

} /* end main */
```

```
int  max (int n1, int n2)
{
    int  largest;

    if (n1 > n2)
        largest = n1;
    else
        largest = n2;

    return largest;

} /* end max */
```

OR:

```
int  max (int n1, int n2)
{
    if (n1 > n2)
        return n1;
    else
        return n2;
} /* end max */
```

Note: Function **max** could be written in either of the 2 ways above, or even as shown below using the **conditional operator** (remember that?!):

```
int max (int n1, int n2)
{
    return  (n1 > n2)  ?  n1  : n2;
}
```

A sample run of the above program might look like:

Please enter 2 numbers: 25 35
The maximum of 25 and 35 is 35.

Pretty cool huh?

Now, back to local variables, in our example above, variables **num1**, **num2** and **maximum** are *local* to function **main.** Variables **n1**, **n2** and **largest** are *local* to function **max**. Function **main** knows nothing about variables **n1**, **n2** or **largest**. Function **max** knows nothing about variables **num1**, **num2**, and **maximum**. But that is okay. In fact, that is exactly what we want. That way, function **max** cannot modify any variables that it does not have access to. Keep in mind, in the real world, you will be coding up some functions for a program, and others will be

coding up other functions for the same program. You don't want to give the other programmers the power to easily change your variables. It leaves room for too much error.

If function **max** above had the following statement in it:

```
n1 = 100;
```

as shown below in **blue**...

```
int  max (int n1, int n2)
{
    int  largest;

    if (n1 > n2)
        largest = n1;
    else
        largest = n2;

    n1 = 100;

    return largest;

} /* end max */
```

Would that statement affect variable **num1**? No. Variable **num1** in function **main** would be unchanged. Although **n1** contains the *value* that was stored in variable **num1**, it only contains a *copy* of the value. Hence: "**pass by value**". Even if we named the 2 arguments in function **max** as follows:

```
int  max (int num1, num2)
{
    int  largest;

    if (num1 > num2)
        largest = num1;
    else
        largest = num2;

    num1 = 100;

    return largest;

} /* end max */
```

and if we set **num1** to be 100 in function **max** (as shown above), it would *still* have no effect on variable **num1** back in **main**. The 2 **num1** variables are different. **main** only has access to the **num1** variable declared in function **main**, and **max** only has access to the other **num1** variable declared in function **max**. This is what we mean by the scope of local variables. Recall: the value of a local variable can only be accessed from within the function that it is declared.

Now, off to global variables.

Global variables are declared outside of any function (usually after the **#include** statement, and before any other functions). Any function can directly access the value of a global variable (as well as change it)! **This is quite dangerous.**

Some programmers will use global variables simply because they are lazy (no offense). They use global variables instead of passing values to functions as arguments. It is easier to use global variables then to pass and return values. For example, if we were the type of programmer that didn't like to pass and return variables to/from functions, our program above, using global variables **num1**, **num2**, and **maximum** as shown below, the program might look like:

```
#include <stdio.h>

int  num1, num2, maximum;   /* declare global variables. */
void max (void);            /* function prototype */

main()
{

    printf ("Please enter 2 numbers: ");
    scanf ("%i%i", &num1, &num2);

    max ();

    printf ("\nThe maximum of %i and %i is %i.\n", num1, num2, maximum);

    return 0;

} /* end main */

void max (void)
{
    if (num1 > num2)
        maximum = num1;
    else
        maximum = num2;
} /* end max */
```

Notice how all of the functions have access to the global variables? No need to pass information around? This looks great doesn't it? Well, like candy, it might look great, but it's not that good for you. The same is true here. Below lists a few reasons why we shouldn't use global variables.

DO NOT USE GLOBAL VARIABLES BECAUSE:

1. Functions are no longer general purpose. They are tied to the global variables, and require them to be declared globally in any program which uses the function.

2. Program readability is reduced because it is not clear which values the functions are using. Also, the function call does not indicate what the function needs as inputs, and what the function is returning to the caller.

3. Reusability is reduced.

All in all, over the years, it has been proven that using global variables puts your program at risk for all sorts of problems. They are not a good practice.

Have I harped on that enough? **Remember, -10 points for each global variable in your programs!**

Next, we discusses automatic vs. static variables. Don't be afraid, both of these types of variables are acceptable. :-)

Automatic vs. Static Variables:

Automatic and Static variables apply to local variables only. All of the variables we have worked with in this class have been considered "automatic" variables. Let's begin with those.

Automatic Variables:

- Local variables are considered automatic by default.
- They are automatically created each time the function is executed.
- They are destroyed when the function completes.
- The keyword "auto" may precede the variable declaration, but it is not needed.
- The value of an automatic variable has when a function finishes executing is guarenteed not to exist the next time the function is called.

Static Variables:

- The value of a static variable retains its value once the function completes.

- The value that the static variable has upon leaving a function will be the same value that the variable will have the next time the function is called.
- Static variables are initialized only once at the start of the overall program execution, and not initialized again each time the function is called.
- If a static variable does not have an initial value, it will be initialized to zero.
- The keyword "static" precedes the static variable.
- Use static variables if you want a variable to retain its value from 1 function call to another (example: a function that counts the number of times it has been called).
- Use static variables if the function uses a variable whose value is set once and never changes.

Sample program using automatic and static variables:

The following program is pretty useless, however it illustrates the concept of automatic and static variables. Try to step through the program and see if you can come up with the correct results (also shown below).

```
#include <stdio.h>
void auto_static (void);   /* prototype */

main()
{
    int i;    /* could have been:  auto int i;

    for (i = 1;  i <= 5;  i++)
        auto_static();

    return 0;

} /* end main */

void auto_static (void)
{
    int          auto_var = 1;       /* could have been:  auto int auto_var = 1; */
    static int  static_var = 1;      /* this line will only be executed once.       */

    printf ("Automatic Variable = %i,  Static Variable = %i\n", auto_var,
static_var);

    auto_var = auto_var + 1;     /* could have been: auto_var++;    */
    static_var = static_var + 1;   /* could have been: static_var++; */

} /* end auto_static */
```

A sample execution of the above program would be:

Automatic Variable = 1 Static Variable = 1
Automatic Variable = 1 Static Variable = 2
Automatic Variable = 1 Static Variable = 3
Automatic Variable = 1 Static Variable = 4
Automatic Variable = 1 Static Variable = 5

Notice how the automatic variable keeps getting reset to 1 each time the function is called, while the static variable retains its value once the function is called again. Even though variable **static_var** is initialized to 1 in the function, that line of code is actually only executed the first time that the function is called. All times after that, that particular line of code is ignored. If we didn't initialize variable **static_var** at all, it would have been initialized to zero by default.

Lastly, we introduce Register Variables.

Register Variables:

C allows programmers to select certain variables to be placed in one of the computers "high-speed" hardware registers. These registers process the data much more quickly than variables that are stored in memory. If you precede your variable name with the key word **register**, then you are simply making the *request* for one of the high-speed registers. You are not guaranteed to get one! Each computer has a different number of these registers available, and depending what is running along with your program, you there may not be one available to you.

When would you use register variables? Well, how about for a loop control variable for a loop that is running hundreds of thousands, or millions of times? How about for a counter variable that is within a loop?

A register variable declaration might look like:

register int index;

You can declare any simple variable to be a register variable (that is, variables of type int, float, char, long, double, short, unsigned, etc). Arrays cannot be declared as register variables.

Note: This information is not found in chapter 7. I think it might be found later on in the textbook, as it does not necessarily relate to functions. However, I thought this would be a nice place to at least introduce to the concept.

The next topic (**and final topic for the course**) discusses: recursion. I've saved the worst for last. :)

Recursion: A function that calls itself.

I can't believe that we are here. Our last lesson. How time flies when you're having fun. Where has the semester gone? I know you are disappointed too that this is your last lesson, but hey, there's always C Programming Part 2!!! :)

I normally try to stay positive, but this is as bad as it sounds. (Now you're in trouble!) Recursion is not for most programmers. In fact, I have rarely coded up a function recursively. You usually don't have to. Almost every problem that you solve recursively, can be solved non-recursively (a.k.a. - iteratively). The reason I am explaining recursion is because other programmers do code recursively, and you will have to a least recognize it and understand what is going on.

With that said, let me explain what this is all about.

So far, all of our programs have been written in a disciplined, hierarchical manner. Functions call one-another, and return to the calling function, in an organized, methodical, straightforward approach. Well, if you'd like, you can choose to program in a less hierarchical manner. More of a spastic manner, using recursion. A recursive function is a function which calls itself, either directly or indirectly via another function. It is very complex.

Let me start with a common programming algorithm that is written recursively. It is the problem of determining the factorial of a number. Recall, from one of our past math classes, that the factorial of a number is that number multiplied by the number before it, and that number multiplied by the number before it, etc.

So, for example:

5! (5 factorial) is equal to: 5 * 4 * 3 * 2 * 1 = 120.

4! is equal to: 4 * 3 * 2 * 1 = 24.

If you look at the 2 factorial calculations above, you can see that 5! is really 5 * 4! (In English: 5 factorial is really 5 times 4 factorial.) And 4! is really 4 * 3!. Etc. We can sort of see "recursion" going on here. (More like regurgitation, if you ask me.)

Anyway, to code up this problem recursively, it would look like:

```
#include <stdio.h>
long factorial (int);   /* function prototype*/

void main(void)
{
   int     num;
```

```
    long  num_fact;

    printf ("Please enter a number: ");
    scanf ("%i" &num);

    num_fact = factorial (num);
    printf (" %i factorial is %li.\n" , num, num_fact);
              ;
}/* end main*/

long  factorial (int n)
{
    long result;
    if (n == 0)
         result = 1;
    else
        result = n * factorial (n - 1);    /* here is the function call to itself.*/

    return result;

} /* end function factorial*/
```

A sample output of the above program might look like:

```
Please enter a number: 6
6 factorial is 720.
```

This looks harmless enough, but it really is very confusing. Too confusing to explain online. Try to step through the program. Also, read through any textbook you might have on recursion. It might help.

I want to point out that, as with all recursive functions, this same problem can be solved non-recursively (iteratively). All of our functions in this class have solved problems iteratively so far. That is, through iterations of a loop, the problem eventually gets solved.

Using the same function **main** above, we can solve the factorial problem iteratively instead, as shown below:

```
long  factorial (int n)
{
    long result = 1;
    if (n > 1)
    {
        for (i = n; i > 1; i--)
```

```
        result = i * result;
    }
    return result;

} /* end function factorial*/
```

Ahhhh. Much better. If you step through this code, you will see that it is much easier to digest.

Here are some interesting points regarding iteration vs. recursion:

(This is not in the textbook).

Iteration vs. Recursion.

1. Iteration uses a repetition structure (loop).
   Recursion uses a selection structure (**if** - **else**).

2. Iteration repeats itself through the repetition structure (loop).
   Recursion repeats itself through repeated function calls.

3. Iteration terminates when the loop continuation condition becomes false.
   Recursion terminates when the base case is recognized (the body of the **if**).

4. Iteration modifies a counter until the counter assumes a value that makes the loop
   continuation condition false.
   Recursion keeps producing simpler versions of the original problem until the base case is
   reached.

5. Iteration occurs within a function, so the overhead of repeated function calls and extra memory
   assignment is omitted.
   Recursion repeatedly invokes the overhead of function calls Each recursive call causes another
   copy of the function' s variables to be created. This is expensive in both processor time and
   memory.

Well, there you have it. Why you would ever code recursively, I do not know. However, I do know that there are many algorithms already coded recursively out there. Particularly those that have to do with dynamic data structures such as binary trees and linked lists. But, you don't have to worry about that for a few more classes :) The good news is that if you take Programming in C Part 2, the starting point is functions, so you another dose of this good stuff !

Well, that's all I have. The end of the semester. I hope you enjoyed it, and learned a lot along the way. Give yourself a pat on the back, and a round of applause. You deserve it!