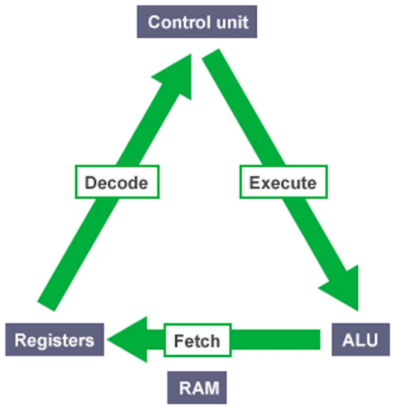
C programming:

The Approach/method that is used to solve a problem is known as algorithm.

Fetch / Execute Cycle



https://www.bbc.co.uk/education/guides/z2342hv/revision/5

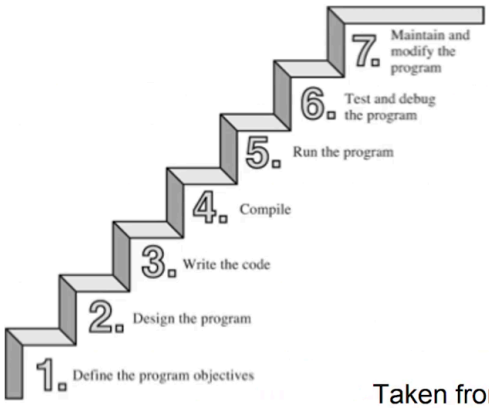
Higher Level Programming Languages

- Compilers
  - a program that translates the high-level language source code into the detailed set of machine language instructions the computer requires
  - the program does the high-level thinking and the compiler generates the tedious instructions to the cpu

- Compilers will also check that your program has valid syntax for the programming language that you are compiling
  - finds errors and it reports them to you and doesn't produce an executable until you fix them

Writing a program

- the act of writing a C program can be broken down into multiple steps



Taken from “C Primer Plus”, Prata

Overview

- C is a general-purpose, imperative computer programming language that supports structured programming
  - Uses statements that change a program’s state, focuses on how
- Currently, it is one of the most widely used programming languages of all time
- C is a modern language
  - has most basic control structures and features of modern languages
  - designed for top-down planning
  - organized around the use of functions (modular design) structured programming
  - a very reliable and readable program

Overview

- There are four fundamental tasks in the creation of any C program
  - Editing
  - Compiling
  - Linking
  - Executing

- These tasks will become second nature to you because you will be doing it so often

- The processes of editing, compiling, linking, and executing are essentially the same for developing programs in any environment and with any compiled language

- Editing is the process of creating and modifying your C source code
  - source code is inside a file and contains the program instructions you write
  - choose a wise name for your base file name (all source files end in the .c extension)
  - we will use an IDE (code blocks) for this class, but, you can use any editor (notepad, etc) to create your source code

Compiling

- A compiler converts your source code into machine language and detects and reports errors in your code
  - The input to the compiler is the file you produce during your editing (source file)

- Compilation is a two-stage process
  - the first stage is called the preprocessing phase, during which your code may be modified or added to
  - the second stage is the actual compilation that generates the object code

- the compiler examines each program statement and checks it to ensure that it conforms to the syntax and semantics of the language
  - can also recognize structural errors (dead code)
  - does not find logic errors
  - typical errors reported might be due to an expression that has unbalanced parentheses (syntactic error), or due to the use of a variable that is not “defined” (semantic error)

Compiling (cont'd)

- After all errors are fixed, the compiler will then take each statement of the program and translate it into assembly language

- the compiler will then translate the assembly language statements into actual machine instructions
  - the output from the compiler is known as object code and it is stored in files called object files (same name as source file with a .obj or .o extension)

- The standard command to compile your C programs will be cc (or the GNU compiler, which is .gcc)
  - cc -c myprog.c or gcc –c myprog.c
  - if you omit the -c flag, your program will automatically be linked as well

Linking

- After the program has been translated into object code, it is ready to be linked
  - The purpose of the linking phase is to get the program into a final form for execution on the computer
  - linking usually occurs automatically when compiling depending on what system you are on, but, can sometimes be a separate command

- The linker combines the object modules generated by the compiler with additional libraries needed by the program to create the whole executable
  - also detects and reports errors
    - if part of your program is missing or a nonexistent library component is referenced

- Program libraries support and extend the C language by providing routines to carry out operations that are not part of the language
  - input and output libraries, mathematical libraries, string manipulation libraries

Linking (cont'd)

- A failure during the linking phase means that once again you have to go back and edit your source code

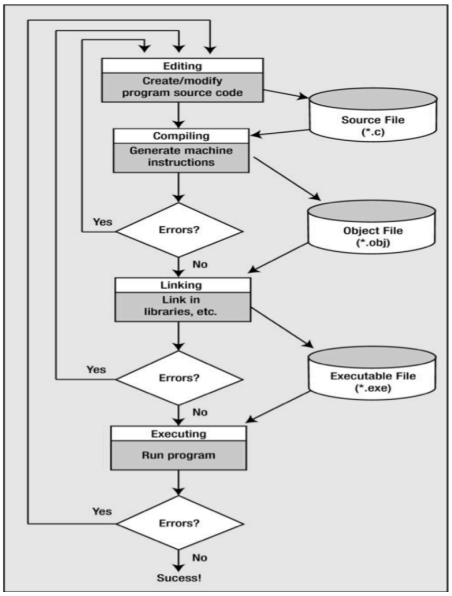
- Success will produce an executable file
  - In a Windows environment, the executable file will have an .exe extension
  - in UNIX / Linux, there will be no such extension (a.out by default)
  - Many IDEs have a build option, which will compile and link your program in a single operation to produce the executable

- A program of any significant size will consist of several source code files
  - each source code file needs the compiler to generate the object file that need to be linked

- the program is much easier to manage by breaking it up into a number of smaller source files
  - It is cohesive and makes the development and maintenance of the program a lot easier
  - the set of source files that make up the program will usually be integrated under a project name, which is used to refer to the whole program

C Stages

Taken from Beginning C, Horton



Overview

- the #include statement is a preprocessor directive

- #include <stdio.h>

- It is not strictly part of the executable program, however, the program won't work without it

- The symbol # indicates this is a preprocessing directive
  - an instruction to your compiler to do something before compiling the source code
  - many preprocessing directives
  - are usually some at the beginning of the program source file, but they can be anywhere
  - similar to the import statement in java

- In the above example, the compiler is instructed to "include" in your program the contents of the file with the name stdio.h
  - called a header file because it is usually included at the head of a program source file
  - .h extension

Header Files

- header files define information about some of the functions that are provided by that file
  - stdio.h is the standard C library header and provides functionality for displaying output, among many other things
  - we need to include this file in a program when using the printf() function from the standard library
  - stdio.h contains the information that the compiler needs to understand what printf() means, as well as other functions that deal with input and output
  - stdio, is short for standard input/output.

- header files specify information that the compiler uses to integrate any predefined functions within a program

Syntax

- header file names are case sensitive on some systems, so you should always write them in lowercase

- Two ways to #include header files in a program
  - Using angle brackets (#include <Jason.h>)
    - tells the preprocessor to look for the file in one or more standard system directories
  - Using double quotes (#include “Jason.h”)
    - tells the preprocessor to first look in the current directory

- Every C compiler that conforms to the C11 standard will have a set of standard header files supplied with it

- you should use #ifndef and #define to protect against multiple inclusions of a header file

Syntax

```
// some header

// typedefs
typedef struct names_st names;

// function prototypes
void get_names(names *);
void show_names(const names *);
char * s_gets(char * st, int n);
```

header files includes many different things
- #define directives
- structure declarations
- typedef statements
- function prototypes

executable code normally goes into a source code file, not a header file

Overview

- In our first challenge, you should have noticed that there was a line of code that included the word “printf”

- printf(“Hi, my name is Jason”);

- printf() is a standard library function
  - it outputs information to the command line (the standard output stream, which is the command line by default)
  - the information displayed is in based on what appears between the parentheses that immediately follow the function name (printf)
  - also notice that this line does end with a semicolon

scanf()

- like printf(), scanf() uses a control string followed by a list of arguments
  - control string indicates the destination data types for the input stream of characters
- the printf() function uses variable names, constants, and expressions as its argument list
- the scanf() function uses pointers to variables
  - you do not have to know anything about pointers to use the function
- Remember these 3 rules about scanf
  - returns the number of items that it successfully reads
  - If you use scanf() to read a value for one of the basic variable types we've discussed, precede the variable name with an &
  - If you use scanf() to read a string into a character array, don't use an &.
- The scanf() function uses whitespace (newlines, tabs, and spaces) to decide how to divide the input into separate fields
- scanf is the inverse of printf(), which converts integers, floating-point numbers, characters, and C strings to text that is to be displayed onscreen

Variable

Overview

- Remember that a program needs to store the instructions of its program and the data that it acts upon while your computer is executing that program
  - This information is stored in memory (RAM)
  - RAM's contents are lost when the computer is turned off
  - Hard drives store persistent data
- You can think of RAM as an ordered sequence of boxes
  - the box is full when it represents 1 or the box is empty when it represents 0
  - each box represents one binary digit, either 0 or 1 (*true* and *false*)
  - each box is called a *bit*
- bits in memory are grouped into sets of eight (byte)
  - each byte has been labeled with a number (*address*)
  - the address of a byte uniquely references that byte in your computer's memory.
- Again, memory consists of a large number of bits that are in groups of eight (called bytes) and each byte has a unique address

Variables

- The true power of programs you create is their manipulation of data
  - So, we need to understand the different data types you can use, as well as how to create and name variables
- Constants are types of data that do not change and retain their values throughout the life of the program
- Variables are types of data may change or be assigned values as the program runs
- Variables are the names you give to computer memory locations which are used to store values in a computer program
- For example, assume you want to store two values 10 and 20 in your program and at a later stage, you want to use these two values
  - Create variables with appropriate names
  - Store your values in those two variables.
  - Retrieve and use the stored values from the variables

Naming Variables

- The rules for naming variables in C is that all names must begin with a letter or underscore (_) and can be followed by any combination of letters (upper- or lowercase), underscores, or the digits 0–9

Jason
myFlag
i
J5x7
my_data
_anotherVariable

Naming Variables (cont'd)

- The below lists some examples of invalid variable names

temp$value - $ is not a valid character
my flag - embedded spaces are not permitted
3Jason – variable names cannot start with a number
int - int is a reserved word

- use meaningful names when selecting variable names
  - can dramatically increase the readability of a program and pay off in the debug and documentation phases

Declaring Variables

- declaring a variable is when you specify the type of the variable followed by the variable name
  - specifies to the compiler how a particular variable will be used by the program

- for example, the keyword int is used to declare the basic integer variable
  - first comes int, and then the chosen name of the variable, and then a semicolon
    - *type-specifier variable-name;*
  - to declare more than one variable, you can declare each variable separately, or you can follow the int with a list of names in which each name is separated from the next by a comma
  - C requires that all program variables be declared before they are used in a program

Initializing Variables

- To initialize a variable means to assign it a starting, or initial, value

- this can be done as part of the declaration
  - follow the variable name with the assignment operator (=) and the value you want the variable to have

```
int x = 21;
int y = 32, z = 14;
int x, z = 94;       /* valid, but poor, form */
```

- In the last line, only z is initialized
  - it is best to avoid putting initialized and noninitialized variables in the same declaration statement

Overview

- We understand that C supports many different types of variables and each type of variable is used for storing kind of data
  - types that store integers
  - types that store nonintegral numerical values
  - types that store characters

- Some examples of basic data types in C are:

```
int
float
double
char
_Bool
```

- the difference between the types is in the amount of memory they occupy and the range of values they can hold
  - the amount of storage that is allocated to store a particular type of data
  - depends on the computer you are running (machine-dependent)
  - an integer might take up 32 bits on your computer, or perhaps it might be stored in 64

int

- a variable of type int can be used to contain integral values only (values that do not contain decimal places)
- a minus sign preceding the data type and variable indicates that the value is negative
- the int type is a signed integer
  - it must be an integer and it can be positive, negative, or zero
- if an integer is preceded by a zero and the letter x (either lowercase or uppercase), the value is taken as being expressed in hexadecimal (base 16) notation
  - int rgbColor = 0xFFEF0D;
- the values 158, −10, and 0 are all valid examples of integer constants
  - no embedded spaces are permitted between the digits
  - values larger than 999 cannot be expressed using commas (12,000 must be written as 12000)

float

- A variable to be of type float can be used for storing floating-point numbers (values containing decimal places)
- the values 3., 125.8, and −.0001 are all valid examples of floating-point constants that can be assigned to a variable
- floating-point constants can also be expressed in scientific notation
  - 1.7e4 is a floating-point value expressed in this notation and represents the value 1.7 × 10 to the power of 4

double

- the double type is the same as type float, only with roughly twice the precision
  - used whenever the range provided by a float variable is not sufficient
  - can store twice as many significant digits
  - most computers represent double values using 64 bits
- all floating-point constants are taken as double values by the C compiler
- To explicitly express a float constant, append either an f or F to the end of the number
  - 12.5f

_Bool

- the _Bool data type can be used to store just the values 0 or 1
  - used for indicating an on/off, yes/no, or true/false situation (binary choices)
- _Bool variables are used in programs that need to indicate a Boolean condition
  - a variable of this type might be used to indicate whether all data has been read from a file
- 0 is used to indicate a false value
- 1 indicates a true value

Other Data Types

- the int type will probably meet most of your integer needs when beginning in C
- However, C offers many other integer types
  - gives the programmer the option of matching a type to a particular use case
  - integer types vary in the range of values offered and in whether negative numbers can be used
- C offers three adjective keywords to modify the basic integer type (can also be used by itself)
  - short, long, and unsigned
- The type short int, or short may use less storage than int, thus saving space when only small numbers are needed
  - can be used when the program needs a lot of memory and the amount of available memory is limited
- The type long int, or long, may use more storage than int, thus enabling you to express larger integer values
- The type long long int, or long long may use more storage than long
  - A constant value of type long int is formed by optionally appending the letter L (upper- or lowercase) onto the end of an integer constant
    - long int numberOfPoints = 131071100L;

Other Data Types (cont'd)

- Type specifiers can also be applied to doubles
  - long double US_deficit_2017;
- A long double constant is written as a floating constant with the letter l or L immediately following
  - 1.234e+7L
- The type unsigned int, or unsigned, is used for variables that have only nonnegative values (positive)
  - unsigned int counter;
  - the accuracy of the integer variable is extended
- The keyword signed can be used with any of the signed types to make your intent explicit
  - short, short int, signed short, and signed short int are all names for the same type

Enums

- a data type that allows a programmer to define a variable and specify the valid values that could be stored into that variable
  - can create a variable named "myColor" and it can only contain one of the primary colors, red, yellow, or blue, and no other values
- Your first have to define the enum type and give it a name
  - initiated by the keyword enum
  - then the name of the enumerated data type
  - then list of identifiers (enclosed in a set of curly braces) that define the permissible values that can be assigned to the type

```
enum primaryColor  { red, yellow, blue };
```

- variables declared to be of this data type can be assigned the values red, yellow, and blue inside the program, and no other values

Enums (cont'd)

- To declare a variable to be of type enum primaryColor
  - use the keyword enum
  - followed by the enumerated type name
  - followed by the variable list. So the statement

- enum primaryColor myColor, gregsColor;

- defines the two variables myColor and gregsColor to be of type primaryColor
  - the only permissible values that can be assigned to these variables are the names red, yellow, and blue
  - myColor = red;

- Another example

enum month { January, February, March, April, May, June, July, August, September, October, November, December };

Enums as ints

- the compiler actually treats enumeration identifiers as integer constants
  - first name in list is 0

```
enum month  thisMonth;
  ...
thisMonth = February;
```

- the value 1 is assigned to thisMonth (and not the name February) because it is the second identifier listed inside the enumeration list

- if you want to have a specific integer value associated with an enumeration identifier, the integer can be assigned to the identifier when the data type is defined

```
enum  direction  { up, down, left = 10, right };
```

- an enumerated data type direction is defined with the values up, down, left, and right
- up = 0 because it appears first in the list
- 1 to down because it appears next
- 10 to left because it is explicitly assigned this value
- 11 to right because it appears immediately after left in the list

Declaring a char

```
char broiled;       /* declare a char variable       */
broiled = 'T';      /* OK                            */
broiled = T;        /* NO! Thinks T is a variable    */
broiled = "T";      /* NO! Thinks "T" is a string    */
```

- If you omit the quotes, the compiler thinks that T is the name of a variable
- If you use double quotes, it thinks you are using a string
- you can also use the numerical code to assign values

Overview

- format specifiers are used when displaying variables as output
  - they specify the type of data of the variable to be displayed

```
int sum = 89;
printf ("The sum is %d\n", sum);
```

- The printf() function can display as output the values of variables
  - has two items or arguments enclosed within the parentheses
  - arguments are separated by a comma
  - first argument to the printf() routine is always the character string to be displayed
  - along with the display of the character string, you might also frequently want to have the value of certain program variables displayed

- The percent character inside the first argument is a special character recognized by the printf() function
  - the character that immediately follows the percent sign specifies what type of value is to be displayed

Code Walkthrough

- ```
  #include <stdio.h>

  int main (void)
  {
      int      integerVar = 100;
      float    floatingVar = 331.79;
      double   doubleVar = 8.44e+11;
      char     charVar = 'W';

      _Bool    boolVar = 0;

      printf ("integerVar = %i\n", integerVar);
      printf ("floatingVar = %f\n", floatingVar);
      printf ("doubleVar = %e\n", doubleVar);
      printf ("doubleVar = %g\n", doubleVar);
      printf ("charVar = %c\n", charVar);

      printf ("boolVar = %i\n", boolVar);

      return 0;
  }
  ```
  ```

Summary

Type	Constant Examples	printf chars
char	'a', '\n'	%c
_Bool	0, 1	%i, %u
short int	—	%hi, %hx, %ho
unsigned short int	—	%hu, %hx, %ho
int	12, -97, 0xFFE0, 0177	%i, %x, %o
unsigned int	12u, 100U, 0XFFu	%u, %x, %o
long int	12L, -2001, 0xffffL	%li, %lx, %lo
unsigned long int	12UL, 100ul, 0xffeeUL	%lu, %lx, %lo
long long int	0xe5e5e5e5LL, 50011	%lli, %llx, &llo
unsigned long long int	12ull, 0xffeeULL	%llu, %llx, %llo
float	12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1	%f, %e, %g, %a
double	12.34, 3.1e-5, 0x.1p3	%f, %e, %g, %a
long double	12.341, 3.1e-51	%Lf, $Le, %Lg

Taken from “Programming in C”, Kochan

Overview

- There are times when a program is developed that requires the user to enter a small amount of information at the terminal
- This information might consist of a number indicating the triangular number that you want to have calculated or a word that you want to have looked up in a dictionary
- Two ways of handling this
  - Requesting the data from the user
  - supply the information to the program at the time the program is executed (command-line arguments)
- We know that the main() function is a special function in C
  - Entry point of the program
- When main() is called by the runtime system, two arguments are actually passed to the function
  - the first argument (argc for argument count) is an integer value that specifies the number of arguments typed on the command line
  - the second argument (argv for argument vector) is an array of character pointers (strings)

Expressions and Statements (cont'd)

- Statements are the building blocks of a program (declaration)
  - A program is a series of statements with special syntax ending with a semicolon (simple statements)
  - a complete instruction to the computer

- Declaration statement: **int Jason;**
- Assignment Statement: **Jason = 5;**
- Function call statement: **printf("Jason");**
- Structure Statement: **while (Jason < 20) Jason = Jason + 1;**
- Return Statement: **return 0;**

- C considers any expression to be a statement if you append a semicolon (expression statements)
  - So, 8; and 3-4; are perfectly valid in C

Compound Statements

- two or more statements grouped together by enclosing them in braces (*block)*

```
int index = 0;
while (index < 10)
{
   printf("hello");
   index = index + 1;
}
```

Arithmetic Operators in C

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

****** All tables taken from tutorials points website *******

Logical Operators

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Assignment Operators

Operator	Description	Example
=	Simple assignment operator	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A

Relational Operators

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Bitwise Operators (tutorials point)

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -61, i.e,. 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Type Conversions

- conversion of data between different types can happen automatically (implicit conversion) by the language or explicit by the program
  - to effectively develop C programs, you must understand the rules used for the implicit conversion of floating-point and integer values in C

- Normally, you shouldn't mix types, but there are occasions when it is useful
  - remember, c is flexible, gives you the freedom, but, do not abuse it

- Whenever a floating-point value is assigned to an integer variable in C, the decimal portion of the number gets truncated

```
int x = 0;
float f = 12.125;
x = f;  // valu
```


ion is stored

Type Conversion (cont'd)

- assigning an integer variable to a floating variable does not cause any change in the value of the number
  - value is converted by the system and stored in the floating variable

- when performing integer arithmetic
  - if two operands in an expression are integers then any decimal portion resulting from a division operation is discarded, even if the result is assigned to a floating variable

  - If one operand is an int and the other is a float then the operation is performed as a floating point operation

The Cast Operator

- As mentioned, you should usually steer clear of automatic type conversions, especially of demotions
  - better to do an explicit conversion

- it is possible for you to demand the precise type conversion that you want
  - called a cast and consists of preceding the quantity with the name of the desired type in parentheses
  - parentheses and type name together constitute a cast operator, i.e. (type)
  - The actual type desired, such as long, is substituted for the word type

The Cast Operator

- The type cast operator has a higher precedence than all the arithmetic operators except the unary minus and unary plus

(int) 21.51 + (int) 26.99

- is evaluated in C as

21 + 26

sizeof operator

- You can find out how many bytes are occupied in memory by a given type by using the sizeof operator
  - sizeof is a special keyword in C
- sizeof is actually an operator, and not a function
  - evaluated at compile time and not at runtime, unless a variable-length array is used in its argument
- The argument to the sizeof operator can be a variable, an array name, the name of a basic data type, the name of a derived data type, or an expression

Other Operators

- the asterisk "*" is an operator that represents a pointer to a variable.

*a;

- ? : is an operator used for comparisons
  - If Condition is true ? then value X : otherwise value Y
  - name is the ternary operator

Overview

- the order of executing the various operations can make a difference, so C needs unambiguous rules for choosing what to do first
- In C, x is assigned 13, not 20 because operator * has a higher precedence than +
  - first gets multiplied with 3*2 and then adds into 7
- Each operator is assigned a *precedence* level
  - multiplication and division have a higher precedence than addition and subtraction, so they are performed first
- Whatever is enclosed in parentheses is executed first, should just always use () to group expressions

Associativity

- What if two operators have the same precedence?
  - Then associativity rules are applied

- If they share an operand, they are executed according to the order in which they occur in the statement
  - For most operators, the order is from left to right

1 == 2 != 3

Table (highest to lowest) (tutorials point)

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right

Table (highest to lowest)

Category	Operator	Associativity
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Overview

- The statements inside your source files are generally executed from top to bottom, in the order that they appear
- Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code
  - Decision-making statements (if-then, if-then-else, switch, goto)
  - Looping statements (for, while, do-while)
  - branching statements (break, continue, return)

If statements

Statement	Description
if statement	An if statement consists of a boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
nested if statements	You can use one if or else if statement inside another if or else if statement(s)

Repeating Code

- There may be a situation, when you need to execute a block of code several number of times
  - the statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on
- A loop statement allows us to execute a statement or a group of statements multiple times
- Loop control statements change execution from its normal sequence
  - When execution leaves a scope, all automatic objects that were created in that scope are destroyed (break and continue)
- A loop becomes infinite loop if a condition never becomes false
  - The **for** loop is traditionally used for this purpose

Loops

Loop Type	Description
while loop	It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	It is similar to a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

The conditional operator (ternary statement)

- the conditional operator is a unique operator
  - unlike all other operators in C
  - most operators are either unary or binary operators
  - is a ternary operator (takes three operands)

- the two symbols that are used to denote this operator are the question mark (?) and the colon (:)

- the first operand is placed before the ?, the second between the ? and the :, and the third after the :
  - condition ? expression1 : expression2

The conditional operator (ternary statement)

- The conditional operator evaluates to one of two expressions, depending on whether a logical expression evaluates true or false

- Notice how the operator is arranged in relation to the operands
  - the ? character follows the logical expression, condition
  - on the right of ? are two operands, expression1 and expression2, that represent choices.
  - the value that results from the operation will be the value of expression1 if condition evaluates to true, or the value of expression2 if condition evaluates to false

switch syntax

```
switch ( expression )
{
    case value1:
            program statement
          ...
            break;
    case valuen:
            program statement
            program statement
               ...
            break;
    default:
            program statement
             ...
            break;
}
```

switch statement details

- The expression enclosed within parentheses is successively compared against the values: value1, value2, ..., valuen
  - cases must be simple constants or constant expressions
- If a case is found whose value is equal to the value of expression then the statements that follow the case are executed
  - when more than one statement is included, they do not have to be enclosed within braces
- The break statement signals the end of a particular case and causes execution of the switch statement to be terminated
  - include the break statement at the end of every case
  - forgetting to do so for a particular case causes program execution to continue into the next case
- The special optional case called default is executed if the value of expression does not match any of the case values
  - same as a "fall through" else

Switch case example

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
enum Weekday today = Monday;

switch(today)
{
  case Sunday:
    printf("Today is Sunday.");
    break;
  case Monday:
    printf("Today is Monday.");
    break;
  case Tuesday:
    printf("Today is Tuesday.");
    break;
  default:
    printf("whatever");
    break;
}
```

goto statement

- The goto statement is available in C
  - has two parts—the goto and a label name
  - label is named following the same convention used in naming a variable

```
goto part2;
```

- For the above to there must be another statement bearing the part2 label

- you should never need to use the goto statement
  - if you have a background in older versions of FORTRAN or BASIC, you might have developed programming habits that depend on using goto

goto example

- Form:

```
goto label;
    .
    .
    .
label : statement
```

- Example:

```
top : ch = getchar();
      .
      .
      .
if (ch != 'y')
   goto top;
```

for loop

- You typically use the for loop to execute a block of statements a given number of times

- If you want to display the numbers from 1 to 10
  - Instead of writing ten statements that call printf(), you would use a for loop

```
for(int count = 1 ; count <= 10 ; ++count)
{
  printf("  %d", count);
}
```

- The for loop operation is controlled by what appears between the parentheses that follow the keyword for
  - the three control expressions that are separated by semicolons control the operation of the loop

- The action that you want to repeat each time the loop repeats is the block containing the statement that calls printf() (body of the loop)
  - for single statements, you can omit the braces

For syntax (cont'd)

- The general pattern for the for loop is:

```
for(starting_condition; continuation_condition ; action_per_iteration)
  loop_statement;
```

- The statement to be repeated is represented by loop_statement
  - could equally well be a block of several statements enclosed between braces

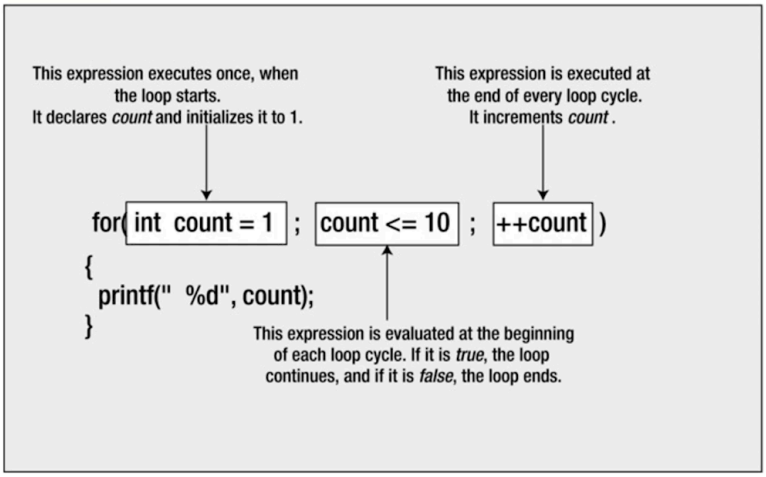
- The starting_condition usually (but not always) sets an initial value to a loop control variable
  - the loop control variable is typically a counter of some kind that tracks how often the loop has been repeated
  - can also declare and initialize several variables of the same type here with the declarations separated by commas
  - variables will be local to the loop and will not exist once the loop ends

Another Example

```
for(int i = 1, j = 2 ; i <= 5 ; ++i, j = j + 2)
  printf("  %5d", i*j);
```

- The output produced by this fragment will be the values 2, 8, 18, 32, and 50 on a single line

For syntax



for example (flexibility)

```
unsigned  long long sum = 0LL;          // Stores the sum of the integers
unsigned int count = 0;                 // The number of integers to be summed

  // Read the number of integers to be summed
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %u", &count);

  // Sum integers from 1 to count
  for(unsigned int i = 1 ; i <= count ; ++i)
    sum += i;

 // OR
for(unsigned int i = 1 ; i <= count ; sum += i++);

printf("\nTotal of the first %u numbers is %llu\n", count, sum);
```

Infinite loop

- you have no obligation to put any parameters in the for loop statement

```
for( ;; )
{
  /* statements */
}
```

- the condition for continuing the loop is absent, the loop will continue indefinitely
  - sometimes useful for monitoring data or listening for connections

do-while loop

- In the while loop, the body is executed while the condition is true

- the do-while loop is a loop where the body is executed for the first time unconditionally
  - always guaranteed to execute at least once
  - condition is at the bottom (post-test loop)

- After initial execution, the body is only executed while the condition is true

```
do
   statement
while ( expression );
```

```
  do
{
      prompt for password
         read user input
  } while (input not equal to password);
```

do-while loop example

- To make a while like a for, preface it with an initialization and include update statements

```
initialize;
while (test)
{
  body;
  update;
}
```

is the same as

```
for (initialize; test; update)
   body;
```

do-while loop example

- a for loop is appropriate when the loop involves initializing and updating a variable
- a while loop is better when the conditions are otherwise
- I usually use the while loop for logic controlled loops and the for loop for counter controlled loops

```
while (scanf("%l", &num) == 1)

for (count = 1; count <= 100; count++)
```

Continue statements

- Sometimes a situation arises where you do not want to end a loop, but you want to skip the current iteration
- The continue statement in the body of a loop does this
  - All you need to do is use the keyword “continue;” in the body of the loop
- An advantage of using continue is that it can sometimes eliminate nesting or additional blocks of code
  - can enhance readability when the statements are long or are deeply nested already

Continue Example

```
enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

for(enum Day day = Monday; day <= Sunday ; ++day)
{
  if(day == Wednesday)
    continue;

  printf("It's not Wednesday!\n");
  /* Do something useful with day */
}
```

Break statement

- normally, after the body of a loop has been entered, a program executes all the statements in the body before doing the next loop test
  - we learned how continue works
  - another statement named break alters this behavior

- the break statement causes the program to immediately exit from the loop it is executing
  - statements in the loop are skipped, and execution of the loop is terminated
  - if the break statement is inside nested loops, it affects only the innermost loop containing it
  - use the keyword “break;”

- break is often used to leave a loop when there are two separate reasons to leave

Break example

```
while ( p > 0)
{
   printf("%d\n", p);
   scanf("%d", &q);
   while( q > 0)
   {
      printf("%d\n",p*q);
      if (q > 100)
         break;          // break from inner loop
      scanf("%d", &q);
   }
   if (q > 100)
      break;             // break from outer loop
   scanf("%d", &p);
}
```

Generating a random number

- To generate a random number from 0-20
  - include the correct system libraries
    - #include <stdlib.h>
    - #include <time.h>
  - Create a time variable
    - time_t t;
  - Initialize the random number generator
    - srand((unsigned) time(&t));
  - Get the random number (0-20) and store in an int variable
    - int randomNumber = rand() % 21;

Sample Output

```
This is a guessing game.
I have chosen a number between 0 and 20 which you must guess.

You have 5 tries left.
Enter a guess: 12
Sorry, 12 is wrong. My number is less than that.

You have 4 tries left.
Enter a guess: 8
Sorry, 8 is wrong. My number is less than that.

You have 3 tries left.
Enter a guess: 4
Sorry, 4 is wrong. My number is less than that.

You have 2 tries left.
Enter a guess: 2

Congratulations. You guessed it!
```

Arrays

- it is very common to in a program to store many data values of a specified type
  - In a sports program, you might want to store the scores for all games or the scores for each player
  - you could write a program that does this using a different variable for each score
  - If there are a lot of games to store then this is very tedious
  - using an array will solve this problem
- arrays allow you to group values together under a single name
  - you do not need separate variables for each item of data
- an array is a fixed number of data items that are all of the same type

Accessing an array's elements

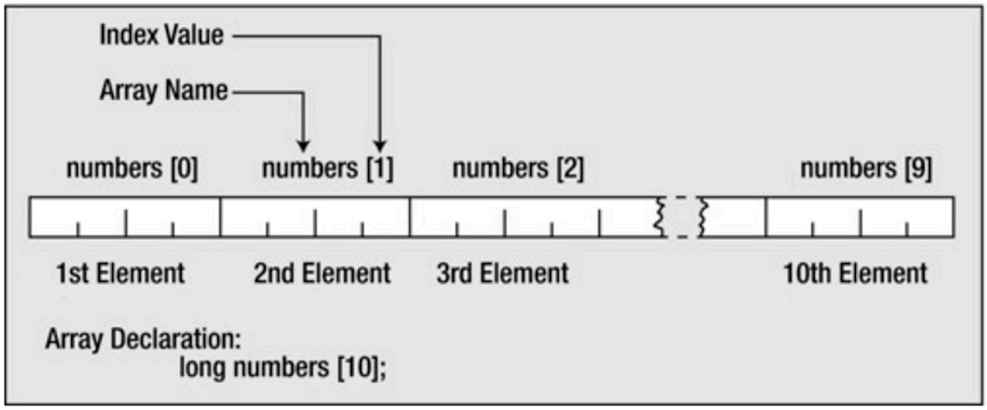
- each of the data items stored in an array is accessed by the same name
- you select a particular element by using an index (subscript) value between square brackets following the array name
- index values are sequential integers that start from zero
  - index values for elements in an array of size 10 would be from 0-9
  - Arrays are zero based
    - 0 is the index value for the first array element
    - for an array of 10 elements, index value 9 refers to the last element

Accessing an array's elements

- It is a very common mistake to assume that arrays start from one
  - referred to as the off-by-one error
  - you can use a simple integer to explicitly reference the element that you want to access
  - to access the fourth value in an array, you use the expression arrayName[3]
- You can also specify an index for an array element by an expression in the square brackets following the array name
  - the expression must result in an integer value that corresponds to one of the possible index values
- it is very common to use a loop to access each element in an array

```
for ( i = 0;  i < 10;  ++i )
    printf("Number is %d", numbers[i]);
```

Accessing an array's elements



Initializing an array

- you will want to assign initial values for the elements of your array most of the time
  - defining initial values for array elements makes it easier to detect when things go wrong
- just as you can assign initial values to variables when they are declared, you can also assign initial values to an array's elements
- to initialize an array's values, simply provide the values in a list
  - values in the list are separated by commas and the entire list is enclosed in a pair of braces

```
int  counters[5] = { 0, 0, 0, 0, 0 };
```

- declares an array called counters to contain five integer values and initializes each of these elements to zero

```
int  integers[5] = { 0, 1, 2, 3, 4 };
```

- declares an array named integers and sets the value of integers[0] to 0, integers[1] to 1, integers[2] to 2, and so on

Initializing an array (cont'd)

- It is not necessary to completely initialize an entire array
- If fewer initial values are specified, only an equal number of elements are initialized
  - remaining values in the array are set to zero

```
float  sample_data[500] = { 100.0, 300.0, 500.5 };
```

- initializes the first three values of sample_data to 100.0, 300.0, and 500.5, and sets the remaining 497 elements to zero

Designated Initializers

- C99 added a feature called designated initializers
  - allows you to pick and choose which elements are initialized
- by enclosing an element number in a pair of brackets, specific array elements can be initialized in any order

```
float sample_data[500] = { [2] = 500.5, [1] = 300.0, [0] = 100.0 };
```

- initializes the sample_data array to 100.0, 300.0, and 500.5 for the first three values

```
int arr[6] = {[5] = 212}; // initialize arr[5] to 212
```

Example of traditional initialization

```
#define MONTHS 12

int main(void)
{
   int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
   int index;

   for (index = 0; index < MONTHS; index++)
      printf("Month %d has %2d days.\n", index +1, days[index]);

   return 0;
}
```

Overview

- the types of arrays that you have been exposed to so far are all linear arrays
  - a single dimension
- the C language allows arrays of any dimension to be defined
  - two dimensional arrays are the most common
- you can visualize a two-dimensional array as a rectangular arrangement like rows and columns in a spreadsheet
- one of the most natural applications for a two-dimensional array arises in the case of a matrix
- two-dimensional arrays are declared the same way that one-dimensional arrays are

```
int  matrix[4][5];
```

- declares the array matrix to be a two-dimensional array consisting of 4 rows and 5 columns, for a total of 20 elements
  - note how each dimension is between its own pair of square brackets

Initializing a two dimensional array

- two-dimensional arrays can be initialized in the same manner of a one-dimensional array

- When listing elements for initialization, the values are listed by row
  - the difference is that you put the initial values for each row between braces, {}, and then enclose all the rows between braces

```
int numbers[3][4] = {
                      { 10, 20, 30, 40 },      // Values for first row
                      { 15, 25, 35, 45 },      // Values for second row
                      { 47, 48, 49, 50 }       // Values for third row
                    };
```

- commas are required after each brace that closes off a row, except in the case of the final row

Initializing a 2D array

- as with one-dimensional arrays, it is not required that the entire array be initialized

```
int  matrix[4][5] = {
                      { 10,  5, -3 },
                      {  9,  0,  0 },
                      { 32, 20,  1 },
                      {  0,  0,  8 }
                    };
```

- only initializes the first three elements of each row of the matrix to the indicated values
  - remaining values are set to 0.
  - in this case, the inner pairs of braces are required to force the correct initialization

Designated initializers

- subscripts can also be used in the initialization list, in a like manner to single-dimensional arrays

```
int matrix[4][3] = {  [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

- initializes the three indicated elements of matrix to the specified values
  - unspecified elements are set to zero by default
  - each set of values that initializes the elements in a row is between braces
  - the entire initialization goes between another pair of braces
  - the values for a row are separated by commas
  - each set of row values is separated from the next set by a comma

Other dimensions

- everything mentioned so far about two-dimensional arrays can be generalized to three-dimensional arrays and further
- you can declare a three-dimensional array this way:

```
int box[10][20][30];
```

- you can visualize a one-dimensional array as a row of data
- you can visualize a two-dimensional array as a table of data, matrix, or a spreadsheet
- you can visualize a three-dimensional array as a stack of data tables

Initializing an array of more than 2 dimensions

- for arrays of three or more dimensions, the process of initialization is extended
- a three-dimensional array will have three levels of nested braces, with the inner level containing sets of initializing values for a row

```
int numbers[2][3][4] = {
                        {                   // First block of 3 rows
                          { 10, 20, 30, 40 },
                          { 15, 25, 35, 45 },
                          { 47, 48, 49, 50 }
                        },
                        {                   // Second block of 3 rows
                          { 10, 20, 30, 40 },
                          { 15, 25, 35, 45 },
                          { 47, 48, 49, 50 }
                        }
                      };
```

Processing elements in a n dimensional array

- you need a nested loop to process all the elements in a multidimensional array
  - the level of nesting will be the number of array dimensions
  - each loop iterates over one array dimension

```
int sum = 0;
for(int i = 0 ; i < 2 ; ++i)
{
  for(int j = 0 ; j < 3 ; ++j)
  {
    for(int k = 0 ; k < 4 ; ++k)
    {
      sum += numbers[i][j][k];
    }
```

Variable length arrays

- so far, all the sizes of an array have been specified using a number
- the term variable in variable-length array does not mean that you can modify the length of the array after you create it
  - a VLA keeps the same size after creation
- variable length arrays allow you to specify the size of an array with a variable when creating an array
- C99 introduced variable-length arrays primarily to allow C to become a better language for numerical computing
  - VLAs make it easier to convert existing libraries of FORTRAN numerical calculation routines to C

Valid and invalid declarations of an array

```
int n = 5;
int m = 8;
float a1[5];              // yes
float a2[5*2 + 1];        // yes
float a3[sizeof(int) + 1];  // yes
float a4[-4];             // no, size must be > 0
float a5[0];              // no, size must be > 0
float a6[2.5];            // no, size must be an integer
float a7[(int)2.5];       // yes, typecast float to int constant
float a8[n];              // not allowed before C99, VLA
float a9[m];              // not allowed before C99, VLA
```

Hints

- The criteria that can be used to identify a prime number is that a number is considered prime if it is not evenly divisible by any other previous prime numbers
- Can use the following as an exit condition in the innermost loop
  - p / primes[i] >= primes[i]
  - a test to ensure that the value of p does not exceed the square root of primes[i]
- Your program can be more efficient by skipping any checks for even numbers (as they cannot be prime)

Advantages

- allow for the divide and conquer strategy
  - it is very difficult to write an entire program as a single large main function
    - difficult to test, debug and maintain

- with divide and conquer, tasks can be divided into several independent subtasks
  - reduces the overall complexity
  - separate functions are written for each subtask
  - we can further divide each subtask into smaller subtasks, further reducing the complexity

- reduce duplication of code
  - saves you time when writing, testing, and debugging code
  - reduces the size of the source code

- If you have to do a certain task several times in a program, you only need to write an appropriate function once
  - program can then use that function wherever needed
  - you can also use the same function in different programs (printf)

Advantages (cont'd)

- helps with readability
  - program is better organized
  - easier to read and easier to change or fix

- the divide and conquer approach also allows the parts of a program to be developed, tested and debugged independently
  - reduces the overall development time

- the functions developed for one program can be used in another program
  - software reuse

- many programmers like to think of a function as a "black box"
  - information that goes in (its input)
  - the value or action it produces (its output)

- using this "black box" thinking helps you concentrate on the program's overall design rather than the details
  - what the function should do and how it relates to the program as a whole before worrying about writing the code