C PROGRAMMING GUIDE

This document provides a basic introduction to the C/C++ programming language as is needed to program the ARDUINO micro-controller.

CHAP 1 - INTRO

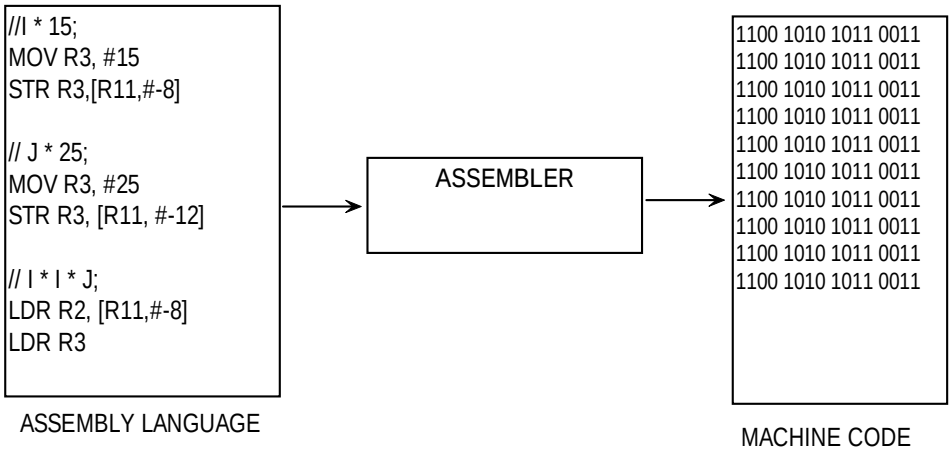
1.1 INTRO TO COMPUTER PROGRAMMING

A computer is a **machine** that **manipulates data** (usually numbers) according to a set of instructions. The set of instructions is called a **computer program**. There are many different programming languages (C, C++, Python, Java, Fortran, etc.). Each language has its own SYNTAX or form. C and C++ are probably the most important of the languages to learn for engineers.

A computer program is just like a recipe for food preparation. It provides enough information for the person to make the food properly, making no assumptions about what the reader knows. Similarly a good computer program is a set of step-by-step instructions that the computer is to follow. Computer programs can do many things – word processing, graphics, and controlling robots or machines!

When writing programs you must remember that a computer CANNOT THINK, it merely follows the instructions you give it. The smallest error in a computer program can cause it to not compile, or to not function as intended.

There are LOW-LEVEL & HIGH-LEVEL languages. High level languages are written more like how people talk, so programmers tend to write in high-level languages. But computers can only understand low-level languages. So high-level languages must be COMPILED, or translated, into a set of instructions that the computer understands. Compilers generally translate for a specific processor. C and C++ are compiled languages.



C is a block-structured procedural language. C++ is an extension of C that allows for OBJECT-ORIENTED programming. Object-oriented programming allows for the creation of so-called CLASSES. Both C and C++

must be COMPILED. Other languages are "interpreted" (like BASIC or Python). Compiled programs are faster and more efficient. After your program is compiled, it must be RUN or EXECUTED.

Don't think of computers as smart. They are simply machines that do what we program them to do. They cannot do what we INTENDED them to do if we did not program them that way. The art and challenge of programming is to translate our INTENT into an appropriate computer program.

1.2 VARIABLES

At the most basic level, a computer mainly performs mathematical computation. Math is behind everything a computer does (making sounds, producing images, or controlling a robot)

To do math, a computer must store numbers (values) in containers called VARIABLES. Variables have a name, a type, and they store a value.

1.2.1 NAMING VARIABLES

There are rules for how to name variables (don't memorize 1 thru 4)

1. Variable names cannot start with a number (**`1var`** not allowed).
2. Variable names CAN start with an underscore (**`_1var`** is ok).
3. Variable names cannot have a space. Use an underscore instead (**`var name`** not ok, var_name ok).
4. Variable names are case sensitive. The following variables are all different (**`VarX`**, **`varx`**, **`varX`**).

Give variables meaningful names. If pin 5 will be used to light an LED, name it "**`LEDpin`**" not just "**`x`**".

```
// "LEDpin" is better than "x"
```

1.2.2 VARIABLE DATA TYPES

You must **DECLARE** a variable before you can use it. When declaring a variable, you specify the variable TYPE (e.g., int or float), which tells the computer to set aside a certain amount of memory to store the value. The computer needs to set aside sufficient memory for the values that may end up being stored there. Below is a summary of some different data types and what they can store.

```
int LEDpin;          // declaring a variable
```

The most typical types will be INT and FLOAT, which hold integers and decimal numbers, respectively

```
int i = 1;
float f = 1.23;
```

Below is a table that shows some different data types, the number of bits used in memory, and the range of values that they can hold (you will not be tested on these values, so do not memorize).

Bits	Type	Range
8	boolean	True or false, 0 or 1 (needs 8 bits to be addressable)

8	byte	0 to 255 (or -128 to 127)
16	int	-32,768 to 32,767
16	unsigned int	0 to 65,535
32	long int	-2,147,483,648 to 2,147,483,647
32	unsigned long int	0 to 4,294,967,295
16	short	-32,768 to 32,767
32	float	-3.4028235E+38 to 3.4028235E+38 (6-7 decimal precision)
64	double	-1.7E+308 - 1.7E+308 (double precision floating point, 15-17 decimal precision)
64?	long double	-1.2E+4932 to 1.2E+4932 (long double prec floating point)

https://en.cppreference.com/w/cpp/language/types

Select the appropriate data type that will hold the value that you can possibly have. For instance, an "int" is no good if the value that can go into it can get bigger than 32,767. If the value can become bigger than 32,767, then "int" won't work. Unsigned data types are used when they are only holding positive numbers (no negative numbers). This may be appropriate if you are storing a time value (time is always > 0).

1.2.3 ASSIGNING VALUES TO VARIABLES

The equal sign ("=") is called an **ASSIGNMENT OPERATOR**.
It is used to place a values or expressions on the RIGHT into the variable (container) on the LEFT.

```
LEDpin = 5;
```

Assignment operator works right to left. It can be placed within an expression.

```
a = b = c = d = e = 100;   //e = 100 is assigned first
val = 5 + (r = 9 * x);     //this is valid:  r = 36, & then val = 41
```

Do not think of the "=" to indicate equality as in mathematics. The statement below is valid in C, but makes not sense in math:

```
val = 2;
val = val + 5;       // val now equals 7
```

Note you can declare a variable and assign a value all at once (note - C does NOT automatically assign zero to variables upon declaring them)

```
int LEDpin = 3;
int pinX;                  //pinX is not initialized to 0 here
```

1.3 ELEMENTS OF A PROGRAMMING LANGUAGE

Most computer programming languages have the following elements (don't memorize this list):

1. Data types - int (integers), floats (store decimal values), char (store letters)
2. Mathematical operations - add (+), subtract (-), multiply (*), divide (/), modulus (%)
3. Loops (while, for) - to repeat portions of code

- while loops - loop until a condition is met
- for loops - loop a certain number of times

4. "if" statement - to execute code only if a certain CONDITION is met
   eg, only do this IF that is true or false.
5. Relational operators (= =, <=, >=, <, >) - use with "if" to test how variables compare to each other
6. Logical operators (AND = &&; OR = ||; NOT = !) – combines relational tests
7. Arrays – are matrices of numbers where several values are grouped under the same name
8. Strings – an “array” of “characters” (or letters)
9. Input/output - for Arduino, usually to the Serial Monitor, or to its I/O pins

1.4 PROGRAM DESIGN

AVOID REPETITION

In programming there should be very little repetition. If there is, the code can probably be improved (for e.g., use variables, loops, or functions).

USE VARIABLES

Try to avoid just typing values (numbers). For example, if you are using pin 3 to control an LED, you may be inclined to just type 3 whenever a pin # must be specified in your program. You may need to specify this in many places. If you specify a value more than once, it’s a good sign you should use a variable instead (e.g., "LEDpin = 3;". This way, if you decide to switch the pin to, say, 4, you can do it in one line of code. Think of variables as containers that hold a value.

USE FUNCTIONS

A computer program must be planned and designed. Break up your code into logical steps. Whenever possible, group steps into so-called FUNCTIONS – sections of code that perform one specific (and usually simple) task. Functions are usually just a few lines of code. Functions should be designed to be re-usable.

Books on C indicate that the `main ( )` function is required and is always the first function executed. With Arduino, it is there, but we cannot see it.

See the section on functions.

1.5 READABILITY AND COMMENTING

C is sequential and FREE-FORM. Sequential means it executes statements one after another. Free-form means open spaces (and hard returns) are ignored by the compiler. You could write a program on a single line if you wanted to, but it would be very hard to read, so it would be poor practice to do so.

READABILITY is important in programming. Easily readable computer programs have:

1. Appropriate "white" (or blank) space including blank lines (hard returns)
2. Includes descriptive commenting

Briefly COMMENT your code to remind yourself and explain to others what you are doing. Comments are ignored by the computer, but are important to readers of your program, including yourself.

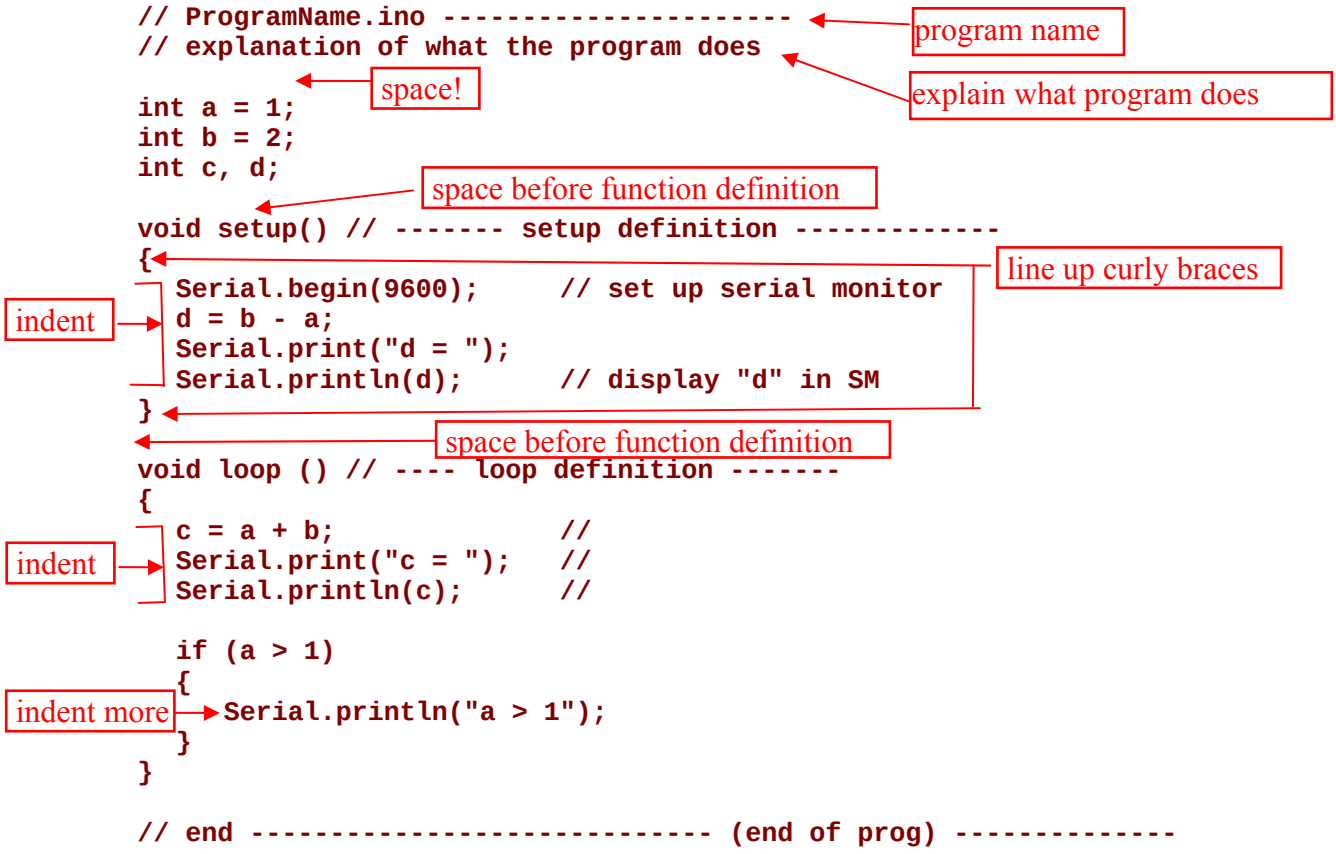
There are single-line and multiple-line comments (per below). You cannot nest a multi-line comment inside another multi-line comment. Insert comments "as you go" (as you program), don't try to add them later on.

```
// single line comment

/*  multiple
   Line
   Comment
*/
```

Commenting and white space make it easier to remember how your code works and also how to DEBUG your code. DEBUG means to find errors in the code.

Below is an example of a simple ARDUINO program written in C++. It shows appropriate commenting and white space to enhance readability.



1.6 PARENTHESES, BRACES, BRACKETS - AND SEMI-COLONS

In C++ there are parentheses ( ) and curly braces (or just "braces") { } and brackets [ ]. They are different and used for different purposes (more on this later). Never try to interchange them. These items must always come

IN PAIRS. For example, every open brace ({) must include a closing brace (}). The same is true for brackets and parentheses. This is a common error causing C programs to not compile.

Everything between curly braces is called a BLOCK. A block is a sequence of statements or commands.

Executable C++ statements end with a semi-colon (;). This tells C++ when the statement ends.
Example (notice the semi-colons).

```
int i = 5;
int j = 11;
callFunction();
```

1.7 INTRO TO FUNCTIONS

Computer programs are best if broken up into FUNCTIONS. Functions are blocks of organized reusable code that perform a single related action. A function is often not much more than a couple of statements. Functions helps you modularize code and prevent repetition. Functions have the same naming rules as variables. A function can be distinguished from a variable in that it always has parentheses after the name. The parentheses may be empty, or they may have something in them (more on this later). Functions are DEFINED and they are CALLED (by other functions).

The bit of code below illustrates how a program might look from a high level after being broken up into functions. This sequence of code CALLS 3 different functions in sequence.

```
getLetters();
alphabetize();
printLetters();
```

A function is DEFINED using the syntax below. Note there is no semicolon at the end of the first line.

```
returnVarType FunctionName(varType Argument1, varType2 Argument2)
{
  //block of code defining what the function does
}
```

We will talk more about functions later on.

TO KNOW FOR EXAM:
What is a computer? What is a computer program? What are low vs. high level languages?
Is C++ compiled? What does compiling do?
What are variables? Naming rules for variables (and use meaningful names)?
Data types (int, float) hold what?
(No need to memorize value ranges for types)? What does declaring a variable do?
How do you assign a value to a variable (what is the "=" sign?)? (Elements of a language - no)
Programming guide (avoid repetition, use variables, readability, comments)
How to do single and multi-line commenting? () [] {} (always in pairs). What's a block?
Semi-colons,
Functions intro (how to define, call)

CHAP 2 - ARDUINO PROGRAMMING & IDE

Refer to the ARDUINO HANDOUT for specific information about Arduino.

Arduino programming uses the C and C++ programming language. The Arduino software is called Arduino IDE, which is free and open source. The Arduino IDE includes a text editor, compiler, libraries of pre-written functions, and lots of example code. Arduino programs are called SKETCHES. Sketches are written, verified (compiled), and then uploaded to the Arduino board. Arduino boards can only store one program at a time.

2.1 FIRST ARDUINO PROGRAM

All Arduino programs have 2 functions: setup ( ) and loop ( ). A function is a set of programming code that is designed to do one specific thing (more on this later).

The function **`setup()`** is executed first and is executed one time.
The function **`loop()`** is done after setup(), and it executes over and over in a loop "forever".

Thus, any code to be executed once is generally placed in **`setup()`**.
Code that is executed repeatedly is placed in **`loop()`**.

When we program the Arduino we will write a set of statements and place them in the braces of the setup and loop functions. In doing so, we are defining what the setup and loop functions do. The setup and loop functions are being called by some other function that is unseen by us. We don't worry about that - we just define what setup and loop do.

```
void setup ()
{
  //"block" or set of statements that define what setup() does
  statement1;
  statement2;
}

void loop ()
{
  //"block" or set of statement that define what loop () does
  statement3;
  statement4;
}
```

TO KNOW FOR EXAM:
What 2 functions are required in all Arduino programs?
What do they do?

CHAP 3 - BASIC TOPICS

TOPICS: PREPROCESSOR DIRECTIVES, MATH OPERATORS, PRECEDENCE, COMPOUND ASSIGNMENT, COMPOUND OPERATORS, INCREMENT/DECREMENT, TYPE CASTING

3.1 PREPROCESSOR DIRECTIVES

Sometimes you will see the following at the beginning of a C program

```
#define AGELIMIT    21          //
# include <Servo.h>       //include servo lib to access its fcns
```

The syntax (syntax means form) is below. Note there is no equal sign and no semi-colon (NO GOOD: # define **AGELIMIT = 21;**):

```
#define ARGUMENT1   argument2
```

The **# define** allows you to define a "constant". Common practice is to capitalize all letters. This statement tells the compiler to look for the "ARGUMENT1" and replace it with the "argument2". It is essentially a compile-time find-and-replace.

The **# include** allows you to "link" to other files, often a so-called h file. The h file refers to a library of pre-written functions. When including the h file, you allow your program to access the functions in that library. This statement also uses no semi-colon (Don't do this! **# include <Servo.h>;**).

The syntax is:

```
# include <filename>            // no semi-colon (;)
#include "filename"             // here's another way to do # include
```

3.2 ARITHMETIC & MATH OPERATORS

Obviously, the computer will be used to do math. Here we see how we get the computer to process math and the syntax of math expressions.

The primary math operators are indicated below

- ***** multiplication
- **/** division
- **+** addition
- **-** subtraction
- **%** modulus (or remainder)

These are so-called binary operators because they operate on two values.
The spaces do not matter. So the following two expressions are the same to the computer. However, the first statement is preferred because it's easier to read.

```
a = 6 + 2;
a=6+2;
```

Unary operators (such as "-") operate on a single value.

```
a = -25;
```

The forward slash (/) always divides. It produces integer division if both values it operates on are integers. Integer division works by truncating any remainder or modulus (it does NOT round). Thus,

```
b = 7/4;            // b = 1, not 2 or 1.75
c = 7.0/4.0;        // c = 1.75
```

Examples

```
a = 7/4;            // a = 1
b = 7.0/4.0;        // b = 1.75
c = 10 + 3;         // c = 13
d = 9 * 2.0;        // d = 18.0
e = 10 % 3;         // e = 1 (remainder)
```

3.3 ORDER OF OPERATION (PRECEDENCE)

What does a equal for the expression below?

```
a = 2 + 3 * 2;      // a not = 10, it equals 8
```

The order of operations is

1. Multiplication, division, and modulus
2. Addition and subtraction.

There are actually many more precedence levels in C++ (about 15).

Use parentheses to alter the order of operations. Expressions inside of parentheses take a higher precedence than multiplication and division. Parentheses can be nested one inside the other.

```
a = (2 + 3) * 2;                // now a = 10!
b = 5 * (5 + (6 – 2) + 1)  // (6-2) is first, then (5+(6-2)+1), & B = 50
```

3.4 COMPOUND ASSIGNMENT & OPERATORS

COMPOUND ASSIGNMENT

Use compound assignment to update a variable

```
salary = salary * 1.2;
```

This statement takes the value of salary, adds 20%, and then assigns the new value back to salary again. In math, this expression is invalid, but in programming it is fine!

COMPOUND OPERATORS

These are short-hand ways of doing compound assignment.

```
                    // meaning
a += 500;           // a = a + 500;
b -= 50;            // b = b – 50;
c *= 1.2;           // c = c * 1.2;
d /= .50;           // d = d/.50;
e %= 7;             // e = e % 7;
```

3.5 (++) INCREMENT AND DECREMENT (- -) OPERATORS

These are efficient as they compile directly into their assembly language equivalent.
You cannot increment or decrement an expression.

```
//Example               Meaning          or
i++;                    i = i + 1;       i += 1;
i--;                    i = i - 1;       i -= 1;
```

3.6 TYPE CASTING

Sometimes a math expression in C mixes different data types. C will generally convert the smaller type into the larger type.

```
int     bonus = 50;
float   salary = 1400.50;
float   total;

total = salary + bonus;         //bonus is temporarily converted to a float
```

While C is usually successful at automatic conversion, it may have problems with mixing unsigned variables with variables of other types (e.g., unsigned int with float). Type casting can solve this problem.

```
// int age = 20;
// float factor = 0.10;
age_factor = (double) age * factor;
```

TO KNOW FOR EXAM:
Preprocessor directives - syntax, what they do
Math operators (know which to use, eg, "*" is multiplication)
Order of operation (and how to change it using ( ))
Compound assignment, operators

Increment, decrement
Type casting

CHAP 4 - RELATIONAL OPER'S & IF

TOPICS:
RELATIONAL OPERATORS (= = , <=, etc.)
IF, ELSE, ELSE IF

4.1 RELATIONAL OPERATORS

Sometimes we don't want to execute every statement in a program every time. Instead we may want to execute some statements under **SPECIFIC CONDITIONS**. Relational operators help with this. They TEST the relationship between different variables or expressions (i.e., they COMPARE values).

Below is a table of relational operators

```
//Operator     Description

==             // Equal to (not one "="!!)
>              // Greater than, arrow points to the SMALLER value
<              // Less than
>=             // Greater than or equal to (don't use "=>")
<=             // Less than or equal to (don't use "=<")
!=             // Not equal to
```

Note that **==** is a test of equality and **=** is an assignment operator. Don't confuse or interchange them!

CONDITIONS

Relational operators test the relationship between variables or expressions. Together they form a CONDITION that produces a TRUE or FALSE result.

True = non-zero (often 1)
False = 0

For example, the following is TRUE (**15 < 20**)...
The following is FALSE (**35 >= 55**)...

You can put expressions into the condition test (e.g., **(x == (2*y))**)
Note the relational operator "= =" tests LEFT to RIGHT.

Conditions can get tricky. The code below is valid. The condition below evaluates as FALSE!

```
if (10 == 10 == 10)
```

```
//the condition is False!  The left "10 == 10" is true so = 1.
//then 1 == 10 is the last test, false!
```

4.2 if STATEMENT

Relational operators test a CONDITION and that condition is incorporated into an "if" statement. Together they form a **DECISION** statement that tests the relationship and makes a decision about which statement to execute next based on the result of that decision. If the condition is TRUE, perform the block of statements in braces. Otherwise if the condition is FALSE, do NOT execute the block of statements. Note there is no semi-colon in the "if" line. Curly braces may be omitted if the block has only one statement.

```
if (condition)
{
   Statement 1;      // execute if CONDITION is TRUE
   Statement 2;      // execute if CONDITION is TRUE
}
Statement 3;         // execute if CONDITION is TRUE or FALSE (not in the block)
```

In the template above, if "condition" is TRUE, then the sequence of statement execution would be:

Statement1
Statement2
Statement3

If "condition" is FALSE, then the statements executed would be:

Statement3

Here's an example for Arduino:

```
int age = 15;

if (age < 21)
{
   Serial.print ("You are under 21");
}
Serial.print ("Print this all the time");
```

Note in the template code above that Statement3 is executed whether "condition" is TRUE or FALSE. If you only want to execute Statement3 if "condition" is FALSE, you should use an "else" (more on this below).

4.3 else STATEMENT

The **else** statement never appears without the "if" statement, so the if and else go together.
Use **else** when you want to execute code only if the condition is FALSE. The code in the else block is NOT executed if the condition is TRUE.

The format is as follows

```
if (condition)
{
   // Block of statements to execute if "condition" is TRUE
   Statement1;
   Statement2;
}
else    // no condition here, as all remaining possibilities happen here
{
   // Block of statements to be executed if "condition" is FALSE (or 0)
   // statements you don't want to execute if condition is TRUE
   Statement3;
}
Statement4;
```

In the code above, if the condition is TRUE, the sequence of statement execution is:

Statement1;
Statement2;
Statement4;

If the condition is FALSE, the sequence is:
Statement3;
Statement4;

Hopefully you can see why the "else" is needed. Statement4 is executed no matter what the condition is. But statement 3 is only executed if the condition is FALSE.

NOTES

You can use if to compare characters (but it compares the ASCII values).
You can nest if statements.
You cannot compare character strings or arrays of character strings directly with relational operators.

4.4 else if STATEMENT

If you want to test more than one condition, use "**`else if`**". Each "else if" has its own (condition).

You can have many else if's but the else must appear LAST if it is used.
Once an else if succeeds, none of the other else if's will be tested

1. QUESTION – SO ORDER OF ELSE IF's MATTERS?
2. DO THE ELSE IF'S NEED TO BE EXCLUSIVE? (IE – CAN 2 BE TRUE?)

The syntax is:

```
if (condition)
{
   //code if condition is TRUE
}
else if (condition2)
```

```
{
  //code if condition2 is TRUE
  //if there's overlap in condition2 & 3, then 1st else if runs...
}
else if (condition3)
{
//code if condition3 is TRUE
}
else    // this appears last, has no condition, & runs if condition is FALSE
```

EXAMPLE

```
int a = 100;

if( a == 10 )
{
  Serial.print("Value of a is 10");
}
else if( a == 20 )
{
  Serial.print("Value of a is 20");
}
else if( a == 30 )
{
  Serial.print("Value of a is 30");
}
else
{
  Serial.print("Value of a does not match");
}

Serial.print ("Exact value of a is ");
Serial.print (a);
```

TO KNOW FOR EXAM:
Relational operators (==, <, >, etc.)
If, else, else if
How to use the above in a program, or identify result of a given program that has these

CHAP 5 - LOGICAL OPERATORS

TOPICS: LOGICAL OPERATORS (AND, OR, NOT)

5.1 LOGICAL OPERATORS

We may need to test our data in a more complex way. We can do this by combining relational operators with LOGICAL OPERATORS. This combination allows us to create more powerful data-testing statements. For instance, I may wish to test whether a variable is greater than some value, AND also less than another value.

Logical operators are below

Operator	Meaning
&&	AND
||	OR
!	NOT

Truth Table for AND (both sides of operator must be TRUE for the result to be TRUE)

False	&&	False	=	False
False	&&	True	=	False
True	&&	False	=	False
True	&&	True	=	True

Truth Table for OR (only one side or the other must be TRUE for the result to be TRUE)

False	||	False	=	False
False	||	True	=	True
True	||	False	=	True
True	||	True	=	True

Truth Table for NOT (the opposite relation is produced)

NOT True = False
NOT False = True

USING LOGICAL OPERATORS

Below is an example of the syntax of compound logical operators

```
if ((50 > 30) || (90 < 81))
{
   // (50 > 30) TRUE ..... (90 < 81)  FALSE
   // TRUE OR FALSE --> TRUE
   // the overall condition above TRUE
}
```

Use the NOT operator (!) sparingly as it tends to confuse.
Note: `!(var1 == var2)` is same as (var1 != var2).

It is suggested that you use a limit yourself to 2 relational tests at a time (although you can do more). Instead, us nested if's.

5.2 C++'s LOGICAL EFFICIENCY (skip)

C attempts to be efficient & will not always interpret the full expression if it is not needed.

Example:

```
if ((7 < 3) && (sales < 15) && (15 != 15)) ...
```

C looks at the first condition (7 < 3), which is False and does not look further because False && (and) anything else remains False.

Example

```
//YearTest.ino --------------------
//Determine if it is a Summer Olympics year, US census yr, or both


void setup ()
{
  int year = 2000;

  if ((year % 4) == 0 && (year % 10) == 0)
  {
     Serial.print("Both Olympics and US Census!");
  }
  else
  {
     if ((year % 4) == 0)
     {
        Serial.print("Summer Olympics only");
     }
     else
     {
        if ((year % 10) == 0)
        {
           Serial.print("US Census only");
        }
        else
        {
           Serial.print("Neither");
        }
     }
  }
}

void loop ()
{ // nothing here}
```

5.3 PRECEDENCE OF LOGICAL OPERATORS

As with math operators, logical operators have an order of precedence

1. Multiplications first
2. Relational operators (< >)
3. AND logical operator (&&)
4. OR logical operator (||)

TO KNOW FOR EXAM:
Logical operators (and, or, not)
Precedence - what is it?
How to use the above in a program, or identify result of a given program that has these

CHAP 6 - FOR, WHILE LOOPS

TOPICS: FOR LOOPS, WHILE LOOPS

6.1 for LOOPS

Sometimes we wish to repeat a section of code a certain number of times. We can use a "for" loop to repeat a section of code until a certain value is reached.

SYNTAX

```
for (start expression; test expression; count expression)
{
   //block code
}
```

EXAMPLE

NO semicolon is placed at the end of the "for" line. Braces may be omitted if the block code has only one statement. Generally an index variable is included in the "for" statement. It includes when it starts, how it increments (the count expression), and when it ends (test expression). The "for" loop tests at the top of the loop. If the test expression is FALSE when the "for" loop begins, the statements in the block will never execute.

The example below outputs numbers 1 through 4 in the serial monitor.

```
//Example of code that outputs numbers 1 thru 4 in the serial monitor
for (i = 1; i <= 4; i++)
{
   Serial.println(i);
}
```

The output would be

```
1
2
3
4
```

6.2 NESTED "for" LOOPS

You can "nest" a "for loop" inside of another for loop. If so, the inner loop executes completely before the outer loop's next iteration.

```
for (i = 1; i<=2; i++)
{
  for (j = 1; j<=3; j++)
  {
        Serial.print(i);
        Serial.print(", ");
        Serial.println(j);
  }
}
```

The output of the code above would be:

```
1, 1
1, 2
1, 3
2, 1
2, 2
2, 3
```

6.3 while LOOPS

You may wish to repeat a section of code as long as a certain CONDITION is met. Use a "while" loop for this.

SYNTAX

```
while (test expression)         //no semicolon
{
  //block code here executes over & over while the test expression is TRUE
}
```

The test expression usually contains relational and possibly logical operators that, together, result in either a TRUE or FALSE condition. As long as the test expression is TRUE (non-zero), then the block will execute. If the condition is FALSE (or zero), then the block is skipped. If the block contains only one statement the braces are not needed (it's good practice to do it anyways). The statements in the block should change the variables in the test expression. Otherwise the test expression will never change and the while loop repeats forever in an infinite loop (generally not desired).

EXAMPLE

```
while (test expression)         //no semicolon
{
```

6.4 do-while LOOPS

This loop is similar to the while loop but the relational test occurs at the bottom of the loop. This ensures the body of the loop executes at least once. Here the while statement ends with a semi-colon.

```
do
{
  //block code here
```

```
}
while (test expression);  //put the semicolon at the end here!
```

6.5 for VS. while

Both "for" and "while" loops repeat sections of code. So what's the difference?

while loops continue until a certain condition is met.
for loops continue until a certain value is reached.

6.6 if vs. while

"if" does not repeat sections of code.

while loops may repeat a section of code **many times**.
"if" code may or may not execute code, but if it does execute, it does so only **ONCE**.

TO KNOW FOR EXAM:
For, while loops (how use, difference)
How to use the above in a program, or identify result of a given program that has these

CHAP 7 - FUNCTIONS

TOPICS:
FUNCTIONS, VARIABLE SCOPE (global vs. local), AUTOMATIC vs. STATIC variables,
PASSING AND RETURNING VALUES

7.1 FUNCTIONS

The best computer programs are organized and modular. It's best to break up your programs into smaller routines instead of having one long piece of code. Functions help us modularize our code, and they are designed to be re-usable. Functions are blocks of code that are executed when the function is called.

Functions have
1. A name (same name rules as for variables)
2. The name is followed by a set of parentheses ( ). The parentheses may or may not be empty.
3. The body of the function (following the parentheses) enclosed in braces { }.

SYNTAX

Below is the syntax for defining a function. Note there is no semicolon on the first line where the function name is specified. The statements in the braces DEFINE what the function does.

```
return_type functionName (arg1, arg2)
{
  Statement1;
  Statement2;
}
```

In Arduino you always have the setup and loop functions. But you CAN write your own function if you want

```
void setup()
{
  myFunction();     //call the custom fcn myFunction()
}

void loop()
{
}

void myFunction()   // define the fcn myFunction()
{
  Statement1;
  Statement2;
}
```

7.2 CALLING AND RETURNING FUNCTIONS

A function is executed when it is called (generally by another function). When calling a function, the program takes a DETOUR from the "calling function", executes the code of the "called" function (based on however it was defined), and then it returns to the "calling function.

The partial code below shows what it might look like.

```
void setup ( )   // -------------------------
{
   getLetters();
   alphabetize();
   printLetters();
}

void getLetters()   // define function "getLetters()" ------------
{
   //block of statements defining "getLetters()"
}

void alphabetize()  // define function "alphabetize()" ---------------
{
   //block of statements defining "alphabetize()"
}


void printLetters() // define function "printLetters()" -----------------
{
   //block of statements defining "printLetters()"
}
```

In the code above, setup() is the calling function. Inside of setup(), the function getLetters() is called. The program detours to the definition of getLetters() and executes the code that defines getLetters(). It then returns to setup() which then calls the alphabetize() function. The code detours to alphabetize() and executes its code. It then returns again to setup() which then calls printLetters(). Again it detours to printLetters( ), executes its code and returns once more to setup().

Look at the setup ( ) function. It is easy to see the sequence of execution because the code was broken up into smaller functions!

7.3 VARIABLE SCOPE

Once we start working with functions, we must understand VARIABLE SCOPE. There are GLOBAL and LOCAL variables. Each has a different scope.

GLOBAL variables are declared OUTSIDE any function's definition and they are visible to all functions.

LOCAL variables are declared INSIDE a function can only be seen and changed within the function in which the variable was defined. Local variables go away when their block ends.

Two variables, local to 2 different functions, can have the same name. C++ considers them different variables.

Most programming books will stress that you should only rarely use global variables. This is because most computer programs are developed by teams of programmers. What would happen if 2 different programmers used the same global variable name for 2 different purposes? It would be a problem. Using local variables keeps the programming modular and prevents conflicts. In our case, programming a micro-controller, we don't need to be as diligent.

```
int GlobalVariable = 5;

void setup ( )   // -------------------------
{
   int LocalVariable = 3;
   float anotherLocalVar = 2.1;
   int i = 500;
   delay(i);               // calling the delay() fcn
}

void loop ( ) // ---------------------------
{
   int i = 2;          // different than the other int i
}
```

7.4 PASSING VARIABLES

You may have noticed that all functions have parentheses. What are those for? Sometimes they are empty - other times they have something in them.

Sometimes a calling function must PASS a local variable (data, a variable or a value) to a function it is calling (the receiving function). In these cases, you place the variable in the parentheses of both the calling and receiving functions. Global variables do not need to be passed. All functions can "see" these variables anyways.

SYNTAX, EXAMPLE

The example below shows the delay( ) function being passed a variable (which happens to have a value of 500)

```
int timeDuration = 500;
delay(timeDuration);
```

Often more than one variable must be passed

```
digitalWrite(pinNumber, HIGH);
```

7.5 AUTOMATIC VS. STATIC VARIABLES

Local variables are either AUTOMATIC or STATIC.

An AUTOMATIC local variable is ERASED once the function ends. If the function is called a second time, it will have no memory of the value that it computed previously.

If you need the function to "remember" that local variable, you must declare it as a STATIC variable.

Local STATIC variables are NOT erased when their functions end. Note all global variables are static.

Note that static does not mean global. Local static variables remain local. That is, they are still only visible to the function in which they are declared (i.e., in which they are local)

```
void fcnName()
{
static int lastValue;
}
```

7.6 FUNCTION RETURN VALUES

A function may also return a value to the calling function.

The Arduino functions setup ( ) and loop ( ) return nothing to the function that calls them. You can tell because the term "void" is in front of their definitions.

As you have seen with our prior work, we can write our own functions in Arduino if we want. And we can have these functions return a value to the calling function if it's appropriate.

Note that is makes no sense to return a GLOBAL variable since the calling function already has access to it.

The calling function must have a use for the return value.

SYNTAX

Here the syntax for defining a function that returns a value to the calling function

```
void Calling_Fcn
{
  data_type VarName;

  VarName = Called_function_Name(var1, var2);   // call the fcn
}

data_type Called_function_Name (dataType1 Variable1, dataType2 Var2)
{
  // block code here
  dataType localVar;
  // msc code

  return(localVar);       // must have this "return" statement
}
```

EXAMPLE

```
void setup ( )   // -------------------------
{
  float x1 = 4.2;
  float x2 = 5.7;
  float x3 = 6.2;
  float localAvg;
  localAvg = getAverage(x1, x2, x3);
}
```

```
float getAverage(float val1, float val2, float val3)
{
   float avg;
   avg = (val1 + val2 + val3)/3;
   return (avg);
}
```

TO KNOW FOR EXAM:
Functions (defining, calling, returning) - how to do these
Variable scope (local, global)
Automatic vs. static variables
Returning values
How to use the above in a program, or identify result of a given program that has these

CHAP 8 - OTHER TOPICS

TOPICS: ARRAYS

8.1 ARRAYS

A regular variable holds a single value. ARRAYS allow us to store multiple values under the "same" name. Those values are stored in contiguous memory locations. Individual locations are referenced using the array name and an index (e.g., ArrayName[index]).

CHAP 9 - PASS BY ADDRESS, POINTERS

TOPICS:

9.1 PASSING BY VALUE OR PASS BY ADDRESS

C++ provides two methods for passing variables between functions.

Passing by value – you would use if the variable is NOT to be changed in the called function.
Pass by address – use this method if the variable WILL be changed in the called function.

9.1.1 PASS BY VALUE (= PASS BY COPY)

When passing by value "by copy", only a copy of the VALUE of the variable is passed, not the variable itself.

If the called function may change the value of the variable, but it is only changing ITS copy of the variable, and not the calling function's copy of it.

9.1.2 PASS BY ADDRESS (= PASS BY REFERENCE)

If you need to pass a variable but need the called function to be able to change that variable in the calling function too, you must pass by address.

Note – arrays are always "pass by address".

9.1.3 PASSING NON-ARRAYS BY ADDRESS

1. Precede the variable in the calling function with an ampersand (&) symbol.
2. Precede the variable in the receiving function with an asterisk (*) everywhere the variable appears.

&	"Address of" operator
*****	"dereferencing" operator ("value held in")

CHAP 10 - STOP HERE

CHAP 11 - C++ BY EG

The following comes from C++ By Example (1994) by Perry and Ross

11.1.1 INTRO COMPUTER PROGRAMMING
11.1.2 PROGRAM DESIGN
11.1.3 READABILITY AND COMMENTING
11.1.4 ARDUINO SOFTWARE
11.1.5 FIRST ARDUINO PROGRAM & SERIAL MONITOR
11.1.6 PREPROCESSOR DIRECTIVES
11.1.7 ARITHMETIC & MATH OPERATORS
11.1.8 RELATIONAL OPERATORS
11.1.9 IF, ELSE, ELSE IF
11.1.10 LOGICAL OPERATORS
11.1.11 ADDITIONAL C++ OPERATORS (later)

THE ? CONDITIONAL OPERATOR

```
// conditionalExpression ? expression1 : expression2;

(sales > 8000) ? bonus = 500 : bonus = 0;

//What it does
//If conditional is True, execute expression1; else do expression2
```

11.1.12 (++) INCREMENT AND DECREMENT (– –) OPERATORS
11.1.13 sizeof OPERATOR

```
This is compile-time operator (done at compile, not run time)

sizeof data                //but good idea to use ( ) all the time
sizeof (data type)

//Example
sizeof(float)              //result is 4
```

11.1.14 comma (,) OPERATORS
11.1.15 BITWISE OPERATORS (And, Or, Xor) (skip)
11.1.16 while LOOPS
11.1.17 do-while LOOPS
11.1.18 if vs. while
11.1.19 exit ( ) and break (skip)
11.1.20 counters and totals

You may need your code to count or produce totals. A good example would be if you are using an optical encoder with your Arduino, you may need to write code for counting the pulses coming from the encoder. The example below may be placed inside the setup ( ) function. Note the counter i starts at 0 and ends with 4 (that's 5 total). It is standard in C/C++ to start counting at 0.

```
//this code sends a string to the serial monitor 5 times
int i = 0;
do
{
Serial.println("Computers are fun")
i++;    // i = i+1
}
while (i < 5)
```

11.1.21 for LOOPS
11.1.22 break and continue (skip)
11.1.23 switch and goto (skip)
11.1.24 FUNCTIONS
11.1.25 CALLING AND RETURNING FUNCTIONS
11.1.26 VARIABLE SCOPE
11.1.27 PASSING VARIABLES
11.1.28 AUTOMATIC VS. STATIC VARIABLES
11.1.29 PASSING VALUES (CH 20)
11.1.30

mixing data types, type casting
(comparison) relational operators ==, <=
boolean &&, ||,
if, else
logical operators, &&, ||, true/false
other operators: ?, ++, --, comma
while loop (ch 14), do while, if while, exit( ) , break, totals
for loops (ch 15), nested
continue ? (ch 16),
switch, goto (ch 17)
functions (ch 18), break prog into fcns, calling and returning fcns,
variable scope (ch 19), global variables, passing variables, automatic vs. static var's,
passing values (ch 20), pass by value (copy), pass by address
Fcn return values, prototypes (ch 21),

device/charac I/O (ch 22), get, put
charac, string, numeric fcns (ch 23), string fcns,
numeric fcns – trig, logarithmic (exp, log), rand
arrays (ch 24), initialize,
ch 25 – array processing, searching arrays, sorting,
ch 26 – multi-dimensional arrays
ch 27 – pointers
ch 28 – pointers, arrays
data structures

TOPICS TO BE COVERED

We need to know a little C/C++ programming to program the Arduino.
Topics we will cover include:

include, #define (preprocessor directives)
Variables – declaring, assigning value to
Functions – syntax, return
Conditionals – if, else if; switch case
Loops – for, while; do while; break; continue
(goto)?

Syntax (form)
; (semicolon)
{ } (curly braces)
Arithmetic operators:
= (assignment operator)
+ (addition)
– (subtraction)
* (multiplication)
/ (division)
% (modulus or remainder)
Comparison (relational) operators
= = (equal to)
!= (not equal)
<
>
<=
>=
Boolean (logical) operators
&& (and)
|| (or)
! (not)
Data Types (int, float, boolean, byte, long, short, double)

include, #define (preprocessor directives)

```
#include <iostream.h>          //a preprocessor directive

main()                         // define the main () function
{
//main program goes here in this block
//
/*  example of
        multi-line commenting
```

```
*/
}
```

Consider the following program and the explanations afterwards

```
// Filename: C4first.cpp
// To demonstrate comments, variables, and declarations

// preprocessor directive
# include       <iostream.h>   //include a library to access its fcns

main()
{                       // curly brace starts the "block"
int i;          //Declare all variables
int j;          //This sets aside memory to store their values
char c;
float x;

i = 4;          // Assign value for i

j = 1 + 7;      // Eval 1+7, which is 8, then place that into variable j

c = 'A';                //enclose character constants in single quotes

x = 9.087;      //x requires a decimal value since declared as float

x = x * 4.5;    // change what is in x by this formula

//Send these values to the screen
cout << i << j << c << x;

return 0;       // always end programs with return
}
```

#include is a pre-processor directive.

The program essentially describes the DEFINITION for the main ( ) function.
The curly braces ({ }) enclose the DEFINITION of the main ( ) function. They are a list of statements that make up the definition for main ( ).
int i is an example of the declaring of a variable named "i". Declaring a variable sets aside memory to store the value of that variable.
A variable is like a container that holds a value. The value can be a number or a letter.
x = 1; is an example of assigning a value (in this case 1) to a variable (in this case x). The equal sign (=) is called an assignment operator.

Other math operators include addition (+), subtraction (□), multiplication (*), and division (/).

setup( ) and loop ( ) – all Arduino programs have them

if
if else
for
switch case
while
do while
break
continue
return
goto

II. Syntax

; (semicolon)
{ } (curly braces)
// (single line comment)
/* */ (multi-line comment)
#define
#include

III. Arithmetic Operators

= (assignment operator)
+ (addition)
– (subtraction)
* (multiplication)
/ (division)
% (modulus or remainder)

IV. Comparison (relational) operators

= = (equal to)
!= (not equal)
<
>
<=
>=

V. Boolean (logical) operators

&& (and)
|| (or)
! (not)

VI. Data Types

Type	Bits	Range
boolean	8	True or false
byte	8	0 to 255
int	16	-32,768 to 32,767
unsigned int	16	0 to 65,535
long	32	-2,147,483,648 to 2,147,483,647
unsigned long	32	0 to 4,294,967,295
short	16	-32,768 to 32,767
float	32	-3.4028235E+38 to 3.4028235E+38 6-7 decimal

CHAP 12 - C++ BY EXAMPLE *******

The notes below are based on Visual C++ 1.5 By Example, Que, 1994 (Greg Perry , Jim Ross), 840 pages, 658 to MFC.

12.1.1 Computer programming

Computer program – a set of instructions that causes a computer to manipulate data
Programming language – a means for a person to generate a computer program
Compilers – translate high-level languages to the set of instructions a computer understands.
High level languages
Examples – Fortran, COBOL, BASIC, etc.
Written more like how people talk and communicate.
Portable – written the same regardless of hardware.
Lower level languages
Assembly – machine-specific instructions (e.g., "MOV AL, #61h")
Machine code – binary-level code (0's & 1's) that works with a computer's native set of instructions
Compiler – translates high level languages to machine-specific code
Not portable – it is specific to computer hardware
C and C++
Is in between low and high-level language (so it is fast)
A block-structured, procedural language

C++ is a version that includes "object-oriented" programming (which uses so-called "classes")

Create "classes", a type of data object
Then create an instance of a class, which shares the same characteristics of the existing object.

The C++ Visual Environment (skip)
COMPUTER PROGRAMS (p. 29)
Computer programs
You write a detailed set of instructions for a computer (like a recipe)
Steps
Write the program (usually using an "editor")
Compile the program (often, an executable file is created, e.g., "Word.exe")
Execute (or run) the program
Interpreted languages (like BASIC) are not compiled. Instead an interpreter translates each program instruction and then executes it before translating the next one.

Design a program
Plan out your program before writing it
Steps
Define program
Design output
Break into logical steps

Compile program
Test
Debug (fix errors)
Test again
Debugging – you must be very accurate in your typing.

12.1.2FIRST C++ PROGRAM (p. 41)

A basic program

```
#include <iostream.h>     //a preprocessor directive

main()                    // all C programs have the "main()" function
{
  //main program goes here in this block
  //
  /*  example of
        multi-line commenting
  */
}
```

C is sequential
Pre-processor directives
The C program is normally routed thru a pre-processor before it is compiled.
Pre-processor directives control how the program is compiled.
C is "free-form"
Program statements can start in a column of any line.
You can insert blank lines
Make programs compact READABILITY is key (have some white space)
C is case sensitive
VariableXX is different from variableXx.
Braces and main( )
Parentheses ( ) and curly braces { } and brackets [ ] are different!
Always come in pairs (and these items are used for different purposes)

Executable C++ statements end with a semi-colon (;)
main ( ) is required and is always the first function executed.
Functions are small sections of code.
Comments

```
// Single line
/* multiple line */

// Filename: C4first.cpp
// To demonstrate comments, variables, and declarations

// preprocessor directive
# include     <iostream.h>      //include a library to access its fcns

main()
{             // curly brace starts the "block"
int i;        //Declare all variables
int j;        //This sets aside memory to store their values
char c;
float x;
```

```
    i = 4;          // Assign value for i

    j = 1 + 7;      // Eval 1+7, which is 8, then place that into variable j

    c = 'A';        //enclose character constants in single quotes

    x = 9.087;      //x requires a decimal value since declared as float

    x = x * 4.5;    // change what is in x by this formula

    //Send these values to the screen
    cout << i << j << c << x;

    return 0;       // always end programs with return
    }
```

Comments in C (p. 47)
// single line
/* Multiple line */
Cannot nest multiple line comment within another.
Comment as you go
Sample Program explained
Preprocessor directives
main ( ) – the { } enclose the body of the program
Declare variables – set aside memory to store their values
Programs have
Commands
Data
Variables – are a "container" (box) that holds a value. They can be numeric or character-based.
Constants
Assignment operator (=) – is how you put a value in the box.
cout = console output
Return statement – tells C++ that the function is finished and returns a value of 0 to main program
Summarized
Programs have: commands and data
Data are variables and constants.
Variables is like a box that holds something
'=' is an assignment operator (puts something in the variable box)
Math operators: + □ * /

12.1.3VARIABLE AND CONSTANTS (p. 55)

Variables – a place for storing a piece of data
Each variable
Has a name
Has a TYPE
Holds a value
Variable Name Rules
Limit 247 characters
Case sensitive (Case case cAse are all different)

Must start with a letter (ABC) or underscore "_" (do NOT start with a number)
Can't be same name as a C++ command or library function (e.g., "cout")
No spaces allowed (use an underscore "_" instead)
Make names meaningful ("total_payroll")
Variable Types
Variables hold different types of data

Type	Holds	Common ranges	Bits
char	character	-128 to 127	8
int	signed integer	-32768 to 32767	16
unsigned int	unsigned integer	0 to 65535	16
signed int	signed integer	032768 to 32767	16
long int (or long)	long integer	-2147483648 to 2147483647	32
float	floating-point	-3.4E+38 to 3.4E+38	
double	double precision floating pt.	-1.7E+308 to 1.7E+308	
long doublelong	double prec floating pt	-1.2E+4932 to 1.2E+4932	

Data types
Integers are whole numbers
Floating-point numbers have decimal points.
Unsigned numbers – is always (+)
char – hold characters (but are associated with #s per ASCII standards)
Rules
Can declare almost anywhere but must declare before referencing (using) them.
Notes
If declared before the function, then the variable is GLOBAL.
Examples

```
// declare on separate lines
int i;
int j;
char xx = 'a';

//declare on same line
float k, m;
```

Look at Data Types (p. 60)
Character – a SINGLE letter (not a string of letters)
But are associated with numbers per ASCII standards
C has no string data type (must do it another way)
Pick the proper data type (large enough to hold values that may be stored in it)
Otherwise you get OVERFLOWS
Suffixes
C++ interprets a typed number as the smallest type that can hold the number
E.g. – 63 is a signed int
But if you type 63L, you can override and make C consider it a long int.
6.82 is a double
Assigning Values
Syntax

```
variable = expression;
```

Expression can be number or equation

" = " is an "assignment operator". It places the value of the expression into the variable.
That is, it places the value on the RIGHT into the variable on the LEFT.

Rules
NO commas (e.g. x = 25,000)
In general, if assigning a value to a float, put the decimal (e.g. – 1000.0)
Don't mix types
Use single quotes to assign data to a char
Can declare and initialize all at once

```
int     age = 30;
float   salary = 250000.00;
```

Examples

```
//program to initialize and output variables

#include <iosteam.h>

main()
{
  int age = 30;                        // declare and assign value
  float salary = 25000.00;

  char first = 'G', middle = 'M', last = 'P';

  cout << age << ' ' << salary << ' ';
  cout << first << ' ' << middle << ' ' << last;
  return 0;
}

//Output is:
30 25000 G M P
```

Special Integer Constants (skip)
Precede with 0 ... octal (base 8) constant (e.g. 012... does not = 12)
Precede with 0x ... hexadecimal (base 16) (e.g., 0x2C4)

STRING CONSTANTS
Has no matching variable
Always enclose in double quotes
E.g. – "C++ Programming" or "1 2 3 4 5" (not a number to C)
Always have a zero at the tail called a "string delimiter".
It tells C where the string ends in memory
This 0 is not an ASCII zero (valued at 48),
Example of how "I am 30" is stored in memory

```
I       01001001
        00100000
a       01100001
```

```
m       01101101
        00100000
3       00110011
0       00110000 (character 0 = 48 in binary)
        00000000 (string terminating null zero)
```

Length of a string – number of characters, including space, but EXCLUDING the null zero
String vs. character constants
"char's" are enclosed in single quotes (e.g., 'w')
String constants are enclosed in double quotes ("w")
Character constants have NO NULL ZERO
Special characters
n-tilda is '\xA5' (since there's no key on the keyboard for it)
ESCAPE SEQUENCE characters

```
\a      alarm
\b      backspace
\n      new line (return)
\t      tab
\?      Question mark
```

12.1.4CHARACTER STRINGS AND CHARACTER ARRAYS (p. 79)

CHARACTER ARRAYS
C has no "type" for strings (sequences of characters)
Must use an array of characters (character array)
An array is a table or list of variables under a single name
The index of an array STARTS AT 0, not 1.

```
char name [ 7 ] = "Bob Le";
```

There is a NULL ZERO!

```
name [0] B
name [1] o
name [2] b
name [3]
name [4] L
name [5] e
name [6] \0
name [7]
```

Size
Make sure you size the array correctly.
Or let C do it

```
char horse [ ] = "Stallion";
```

Cannot assign value after declaration (must do it at same time?)
Can assign one element at a time
Including the null zero (var[i] = "\0")
Character Arrays vs. Strings (p. 82)
Strings must be stored in character arrays, but not all character arrays contain strings.
If a character array holds a string, then it has a null zero at the end!!

```
//character array holding individual characters, not a string & NO null zero
char cara1[ 10 ] = { 'a', 'b', 'c', 'd'};
//char array holding a string (need extra element to hold NULL ZERO
char cara2[ 5 ] = "open";
```

You cannot assign string values in character arrays using regular assignment statements, this can only be done at the time the character array is declared

```
char name [6] = "Bob Do";       // ok

//
char name2 [6];

name2 = "Bob Do";                    // NOT ok

strcpy (name2, "Bob Do")        // do it this way instead
```

12.1.5 PRE-PROCESSOR DIRECTIVES (p. 91)

Intro – recall that a compiler routes code thru a pre-processor before compiling.
Precede with # (indicates this is a pre-processor command, not a C command)

```
//Eg

#include <iostream.h>
#define AGE 28
#define MESSAGE "Hello, world"
```

#include
It MERGES files (temporarily)
Format

```
#include <filename> //look in default dir set up by your compiler
#include "filename" //look in the source code's dir, & then the include dir
```

Filename is a text file (just like your source code)

When to use "include"
A file that has code you use often (so now you don't have to repeat)
Eg – a few lines of code that prints your name on each of your programs. Save this into "MyName.cpp" and include it in your other code.

```
cout << "Bob Le \n";
cout << "wrote this code." >>
```

To include header files (special system files)
Inform C how to interpret many of the library functions
Eg - <iostream.h> holds info on "cout" or "cin"

#define (p. 97)

```
#define ARGUMENT1 argument2
```

Find-and-replace command
Can think of it as "defined constants"
Easier – can change the value in one place instead of many.
Format
NO EQUAL SIGN (eg - # define PI = 3.1415; is wrong)
NO SEMI-COLON at end
Convention – ARGUMENT1 is CAPITALIZED.
"argument2" can be an expression

Examples

```
#define PI          3.14159265351
#define AGELIMIT    21
#define       X1 (b + c)        //can do this, it'll substit expressions
```

Alternative – use CONSTANT variable

```
const int     i = 12;           //may not have the scope if defined in fcn
```

Can do this, but not recommended (can use to

```
#define X2          ((b + c) + (b + c))
#define X3          (X2 + c + (b + c) – d)
#define X4          (etc.)
```

12.1.6 SIMPLE INPUT AND OUTPUT (p. 107) (skip)

cout command
Sends data to standard output device (like the screen)
Format
Note [<< data] means there are optional additional items that may be added
"data" may be variables, constants, expressions, or combinations of all 3.

```
cout << data [ << data ];
```

Printing strings
X

```
cout << "The rain in Spain";
```

Notes
cout does NOT perform automatic hard returns

```
cout << "Line 1";
cout << "Line 2";
```

Produces

```
Line 1Line 2
```

If you want different lines, then do this

```
cout << "Line 1\n";              OR          cout << "Line 1";
cout << "Line 2\n";                          cout << "\nLine 2\n";
```

Produces

```
Line 1
Line 2
```

cin command
Fills variables with values
Format

```
cin [>> values];
```

Notes
Needs the iostream.h header.
PLACE AN EXAMPLE HERE

12.1.7MATH OPERATORS & PRECENDENCE (p. 131)

Primary Math Operators and sample results
Binary operator – the operator is between 2 numbers and acts on the 2 numbers

```
*       multiplication
/       division or integer division
%       modulus or remainder
+       addition
-       subtraction

a = 4*2;       // a = 8
b = 64/4;            // b = 16
c = 80 – 15          // c = 65
d = 12 + 9           // d = 21
```

Unary Operators – operate on a single value

```
//Unary operator example
a = -25;
```

Division and Modulus
Division (/)
If integers are on both sides, produces integer results (discarding remainder... it does NOT round up).
Modulus (%)
Produces the reminder of integer division (only works with TWO INTEGERS)

```
a = 9 / 4;           //output is a = 2, not 2.25
b = 10.0 / 4.0;      //output is b = 2.5
c = 10 % 3;          //output is c = 1
d = 300 % 100; //output is d = 0 (no remainder)
```

Order of Precedence

* / % are first, then + -(left to right)

Order	Operator
First	* / % (no hierarchy within this level)
Second	+ - (again, no hierarchy within this level)

Result = 2 + 3 * 2; // Result = 8, not 10

Using parentheses to alter order of operation, or to clarify order, even if not needed

Result = (2 + 3) * 2; // Result = 10
B = 5 * (5 + (6 – 2) + 1) // B = 50

The Assignment Statement
Multiple assignment (p. 142)
Note – assignment is from RIGHT TO LEFT so

```
a = b = c = d = e = 100;  //e = 100 is assigned first
```

Note – can assign within an expression

```
x = 4;
value = 5 + (r = 9 * x);  //this is valid:  r = 36, & then value = 41
```

Notes
C does NOT initialize variables to 0 automatically.

```
//good practice
int x, y;
x = y = 0;
```

Compound assignment
Take variable's current value, use it in an expression, & assign it back to the original variable.

```
salary = salary * 2;      //this is valid
```

"Compound operators"

```
                        // meaning
a += 500;               // a = a + 500;
b -= 50;        // b = b – 50;
c *= 1.2;               // c = c * 1.2;
d /= .50;               // d = d/.50;
e %= 7;                 // e = e % 7;
```

Mixing Data Types in Calculations
Notes – If mixing different data types (eg, int and float), C will generally convert the smaller type into the larger type. Example – add int and float, C will first convert the int to a float.

```
int     bonus = 50;
float   salary = 1400.50;
float   total;

total = salary + bonus;          //bonus is temporarily converted to a float
```

Type Casting
Usually C is successful at automatic conversion
But problems may occur when you mix unsigned variables with variables of other types.
Use type casting to solve this problem
Type casting: temporarily change a variable's type from its declared one to a new one

```
(data type) expression
```

Example

```
// int         age = 20;
// float factor = 0.10;
age_factor = (double) age * factor;
```

12.1.8RELATIONAL OPERATORS (p. 153)

Intro
Sometimes we don't execute every statement in a program.
Sometimes we want to execute some statements under SPECIFIC CONDITIONS.
Relational Operators
Compare data
Table

```
//Operator    Description

==            // Equal to
>             // Greater than, arrow points to the SMALLER value
<             // Less than
>=            // Greater than or equal to
<=            // Less than or equal to
!=            // Not equal to
```

Note
"=" is an assignment operator
"= =" tests equality
Do not interchange them
"=>" does not work, it must be ">=".
Relational logic produces a TRUE or FALSE result, where

True = 1 (or really non-zero)
False = 0

if Statement
You incorporate relational operators into "if" statements
Or "decision" statement because it tests the relationship, using relational operators, and makes a decision about which statement to execute next, based on the result of that decision. If the condition is TRUE, perform the block of statements in braces. Otherwise the condition is False, so do NOT execute the block of statements.

```
if (condition)
{
Statement 1;  //execute statements in curly braces if condition is true
Statement 2;  //indent for readability
}
//curly braces not needed if only 1 statement, but do it anyways
```

Notes
No semicolon in the "if" line.
Don't confuse = and = =.
Can put expressions in the "condition"
= = associates left to right

```
if (10 == 10 == 10)

//the condition is False!  The left "10 == 10" is true so = 1.
//then 1 == 10 is the last test, false!
```

Expressions as conditions
Example

```
int age = 21;

if (age = 85)
{
cout << "You have lived through a lot!";
}
// this program runs!  85 is assigned to age, and since age is non-zero the
// condition is TRUE, and the block is executed.
// Basically TRUE is NON-ZERO
```

else Statement
Else never appears without "if".
There is no (condition) after else's block executes when if's (condition) is false.

```
if (condition)
{
// Block of 1 or more statements to be executed
// if "condition" is True
}
else    // no condition here, as all remaining possibilities happen here
{
// Block of 1 or more statements to be executed
// if "condition" is False
}
```

Notes
Can compare characters (it compares the ASCII values)
Cannot compare character strings or arrays of character strings directly with relational operators
Can nest "if's".
else if – if you want to track more conditions. Else if has a (condition)!
Can have many "else if"s but the "else" must appear last if it is used.
Once an else if succeeds, none of the other else if's will be tested
QUESTION – SO ORDER OF ELSE IF's MATTERS?
DO THE ELSE IF'S NEED TO BE EXCLUSIVE? (IE – CAN 2 BE TRUE?)

```
if (condition)
{
//code if condition is TRUE
}
else if (condition2)
```

```
{
//code if condition2 is TRUE
//if there's overlap in condition2 & 3, then 1st else if runs...
}
else if (condition3)
{
//code if condition3 is TRUE
}
else // this appears last, has no condition, & runs if condition is FALSE
```

EXAMPLE

```
int main ()
{
int a = 100;

if( a == 10 )
{
  cout << "Value of a is 10" << endl;
}
else if( a == 20 )
{
  cout << "Value of a is 20" << endl;
}
else if( a == 30 )
{
  cout << "Value of a is 30" << endl;
}
else
{
  cout << "Value of a is not matching" << endl;
}

cout << "Exact value of a is : " << a << endl;

return 0;
}
```

12.1.9LOGICAL OPERATORS (p. 173)

Intro

Logical operators are AND, OR, and NOT.
Combine relational operators (<, >, = =) and logical operators to create powerful data-testing statements.

```
Operator      Meaning
&&            AND
| |                 OR
!             NOT
```

&& AND Truth Table (both sides of operator must be True)

```
False   &&    False       =     False
False   &&    True  =     False
True    &&    False       =     False
True    &&    True  =     True
```

| | OR Truth Table (either side of operator must be True)

```
False   | |   False       =     False
False   | |   True  =     True
True    | |   False       =     True
True    | |   True  =     True
```

The ! NOT Truth Table (either side of operator must be True)

NOT True = False
NOT False = True

Using Logical Operators
Sntax of compound relational operators

```
if ((a < b) && (c > d))
{
cout << "results are invalid.";
}
```

B. Example

```
if ((50 > 30) || (90 < 81))
{
//statement above True
}
```

Notes
Use ! sparingly (it tends to confuse)
Note: ! (var1 == var2) is same as (var1 != var2)
Can have > 2 relational tests but suggest limiting to 2 (& use nested "ifs")
C++'s Logical Efficiency
C attempts to be efficient & will not always interpret the full expression if it is not needed.
Example:

```
if ((7 < 3) && (sales < 15) && (15 != 15)) ...
```

C looks at the first condition (7 < 3), which is False and does not look further because False && (and) anything else remains False.

Example

```
//Filename: C11YEAR.CPP
//Determine if it is a Summer Olympics year, US census yr, or both

#include <iostream.h>

main ()
{
int year;

//Ask for a year
cout << "What is a year for the test?";
cin >> year;

//Test the year
```

```
if ((year % 4) == 0 && (year % 10) == 0)
{
  cout << "\nBoth Olympics and US Census!\n";
}
else
{
  if ((year % 4) == 0)
  {
        cout << "\nSummer Olympics only\n";
  }
  else
  {
        if ((year % 10) == 0)
        {
              cout << "\nUS Census only\n";
        }
        else
        {
              cout << "Neither\n"
        }
  }
}
```

Precedence of Logical Operators
Multiplications first
Relational operators (< >)
AND logical operator (&&)
OR logical operator (||)

12.1.10 ADDITIONAL C++ OPERATORS (p. 12)

The ?: Conditional Operator
Syntax and example

```
// conditionalExpression ? expression1 : expression2;

(sales > 8000) ? bonus = 500 : bonus = 0;

//What it does
//If conditional is True, execute expression1; else do expression2
```

Increment (++) and Decrement (□ □) Operators
Nte – efficient as they compile directly into their assembly language equivalent.
Cannot increment or decrement an expression

```
//Example                 Meaning          or
i++;                      i = i + 1;       i += 1;
i--;                      i = i - 1;       i -= 1;
```

sizeof
Compile-time operator (done at compile, not run time)

```
sizeof data               //but good idea to use ( ) all the time
sizeof (data type)
```

```
//Example
sizeof(float)             //result is 4
```

Comma Operator

12.1.11 BITWISE OPERATORS (p. 199)

And, Or, Xor

12.1.12 while LOOPS (p. 217)

while Statement

```
while (test expression)
{
//block code here
//this code executes over and over while the test expression is TRUE
}
```

if vs. while
while loops repeat a section of code many times
if statement – may or may not execute code, but if it does execute, it does it only once.

12.1.13 for LOOPS (p. 243)

To repeat sections of code a specific number of times

```
for (start expression; test expression; count expression)
{
//block code
}

Example
for (i = 1; i <= 4; i++)
{
cout << i << "\n";
}
return 0
//code outputs numbers 1 thru 4.
```

while vs. for
while loops continue until a certain condition is met.
for loops continue until a certain value is reached.

12.1.14 ADVANCED CONTROL OF FOR LOOPS (break and continue) (p. 265)

xx

SWITCH AND GOTO STATEMENTS (p. 277)

xx

WRITING C++ FUNCTIONS (p. 301)

Intro

Structured or modular programming – writing in modules

Good programs consist of many small functions.

Overview of Functions

Plan your program

Don't write 1 long program, but break it up into smaller sub-routines

main ( ) is the first function to execute

Breaking Program into Functions

Each function does one primary task.

```
main ()
{
getLetters();
alphabetize();
printLetters();
return 0;
}

getLetter()
{
//define getLetter() fcn here
return;                    //return to main()
}
alphabetize()
{
//define alphabetize() fcn here
return;                    //return to main()
}
printLetters()
{
//define printLetters() fcn here
return;                    //return to main()
}
```

Function mechanics

Each function has a name – max 247 characters, start w/ a letter, no spaces allowed

Follow the function name with ( ) (to distinguish it from variables.

The body of each function is enclosed by { }

Return command – returns control to calling function

Calling and Returning Functions (p. 306)

When defining a function NEVER follow with a semi-colon

Never define a function inside another function

```
void fcn1 (void)
{
//xxx
}
```

```
//************************************* (for readability)

void fcn2 (void)
{
/xxx
}

//************************************* for readability
```

12.1.15 VARIABLE SCOPE (p. 319)

Intro
Variable scope determines which functions recognize variables
Global vs. Local Variables
Global – are visible across many function and are dangerous (accidental changes to it)
Local – can only be seen & changed within the function in which the variable is defined
Defining Variable Scope
Local variable – if defined AFTER the opening brace of a block
Are destroyed when the function ends.
2 different functions can have local variables of the same name
Global variable – if defined OUTSIDE the function

12.1.16 PASSING VALUES (341)

ntro
2 ways to pass variables (depends on if you want to change the passed variables in called function)
Ways
Pass variables by value
Pass arrays by address
Pass non-arrays by address
Pass by Value (or by Copy)
When an argument (local variable) is passed by value, a COPY of the variable's value is assigned to the receiving function's parameter.

```
main ()
{
int i = 5;
FcnName (i);
return;
}

FcnName(int i)
{
cout << i;
return;
}
```

In the code above there are two i's, and they are not the same.
The receiving function CANNOT change the calling function's variable.

12.1.17 FUNCTION RETURN VALUES & PROTOTYPES (363)
12.1.18 DEVICES & CHARACTER I/O (381) (skip?)
12.1.19 CHARACTER, STRING, & NUMERIC FUNCTIONS (397)
12.1.20 ARRAYS (419)

Intro
Array is list of 2 or more variables with the same name
To initialize elements of a large array to ZERO at the same time, declare the entire array and initialize the first element to zero (C++ fills in the rest w/ zeroes)

```
//the first 3 elements are declared as shown, the rest will be zeroes
int values[100] = { 1, 2, 3 };
```

Do NOT treat arrays as strings (strings end in a null zero)

```
//here we declare & initialize the array
char initials[5] =
{'Q', 'K', 'P', 'G', V'}
```

12.1.21 ARRAY PROCESSING (437)
12.1.22 MUTI-DIMENSIONAL ARRAYS (457)
12.1.23 POINTERS (474)

Pointer variables (or just "pointers") are variables that contain the ADDRESSES of other variables.

POINTER VARIABLES

Pointers are variables. They follow the same naming rules as regular variables.
There is a pointer for every data type in C++ (e.g. – int pointers, float pointers, etc.)
There are global & local pointers.

About the only difference between pointer variables and regular variables is what they hold.
Pointers do not hold data, but ADDRESSES of data.

```
&       "Address of" operator
*       "dereferencing" operator ("value held in")
```

& and * are "overloaded" operators. They can perform more than one function depending on context.

DECLARING POINTERS

The following is declares an int variables (we've done this before). C++ reserves storage for that variable.

```
int age = 30;
```

Where did C++ store "age" in memory? The programmer probably doesn't care. You don't need to know the address since you'll just use the name "age".

Now let's declare a pointer variable

```
int* pAge;          //declare pointer.  The * tells C++ this is a pointer
```

As with any automatic variable, C++ does not initialize pointers. So now assign a value to the pointer

```
pAge = &age;
```

What value is in pAge? We don't know exactly, but we do know it is the ADDRESS of age, whatever it is.

12.1.24 POINTERS AND ARRAYS (487)
12.1.25 DATA STRUCTURES (509)
12.1.26 ARRAYS OF STRUCTURES (531)
12.1.27 SEQUENTIAL FILES (551)
12.1.28 RANDOM ACCESS FILES (569)

12.1.29 OBJECT-ORIENTED PROGRAMMING

12.1.30 CLASSES (585)

Defining a Class
A class is a user-defined DATA TYPE that have both member variables & member functions.
Member functions manipulate member variables, create & destroy class variables, ..
A class only defines a new data type, it does not declare variables of that data type.
Member functions are sometimes called "methods".
Class members have a new attribute: access specifiers or visibility
Member Variables
Can be of any data type
Pretty much just like a structure
Example – a circle might have member variables like radius and position (x,y).

```
// A circle class demo
# include <iostream.h>

//circle class
class Circle
{
public:             // this is the access specifier
float   rad;
float   x, y;
};

main()
{
Circle cir;
float   crad = 3.5;
cir.rad = crad;
cout << "The radius is ==>" << circ.rad;

return 0;
}
```

Member Functions
Member functions are functions defined within a class that act on the member variables in the class.
This is what distinguishes a class from a struct
(Note C++ allows you to define and use member function in a struct, but this is not generally done)

```
// A circle class demo
# define      PI 3.1415

class Circle
{
public:              // this is the access specifier
float   m_rad;            // the "m_" tells me this is a class member
float   m_x, m_y;

//constructor
Circle (float xcoord, float ycoord, float radius)
{
  m_x = xcoord; m_y = ycoord; m_rad = radius;
}
~ Circle () { }

// Member functions
float radius()
{
  return m_rad;
}

float circumference()
{
  return m_rad *2 * PI;
}
};  // end class
```

Constructors and Destructors
The act of declaring a new variable whose type is a class (or struct) is called "declaring an instance" of the class
Constructor functions are used to create a class variable and initialize it all at once.
x
x
x
x

12.1.31 INHERITANCE & POLYMORPHISM (607)
12.1.32 INTRODUCTION TO CLASS LIBRARIES (629)
12.1.33 INTRO TO OBJECT-ORIENTED PROGRAMMING (649)
12.1.34 MICROSOFT FOUNDATION CLASSES (p. 660)

When we want to go to the store, we bring our cart.

CHAP 13 - xx