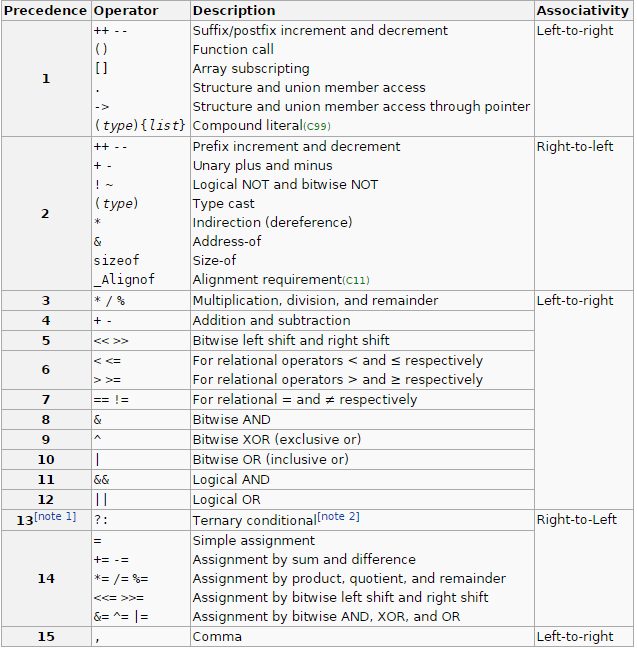
**Part 1**

* **C Operators**



**-Operator types**

-Arithmetic Operators; +, -, \*, /, %

-Increment and Decrement Operators; ++, --

-Relational Operators; ==, !=, >, <, <=, >=

-Logical Operators; &&, ||, !

-Cast Operators; (float)sum, (int)fred

-Bitwise Operators; ~, &, |, ^, <<, >>

-Assignment Operators; =, +=, -=, \*=, /=, %=, >>=, <<=, &=, ^=, |=

**-Order of evaluation**

C, like most languages, does not specify the order in which the operands of an operator are evaluated. (The exceptions are &&, ||, ?: and ‘,’.) For example, in a statement like

x = f() + g();

f may be evaluated before g or vice versa; thus if either f or g alters a variable on which the other depends, x can depend on the order of evaluation. Intermediate results can be stored in temporary variables to ensure a particular sequence.

Similarly, the order in which function arguments are evaluated is not specified, so the statement

printf(“%d %d\n”, ++n, power(2, n));

can produce different results with different compilers, depending on whether n is incremented before power is called. The solution of course, is to write

++n;

printf(“%d %d\n”, n, power(2, n));

The moral is that writing code that depends on order of evaluation is a bad programming practice in any language.

**-Shorthand operators:** A shorthand operator is a shorter way to express a statement that is already available in a language. Examples are compound operators and increment operators.

-**Compound/Composite assignment operators:** x -= 5 is equal to x = x - 5

**-Increment and decrement operators:** Difference between x = ++y operator and x = y++ is, ++y increases the number first then the number is used (for example x = ++y), whereas if you use y++ first the number is used then the increment is done. They dont have a difference if you use them in for loop header increment/decrement section. (tag: postincrement, preincerement, postdecrement, predecrement)

while ( x-- > 0 ) // is the same as

while( (x--) > 0 )

### -Short Circuit Evaluation

### If we use Logical AND ( && ) operator in a condition, if the left side of the condition is false, right side is not evaluated. Because it doesnt matter whether right side is true or false anymore; result will be false either way.

A similar logic is also valid for Logical OR ( || ) operator. If left side is right, there is no need to evaliate the right side. Because for OR operator as long as one of the operands is true, tjhe other operand is not important and returns true.

How could this be used? if( i == 1 & j++ ) , value of j is increased by one even if we dont enter the if block.

A disadvantage of this is, in the code (a <b) && (++c < 10), c is not incremented. But as long as you know about this feature when you are writing your programs, you will be fine. (tag: side effect)

**-Bitwise Operators:** ~, &, |, ^, <<, >>

x << 2 shifts the value of x left by two positions, filling vacated bits with zero.

Right shifting an unsigned quantity always fills vacated bits with zero. Right shifting a signed quantity will fill with sign bits (arithmetic shift) on some machines and with 0-bits (logical shift) on others.

The unary operator ~ yields one’s complement of an integer.

* **C Data types**

**-**In C and C++ **sizes of data types** change from platform to platform.

**-Basic Types:** They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.

**-Enumerated types(enum):**They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.

**-The type void:** The type specifier void indicates that no value is available.

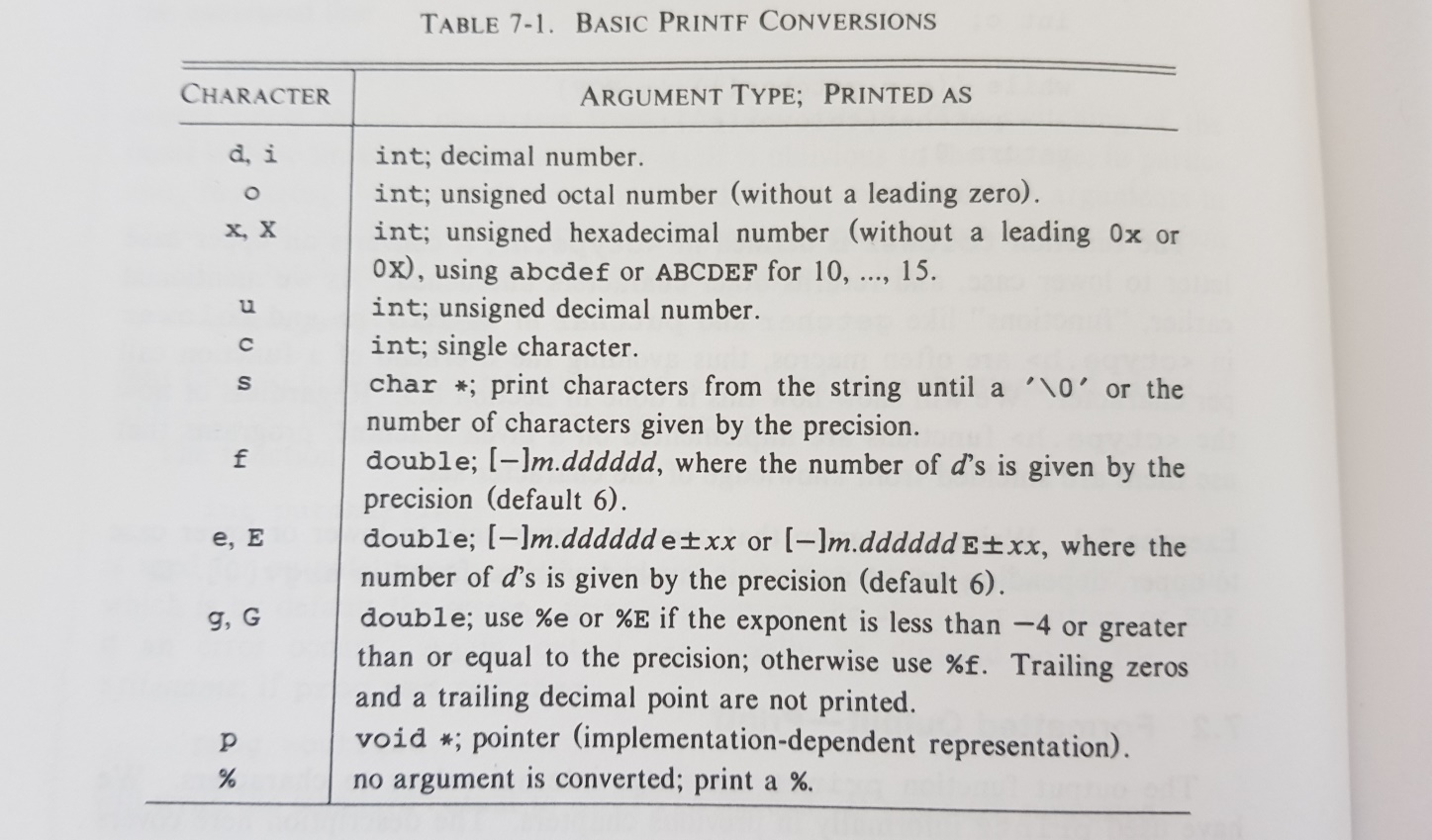
**-Derived types:** They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

|  |  |  |
| --- | --- | --- |
| **Type** | **Storage size** | **Value range** |
| char - c | 8 bit - 1 byte | -128 to 127 or 0 to 255 |
| signed char – c or hhi | 8 bit - 1 byte | -128 to 127 |
| unsigned char – c or hhu | 8 bit - 1 byte | 0 to 255 |
| short – hi | 16 bit - 2 byte | -32,768 to 32,767 |
| unsigned short - hu | 16 bit - 2 byte | 0 to 65,535 |
| int – d or i | 16 or 32 bit - 4 or 8 byte | -32,768 to 32,767 or  -2,147,483,648 to 2,147,483,647 |
| unsigned int – u | 16 or 32 bit – 4 or 8 byte | 0 to 65,535 or 0 to 4,294,967,295 |
| long – ld | 32 bit - 4 byte | -2,147,483,648 to 2,147,483,647 |
| unsigned long – lu | 32 bit - 4 byte | 0 to 4,294,967,295 |

|  |  |  |
| --- | --- | --- |
| Long long – llu | 64 bit - 8 byte | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| Unsigned long long – lli | 64 bit - 8 byte | 0 to 18,446,744,073,709,551,615 |

|  |  |  |  |
| --- | --- | --- | --- |
| float – f, F | 32 bit - 4 byte | 1.2E-38 to 3.4E+38 | 6 decimal places |
| double – f, F, lf, lF(scanf) | 64 bit - 8 byte | 2.3E-308 to 1.7E+308 | 15 decimal places |
| long double - LF | 80 bit - 10 byte | 3.4E-4932 to 1.1E+4932 | 19 decimal places |

|  |  |  |
| --- | --- | --- |
| pointer - p | 64 bit - 8 byte | 0 to FFFFFFFFFFFFFFFF which is equal to (18,446,744,073,709,551,615)10 |

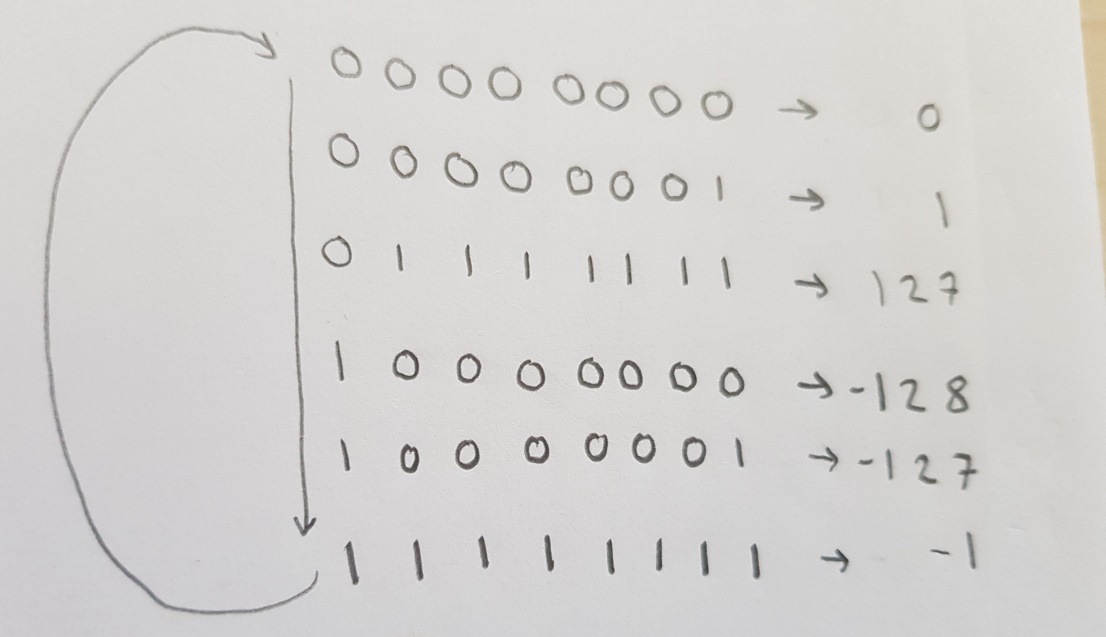
  
-Like other data types, size of pointers change depending on bit of OS, compiler, etc.

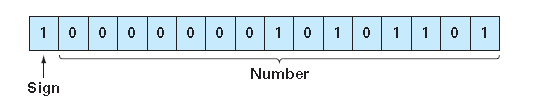
-The definition of C guarantees that any character in the machine’s standart printing character set will never be negative. But arbitrary bit patterns stored in character variables may appear to be negative on some machines, yet positive on others. For portability, specify signed or unsigned if non-character data is to be stored in char variables.

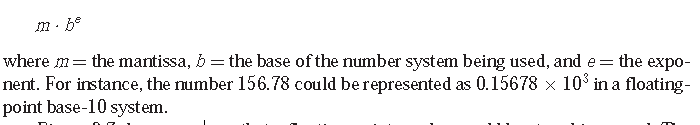
- char c = ‘a’ or char c = 97 does the same job.  
 - printf(“%c”, c-10);  
 - ++ndigit[c – ‘10’]; Chars are just small integers. So c - ‘0’ is an integer expression.  
 **printf”%c”, x);** prints integer value of x as a character.  
  
 **printf”%d”, x);** prints integer value of x as an integer.  
  
 **printf”%c”, ‘x’);** prints the character x.  
  
 **printf”%c”, “x”);** doesnt print anythingbecause a single character is not a string.  
  
 **printf”%d”, ‘x’);** printsascii value of the character x.  
  
 When we say x++, integer value of x is increased and we get the next ascii when we print again.  
  
 **printf”%d”, “x”);** undefined behaviour  
  
**-Undefined behaviour:** Undefined operations in C.

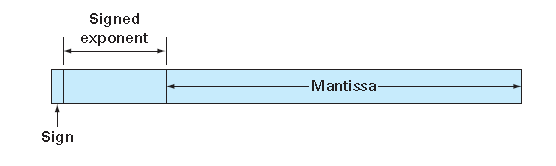
**-Binary representation of integers and floating point values** Signed variables in C use one bit as sign flag. Char is 8 bit, 2 byte. But it is 7 bit long. -128 to 127. Left most digit is sign flag. Short is 16 bit, 4 byte. But it is 15 bit long. -32,768 to -32,767. Int is 32bit, 8byte. But its 31 bit long. -2,147,483,648 to 2,147,483,647. If you define an unsigned variable, it will be twice as long. For example an unsigned short will have a limit of 64k not 32k. But it cant take negative values. If we have a 64 bit signed variable(16 byte), it can be 63 bit long. 9.22337204E+18.

-For signed char, the negative is -128 and the positive is 127 because negative can be 1000 0000. But positive can only be 0111 1111.

-If you have a binary number such as 7 (0000 0111) and you want to find binary representation of -7, all you need to do is find two’s complement of it. (1111 1001) You find positive representation of negative values the same way.  
  


**

****

****

**-Data Type Conversions  
  
 -Explicit type change / Cast Operator**

When you divide two int values, result will be int too. Even if you assign the int result to a float variable or print the int result as float, you will assign/print the truncated value. So we need to use the cast operator.

variable1 = (type) variable2;  
  
 int dividend = 12, divisor = 8;  
 float quotient;  
 quotient = (float) dividend / divisor;  
 printf("Result: %f\n",quotient);

**- Implicit type change**

When we have an operation that consists of two different variable or literal, the smaller data type is converted to bigger data type.

SMALL- char <-> short <-> int <-> long <–> long long <-> float <-> double -BIG

**-Default data types, Literals, Base**

An integer literal is an int. If you use the terminal l or L, it will be a long literal. An integer too big to fit into an int will also be taken as a long. Unsigned int literals are written with terminals u or U and unsigned long literal are written with terminals ul or UL. (tag: int literal)

Floating point literals (123.4, 1e-2) are type double. The suffixes f or F indicate a float literal. l or L indicate a long double.

A leading 0 will on an integer literal means octal. A leading 0x or 0X means hexadecimal. Octal and hexadecimal literals may also be followed by L to make them long and/or U to make them unsigned. A leading 0b or 0B will on an integer literal means binary. All three can be preceeded by a minus (-) to make them negative.

A character literal is an integer, written as one character within single quotes, such as ‘x’. The value of a character literal is the numeric value of the character in the machine’s character set. (tag: char literal)

String literals can be concatenated at compile time. (tag: string concatenation)

“hello,“ “ world”  
  
is equivalent to

“hello, world”  
  
 In order to concatenate string variables with string literals, you can use sprintf.

There is one other kind of literal. The enumeration literal. An enumeration is a list of integer values as in (tag: enum literal)

enum boolean { NO, YES };

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value.

enum escapes { BELL = ‘\a’, BACKSPACE = ‘\b’ };

enum months { JAN = 1, FEB, MAR, APR };

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.  
  
[enum part2](#enum_part2)

**-Void types**

-Function returns as void: void exit(int status);

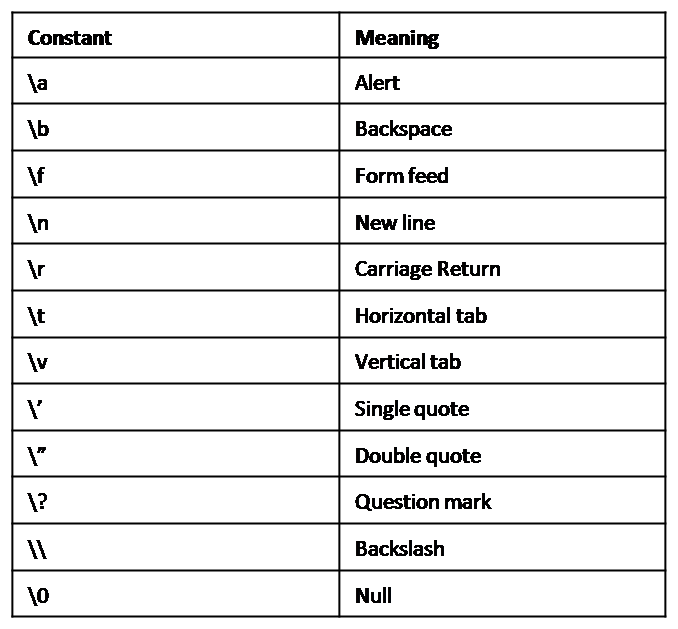
-Function arguments as void:  int rand(void);

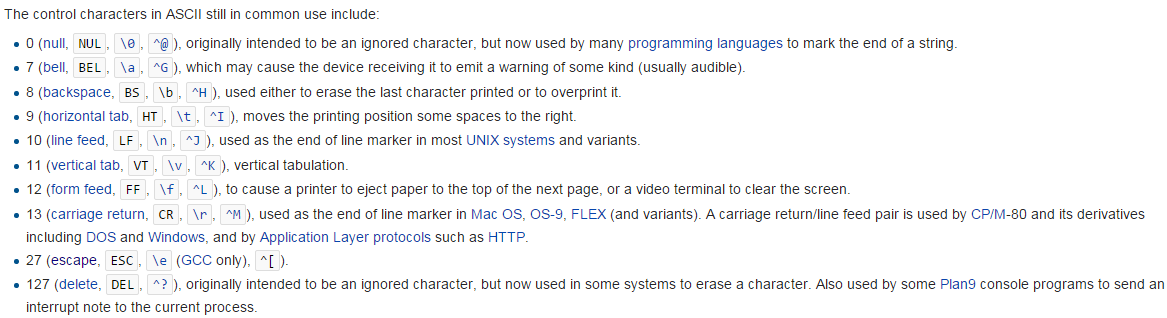
-Pointers to void: A pointer of type void \* represents the address of an object, but not its type. For example, a memory allocation function **void \*malloc( size\_t size );** returns a pointer to void which can be casted to any data type.

**-Escape character and Escape sequences(Control sequences):** Escape characterstrigger escape sequences. “\” is the escape character. Escape character is used to print anything you want to be printed without compiler trying to recognize it as someting else. For example \c will print c and \\c will print \c. \n is an escape sequence.

Escape sequences are single characters. \n is a single character. '\n' stands for the value of the newline character, which is 10 in ASCII. In expressions is just an integer; on the other hand, it is a string literal that happens to contain only one character.







- %% for %.

- \r carriage return:Goes to the beginning of the line and deletes everything in the line.

- \ooo is octal escape sequence and \xhhhh is hexadecimal escape sequence. You can put a hypen (-) before 0 or x to make it negative.  
  
 char \*a = "B\1015"; // B, /101, 5  
 char b = ‘\102’;

printf("%s-%c", a, b);  
  
 Output: BA5-B  
  
 char \*a = "B\x41";  
 char b = ‘\x42’;

printf("%s-%c", a,  b);  
  
 Output: BA-B

**Part 2**

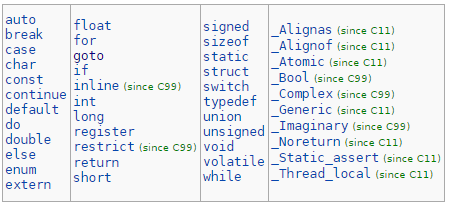
* **Warning and Errors**

**-Warning (warning shown, does run):** Creating variables but not using them.  
  
**-Error:** Syntax error, Using variable without declaration.  
  
 **-Compile time errors (error shown, doesnt run)  
  
 -Syntax error:** Wrong syntax. They are also called compiler errors since compiler show them during compliation phase. Also called compiler error, compile-time error, compliation error.  
  
Forgetting a semicolon.  
When one brace of a block is mising.  
  
 **-Semantic error:** Things that are not meaningful to complier.

b + c = a;  
  
 **-Type error:**   
  
 Using variable without declaration.   
 int x = arena(7); // undeclared function  
 int x = area(7); // wrong number of arguments  
 int x = area(“seven”, 7); // 1st argument has a wrong type  
  
 **-Link time errors:** Errors found by the linker when it is trying to combine object files into an executable program.  
  
 Every function in a program must be declared with exactly the same type in every translation unit in which it is used. We use header files to ensure that.  
 Every function must also be defined exactly once in a program.  
  
 **-Run time(execution time) error/ Logic error  
  
 -Design error/Logic error (error not shown, does run):** Produces wrong output. Cant be detected by complier.   
  
 Using “(a + b) / 2” instead of “a + b / 2”Not using braces might cause problems like “Dangling-else Problem”.  
  
 **-Fatal error/Fatal logic error (error not shown, does run and crash):** No error while compilation. Error happens while the program is running. Infinite loops, dividing by zero, forgetting ampersand while using scanf, lack of memory, input isnt in wanted format or input data file wasnt found at specified location, trying to open a file that wasnt created, hardware errors.  
  
 i = 0;  
 array[--i] = a;

[Warnings, errors in C](http://www.c4learn.com/c-programming/c-pragma-warn-macro-directive/)

* **Keywords**

  
  
**-Storage Classes**  
 **-auto:** These variables are local to their block. Their storage space is reclaimed on exit from the block. Declared when program enters its scope and they are removed from memory when program exits the scope.   
  
Not all local variables are auto. For example a local variable might be static.  
  
 -**extern:** A variable that is outside every function. An external variable. Generally it is used to access a variable in the header file you are using.

[extern part2](#extern_part2)

An external variable must be defined outside functions. This sets aside storage for it.

External variables are always there even when you dont want them. Relying too heavily on external variables leads to programs whose data connections are not at all obvious.

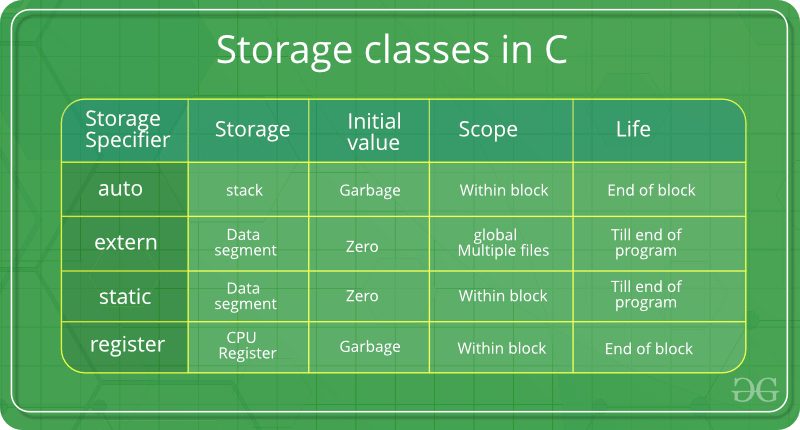
-**static**

If you use the keyword static for a variable in a function, that variable is saved. When you call the function again, the saved version of the variable will be used. Lifetime of the variable is no the block but the entire program.

The static declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled.

-**dynamic** (not a keyword): Dynamic variables can have a different value when the program executes again. The keyword Dynamic is omitted (int x;).

**-register:** If possible, the variable will be stored in a processor register. May give faster access to the variable. If register storage is not possible, then the variable will be auto. Not recommended to use. Compiler will decide what to save in register instead.  
 The register declaration can only be applied to automatic variables and to the formal parameters of a function.

  
  
**-Type Qualifier**  
  
 **-const**: The qualifier const can be applied to declaration of any variable to specify that its value will not be changed. For an array, the const qualifier says that the elements wll not be entered.

const double e = 2.718;

const char msg[] = “warning”;

The const declaration can also be used with array arguments, to indicate that the function does not change the array.

int strlen(const char[]);

-Variable’s value is constant, it doesnt chage. But there is a detail.

C:\Users\Anıl\Desktop\Capture.PNG

A variable has a single value and calling that variable const is enough. But a pointer has two values. The address held by the pointer and the value in that address. If you declare only the variable or only the pointer, you can change the other. For example, in the example above, pointer is const but the variable is points to is not const. We cant change what we are pointing to but we can change the value of the thing we are pointing to.

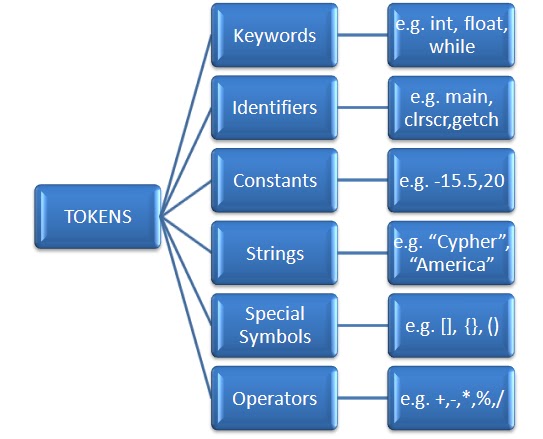
**-volatile:** Volatile variablesmight changebecause of multiple reasons even if code doesnt tell it to change.

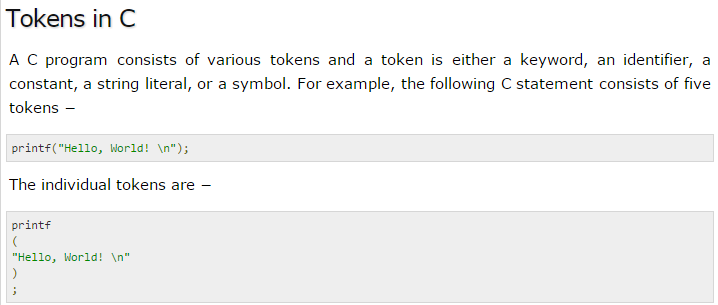
* **Identifier**

An identifier is a name used to identify a variable, function, or any other user-defined item. Identifier cant start with a number. Can start with a character or underscore but not starting with an underscore is convention. Consists of characters(letters, numbers and underscores). ‘-‘ and ‘#’ cant be in the identifier. C is case sensitive. Constant variable identifiers are all uppercase. Object oriented data types such as struct identifiers start with an uppercase letter. If there are multiple words we use camel case or underscores to seperate the words.  
  
-Identifier names dont start with a number because compiler needs to know if this is a number or an identifier. It is easy to understand if first character is never a number.  
  
-Non-obvious literals in code (outside definitions of symbolic constants) are called magic numbers/constants. Use symbolic constants instead.

* **Terms**

-**Tokens**





**-Expression:** An expression is any legal combination of symbols(operands, operators, function calls, etc.) that represents a value.

c = getchar();

The code above is an unusual expression and has the value of the lvalue after the assignment. This type of expressions dont exist in Java. Because in Java, this would be considered a statement, not an expression.

Constant expression is an expression that involves only constants. Such expression may be evaluated during compilation rather than run-time, and accordingly may be used in any place that a constant can occur, as in

#define MAXLINE 1000

char line[MAXLINE + 1]

-**Statement:** Semicolon is a statement terminator. Part of the program that specifies an action and isn’t a preprocessor directive is called a statement.

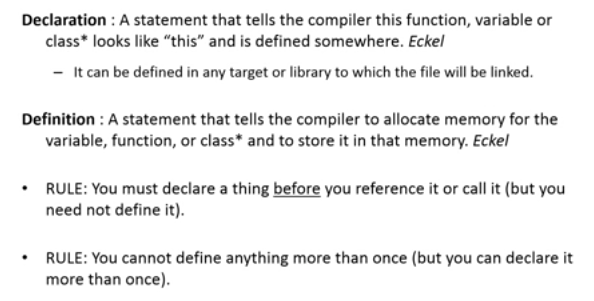
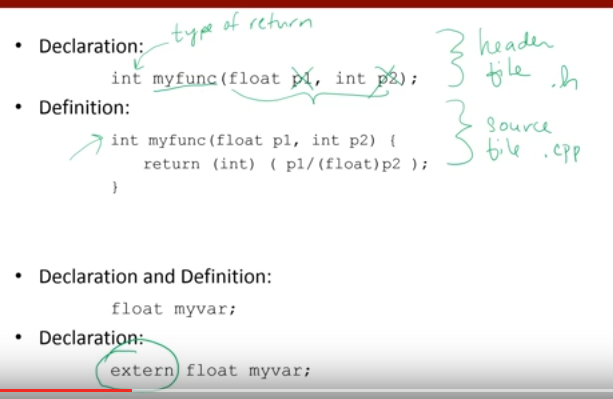
Null statement(empty statement) is a semicolon used in an empty body. Or you could have an empty block ( {} ) or even a block with only a semicolon in it.

for (i = 0; i < 10; line[i++] = 0)

;

**-Block, Compound Statement:** Bracesareused to group statements together into a compound statement, or a block, so that they are syntactically equivalent to a single statement.

-**Whitespace:** A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it. (tag: white space, white space character, whitespace character)  
  
Whitespace is the term used in C to describe space ‘ ‘, tab ‘\t’, newline ‘\n’, vertical tab ‘\v’, feed ‘\v’, carriage return ‘\r’.  
  
**-Declaration, Definition**  
  
[declaration definition part2](#declare_define_part2)



**-Initialization:** (tag: initialize) If the variable in question external or static, the initialization is done once only, conceptually before the program starts executing, and the initializer must be a constant expression. External and static variables are initialized to zero by default.  
 An explicitly initialized automatic or register variable is initialized each time the function or block it is in is entered; the initializer may be any expression. Automatic abd register variables for which there is no explicit initializer, have undefined (i.e., garbage) values.  
 Pointers are initialized to NULL if not explicitly done. Initialization is done each time block is entered.  
 If there are fewer initializers for an array than the number specified, the missing elements will be zero.  
  
If you look at locals window in visual studio during debugging you will see variables are declared and initialized a random value when they get into scope. The line in which you declare it doesnt Change the value and the line where you initialized it gives its first non random value.

**-Comments**

// end of line comment

/\*

...

\*/ traditional comment

/\*\*\*\*\*\*\*\* boxed comments

\* \*

\*\*\*\*\*\*\*\*/

**Part 3**

* **Main Function**

-You dont have to have a main function but then the program is not executable. It is a resource file, a header file. There can be maximum of 1 main function per .c file.

The return value for main should indicate how the program exited. Normal exit is generally represented by a 0 return value from main. Abnormal termination is usually signalled by a non-zero return but there is no standard for how non-zero codes are interpreted.

In C99, int main() can be left without a return value at which point it defaults to returning 0. Whether return 0 should be omitted or not is open to debate.

* **Input/Output**

**-Standart Input and Output** -getch, getchar(conio.h):When you enter a value for getch, it is not displayed on screen. If you use getch(), you don’t need to press enter key after entering the character in the console window because it does not use any buffer, so the entered character is immediately returned without waiting for the enter key. getchar() is mainly used to get character input from the user. Whereas getch() is mainly used to hold the output screen.  
  
 In many environments, a file may be substituted for the keyboard by using the < convention for input redirection: if a program prog uses getchar, then the command line  
  
 prog <infile  
  
causes prog to read characters from infile instead. The switching of the input is done in such a way that prog itself is oblivious to the change; in particular, the string “<infile” is not included in the command-line arguments in argv. Input switching is also invisible if the input comes from another program via a pipe mechanism: on some systems, the command line   
  
 otherprog | prog  
  
runs the two programs otherprog and prog, and pipes the standart output of otherprog into the standart input for prog.

-putchar:Output can usually be directed to a file with >filename: if prog uses putchar,  
  
 prog >outfile  
  
will write the standart output to outfile instead. If pipes are supported,  
  
 prog | anotherprog  
  
puts standart output of prog into the standart input of anotherprog.  
  
 Output produced by printf also finds its way to the standart output.

-gets: Unsecure. Use fgets instead.

[Why gets should not be used](https://stackoverflow.com/questions/1694036/why-is-the-gets-function-so-dangerous-that-it-should-not-be-used)  
  
 [fgets](#fgets)

-puts:puts adds \n terminal to the string even if the string has a \n at the end. **-Formatted Output**  
  
 -f in printf stands for formatted. Prints that drive the user to do something are called prompt. Printf returns the number of characters printed.  
  
 -printf("Result: %f\n", 2 / 4); // Prints 0.0  
  
 -printf("My name is "  
 "Michael Wittmann.\n");  
  
 -The format argument of printf can be an expression too.  
  
 printf((argc > 1) ? “%s ” : “%s”, \*++argv);  
  
-The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next succesive argument to printf. Each conversion specification begins with a % and ends with a conversion character. Between the % and the conversion character there may be, in order:  
  
 A minus sign, which specifies left adjustment of the converted argument.  
 A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left (or right, if left adjustment is called for) to make up the field width.  
 A period, which seperates the field width from the precision.  
 A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.  
 An h if the integer is to be printed as a short, or “l” (letter ell) if as a long. What is the point of this? e, f, g may be preceded by “l” to indicate that a pointer to double rather than float is in the argument list.  
  
 char s[] = "hello, world”;  
   
 :%s: :hello, world:  
 :%10s: :hello, world:  
 :%.10s: :hello, wor:  
 :%-10s: :hello, world:  
 :%.15s: :hello, world:  
 :%-15s: :hello, world :  
 :%15.10s: : hello, wor:  
 :%-15.10s: :hello, wor :  
   
 (tag: dynamic width&precision) A width or precision may be specified as \*, in which case the value is computed by converting the next argument (which must be int). For example, to print at most n characters from a string s,  
  
 printf("%.\*s\n", precision, "test");  
  
 printf("%.\*f\n", precision, 1.23456);  
  
[More on printf](https://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/FUNCTIONS/format.html)  
  
- char s[] = “tes%t”  
 printf(“%s”, s); // prints “tes%t”  
 printf(s); // prints “tes” followed by a tab (the white space character)  
  
-sprintf: Formats the arguments in arg1, arg2, etc., according to format as before, but places the result in string instead of on the standart output; string must be big enough to receive the result.  
  
 int sprintf(char \*string, char \*format, arg1, arg2, ...)  
  
**-Formatted Input**  
  
**-**scanf(%d, &x);

-We need to use & in scanf but not printf because printf needs to access the value, scanf needs to access the address. & wont be needed if you use a variable that points to an address. Examples to this are arrays, structs and pointers.

-while (scanf("%d", &num) == 1)  
  
-When inputing strings with space in them, you can use fgets instead of scanf. Scanf stops when it sees a white space.  
  
-sscanf: Reads from a string instead of the standart input.  
  
 int sscanf(char \*string, char \*format, arg1, arg2, ...)

* **Control Flow**

**-Condition**

Boolean: In C ‘FALSE’ has the value of 0. ‘TRUE’ is anything other than 0. You can assign result of a condition to a numeric variable(char, short, int, long, long long). Statements such as if(false) or if(f) cant be used but if(‘f’), if(‘fa’), if(“fa”) and if(“\n”) can be used.

A non zero value is considered true as well. So !253 will be considered false.

**-Conditional Control Statements  
  
 -if** if (!mark[j] && x)  
  
 if (x && y)  
 **-else** if (condition)  
 {  
 statement(s)  
 }

# else if (condition) { statement(s) } else { statement(s) }

### -Conditional Expression: The operator itself is called the conditional operator. The entire expression is called the conditional expression.

Conditional expression is indeed an expression, and it can be used wherever any other expression can be.

**expr1 ? expr2 : expr3; has the same meaning as if(expr1) expr2 else expr3. But conditional expression can be used as an expresion.**

# s = (x < 0) ? -1 : x \* x;

# result = (grade >= 50) ? "Successful" : "Unsuccessful";

# printf( "%s\n", (count % 2) ? "\*\*\*\*" : "++++++++" );

# printf( “%s\n ”, (not >= 60) ? “Passed” : Failed” );

**-switch**

# switch (expression)

# {

# case constantExpression:

# statement(s)

# break;

# case constantExpression:

# statement(s)

# break;

# default:

# statement(s)

# }

-The expression must be a numeric value and cases must be constant numeric values.

Note:switch–case and if–else-if statements are very similar. But there are two differences between them. First, you cant use intervals in switch statement. You see whether certain values are equal to the expression or not. Second, if you dont use break, codes of cases below will execute aswell. If you want only one case to execute, use break in all cases.

**-while**

# while (condition)

# {

# statement(s)

# }

-Counter controlled while loop as opposed to sentinel controlled while loop

# int i = 0;

# while(i < 5)

# i--;

**-do while:** Executes it’s statement(s) at least once.

# do {

# *statement(s)*

# } while (*condition*);

**-for:** We tend to use the for statement for counter controlled loops. Also, we dont have to increment/decrement by one.

for (i = +0; i < 5; i++)

for (i = 0; i < 10; )

{

printf(“%2d: Hello World\n”, (i + 1));  
 si = i + 1;

}

When you have a calculation in a for header expression or in while header expression it is calculated every time unless the IDE/compiler has an optimizer. So do your own calculation one time and use the calculated value.

-Multiple assignments in for header:

# for (int I = len – 3, j = 0; I >= 0; i--, j++)

# for (i = 0, j = 1; i < 100 && j <= 100; i += 2, y += 1)

**-Unconditional Control Statements**

**-break**

for( ; ; )

{

printf("Enter a number: ");

scanf("%d",&number);

if( number < 0 )

break;

sum += number;

numberCount++;

}

-Break exists the loop or switch it is in.  
  
 **-continue**  
  
 Goes back to the beginning of the loop. Increment/decrement expression is executed before the new iteration when we used continue in a for loop.

# // Printing only odd numbers

# for (int i = 0; i < 10; i++)

# {

# if (i % 2 == 0)

# continue;

# printf(“%d\n”, i);

# } **-goto:** It is advised that even if you will use goto, use it only for rare cases such as when you need to exit a deeply nested structure such as breaking out of two or more loops at once.

# int i = 0;

# start\_point:

# printf("Hello World\n");

# i++;

# if (i < 10)

# goto start\_point;

* **Functions and Program Structure**

-We call functions with arguments. The variables that hold values of arguments in function are called parameters.  
-Functions can return arithmetic result(int, float etc.), pointer, struct, union, void and functions can have these types of values as arguments. If return type is not specified, int is assumed.  
-When declaring methods in C (unlike C++, Java and C#) if there are no parameters then write (void), dont leave it empty. Leaving it empty means unspecified number of parameters.

**-Local variable:** Variables we define in a function are defined only in that function and until that function ends. We can use local variables directly only in their function. In order to use them in order functions we either need to pass them as arguments or use pointers.

if (j > 0)

int i = 100; // 'i' is defined only in the if block.

-**Global variable** They are defined right after include and define statements. They can be used in all functions directly.

**-Variable shadowing:** Global xvariable and local x variable are in different memory addresses.   
 If there are local and global variables in a function with the same name, local one is used.   
 If func1 function takes x as a parameter when there is an x variable in main method and an x global variable, local x will be used.   
  
**-Function Definition**

void odd\_or\_even(int number)

{   
 if (number % 2 == 0)

printf("%d, is an even number.\n", number);

else

printf("%d, is an odd number.\n", number);

}

int main( void )

{

int number;

printf("Enter a number: ");

scanf("%d", &number);

odd\_or\_even(number);

}

**-Function Declaration + Function Definition**

int divide(int, int, int \*);

int main( void )

{

int dividend, divisor quotient, remainder;

dividend = 13;

divisor = 4;

quotient = divide(dividend, divisor, &remainder);

printf("Quotient: %d Remainder: %d\n", quotient, remainder);

return 0;

}

int divide(int dividend, int divisor, int \*remainder)

{

\*remainder = dividend % divisor;

return dividend / divisor;

}

**-Command line arguments**

-They are important for your program specially when you want to control your program from outside instead of hard coding those values inside the code.

-You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''.

-int main(int argc, char \*argv[]);  
 argcrefers to the number of arguments passed, andargv[]is a pointer array which points to each argument passed to the program.  
 It should be noted thatargv[0]holds the name of the program(.c file) itself andargv[1]is a pointer to the first command line argument supplied, and argv[argc - 1] is the last argument. argv[argc] is a null pointer. If no arguments are supplied, argc will be one, otherwise and if you pass one argument thenargc is set at 2.

**-Scope**

[declaration definition part1](#declare_define_part1)

[extern part1, static, register](#extern_part1)

Scope of a name is the part of the program within which the name can be used.

A declaration announces the properties (its type and name) of a variable or a function ; a definition also causes storage to be set aside (both variable values and function bodies are stored in memory as part of the program). A declaration that also fully specifies the entity declared (initialization for variables, function body for functions) is called a definition.

-The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled.

-On the other hand, if an external variable is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an extern declaration is mandatory.  
 It is important to distinguish between the declaration of an external variable and its definition.

If the lines   
  
 int sp;  
 double val[MAXVAL];  
  
appear outside of any function, they define the external variables sp and val, cause storage to be set aside, and also serve as the declaration for the rest of that source file. On the other hand, the lines  
  
 extern int sp;  
 extern double val[];  
  
declare for the rest of the source file that sp is an int and that val is a double array (whose size is determined elsewhere), but they do not create the variables or reserve storage for them.  
  
There must be only one definition of an external variable among all the files that make up the source program; other files may contain extern declarations to access it. (There may also be extern declarations in the file containing the definition.) Array sizes must be specified with the definition, but are optional with an extern declaration.  
 Initialization of an external variable goes only with the definition.  
  
 In file1:  
  
 extern int sp;  
 extern double val[];  
  
 void push(double f) {...}  
  
 double pop(void) {...}  
  
 In file2:  
  
 int sp = 0;  
 double val[MAXVAL];  
  
 Because the extern declarations in file1 lie ahead of and outside the function definitions, they apply to all functions; one set of declarations suffices for all of file1. This same organization would also be needed if the definitions of sp and val followed their use in one file.  
  
**-Recursion**  
   
 unsigned long factorial(unsigned long i)  
 {  
 if (i == 0)  
 return 1;  
 return (i \* factorial(i- 1));  
 }  
  
**-Variable-length Argument Lists**The proper declaration for printf is   
  
 void minprintf(char \*fmt, ...)  
  
where the declaration ... means that the number and types of these arguments may vary. The declaration ... can only appear at the end of an argument list.  
  
 The standart header <stdarg.g> contains a set of macro definitions that define how to step through an argument list. The implementation of this header will vary from machine to machine, but the interface it presents is uniform.  
  
 int ival;  
 // double dval;  
 // char \*sval;  
  
 va\_list ap;  
 va\_start(ap, fmt);  
 ival = va\_arg(ap, int);  
 // dval = va\_arg(ap, double);  
 // sval = va\_arg(ap, char \*);  
  
-More information at page 155-156.

* **Pre Processors (Symbolic Constants)**

Preprocessor executes before compile. The preprocessor parts where the condition is false are omitted just like comments. They dont exist in Java because their usage causes some hardships. Usage of preprocessors have been decreased with newer languages.

**-#include**

For #include "filename" the preprocessor searches in the same directory as the file containing the directive. This method is normally used to include programmer-defined header files.

For #include <filename> the preprocessor searches in an implementation dependent manner, normally in search directories pre-designated by the compiler/IDE. This method is normally used to include standard library header files.

In C, the header file is actually added to the .c file you include it to. In Java, include just makes it so you dont have to use fully qualified name. This is why an .exe file for a hello world C program is 128 KB while it’s .c file is 1 KB.

**-#define**

#define PI 3.14

-Using define for functions:

#define print\_hello(x) int i; for (i = 0; i < (x); i++) printf("Hello\n");  
   
 int main(void)  
 {  
 int print\_count;  
 printf("How many times should it be printed: “);  
 scanf(“%d”, &print\_count);  
 print\_hello(print\_count);

-You can also place ‘\’ at the end of each line to be continued.

# #define print\_hello(x) \ int i; \ for (i = 0; i < (x); i++) \ printf("Hello\n");

**-#undef:** When we use #undef PI, starting from that point, PI will be undefined.

**-#if, #endif**

# #if defined(PI) #define PI ... #endif

**-#ifdef**

# #ifdef PI

# // If PI is defined, calculate area.

# area = PI \* radius \* radius;

# printf("Circle area: %.2f\n", area);

# #else

# // If PI is undefined, print error message.

# printf("ERROR: PI is not defined.\n");

# #endif

**-#ifndef**

#ifndef PI

#define PI 3.14

#endif

area = PI \* radius\* radius;

printf("Area of circle: %.2f\n", area);

# #if #else #endif #if #elif #endif

* **Pointer**

# -A Pointer is a variable that holds an address. Different types of variables have their own pointer type.Pointer size on 16 bit is 16 bit(2 byte), on 32 bit its 32 bit(4byte), on 64 bit its 64 bit(8 byte). -Defining and initializing a pointer, int \*xp = &x; -To use the value (value of int variable will be integer, value of pointer will be address, value of “pointer of pointer” will be address of a pointer) of the variable whose address is held by the pointer, we put indirection operator or dereferencing operator \* in front of the pointer, k = \*p; -To use address of a variable(including pointers) we use the address of operator &. -When we are assigning address of an array to a pointer we dont need the ampersand because array variable already holds the address to the first element of the array. If you define a pointer to the array called ptr, ptr[i] can be used to get address of array elements. \*(ptr + i) can be used to access value of array[i] element. -ptr = month + i assigns address of month[i]. ptr = &month[i] assigns address of month[i]. ptr2 = (ptr + i)[j] assigns address of month[i + j].

# ptr = ptr + i; assigns address of i elements beyond where ptr currently does. val = \*(month + i) assigns value of month[i]. val = \*(ptr + i) assigns value of month[i]. val = ptr[i] assigns value of month[i].

-month[5] is the address of 5th element of month.   
-month[0] and month are pointers that point to the start of the month array.

int i;

int array[6] = { 4, 8, 15, 16, 23, 42 };

int \*ptr;

ptr = array;

for (i = 0; i < 6; i++)  
 printf( "%d\n", \*( ptr + i ));

-There is one difference between an array name and and a pointer that must be kept in mind. A pointer is a variable , so pa = a and pa++ are legal (pa is pointer and a is array). But an array name is not a variable; constructions like a = pa and a++ are illegal.  
-An array parameter is a pointer though. We can use “hello world” or array or ptr as arguments and we can use char s[] or char \*s as parameters. We prefer the latter parameter because it says more explicitly that the parameter is a pointer.  
-We can pass a subarray to a function by using f(&a[2]) or f(a + 2). If one is sure that the elements existm, it is also possible to index backwards in an array; p[-1], p[-2], and so on are syntactically legal, and refer to the elements that immediately precede p[0].  
  
**-Call by reference:** If we want our original argument to change during function or if we want to return multiple values, we use call by reference.  
 Instead of using variables as arguments we use addresses of the variables as arguments in the form of &a, &b. And we take them in the function in the form of int \*x, int \*y. Because the argument was an address and the parameter is a pointer. This way instead of using a copy of value of a and b, we directly work on the values that are in these addresses. In function we use them as \*x, \*y. This allows us to return multiple values.  
  
**-Call by value:** We pass a copy of the value to the function.

**-Pointer/Address arithmetic:** The valid pointer operations assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning or comparing to zero. All other pointer arithmetic is illegal. It is not legal to add two pointers, or to multiply or divide or shift or mask them, or to add float or double to them, or even, except for void \*, to assign a pointer of one type to a pointer of another type without a cast.

-Value of null pointer is 0. Null pointer points to nothing. Pointers and integers are not interchangeable. Zero is the sole exception: the constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero. The symbolic constant NULL is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. NULL is defined in <stdio.h>.  
-Two pointers can be subtracted to find the adress amount between two adresses. Addition, multiplication, division are not allowed. You can use “++”, “--“, “+”, “-“ with adresses. Such as   
  
 ptr + c; // The address that is c units ahead ptr.  
 ptr++; // Next address.

\*++ptr; // Value of variable in next address.  
 \*ptr++; // Value of variable in current address and then increment.  
  
ptr2 = ptr + i // Assigns address of month[i]  
  
If p and q point to members of the same array, then relations like ==, !=, <, >= work properly.  
  
**-Character Pointers and Functions**

-char arr[5] = {'a','b','c','d','e',0};   
 char \*ptr = arr; // Same as char \*ptr = &arr[0]  
 printf("\nvalue: %c", ptr[0]);

-char \*pmessage;  
 pmessage = “now is the time”;

-char amessage[] = “now is the time”;  
 char \*pmessage = “now is the time”;  
  
amessage is an array. Individual characters within the array may be changed but amessage will always refer to the same storage. pmessage is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point somewhere else, but the result is undefined if you try to modify the string contents.  
  
-Two different way to implement strcpy. When the expression is equal to zero (end of char array), while loop will terminate.  
  
 void strcpy(char \*s, char \*t)  
 {  
 int i = 0;  
 while (s[i] = t[i])  
 i++;  
 }  
  
 void strcpy(char \*s, char \*t)  
 {  
 while (\*s++ = \*t++)  
 ;  
 }  
  
**-Pointer Arrays, Pointers to Pointers** char \*month\_name(int n)  
 {  
 static char \*name[] = { “Illegal month”, “January”, “February”, “March”,   
 “April”, “May”, “June”, “July”, “August”, “September”, “October”,   
 “November”, “December” };  
  
 return (n < 1 || n > 12) ? name[0] : name [n];  
 }  
  
-Pointer arrays vs multi dimensional arrays: The important advantage of the pointer array is that the rows of the array may be of different lengths. If we have “int a[10][20];” and “int \*b[10]”, then a[3][4] and b[3][4] are both syntactically legal references to a single int. By far the most frequent use of arrays of pointers is to store character strings of diverse lengths.-When we define a pointer who will hold address of a pointer, we use \*\* instead of \* and so on. int r = 50;  
 int \*p;  
 int \*\*k;  
 int \*\*\*m;  
 printf( "r: %d\n", r );  
 p = &r;  
 k = &p;  
 m = &k;  
 \*\*\*m = 100  
 printf( "r: %d\n", r );  
  
\*\*\*m = 100 ends up assigning 100 to r.  
  
**-Pointers to void:** A pointer of type void \* represents the address of an object, but not its type. It can be used to hold any type of pointer but cannot be dereferenced.   
 A memory allocation function,  
  
 void \*malloc(size\_t size);  
returns a pointer to void which can be casted to any data type.

* **Array**

-Below are two different ways to initialize the array. When we dont explicitly tell compiler what the array size is and initializers are present, compiler counts element number.

int array[7] = { 0, 0, 0, 0, 0, 0, 0 };

int array[7] = { 0 };

Memset: Used to assign a value to the entire array . ptr is starting address of memory to be filled. x is value to be filled. n is number of bytes to be filled starting from ptr such as 4 \* sizeof(int).

memset(void \*ptr, int x, size\_t n);  
  
-Length of array can be a literal or a constant expression.

ptr = month; // ptr gets the address of month[0] / month

elm = ptr[3]; // elm gets the address of month[3]

-We dont have to specify the length of the array if we are going to initialize it as we are defining it. In multiple dimension arrays you dont have to specify the length of last dimension if we are going to initialize it as we define it.

int array1[] = { 4, 8, 15, 16, 23, 42 };

float array2[] = { 11.5, -1.6, 46.3, 5, 21.56 };

-To access 25th element of array a, you use a[24] since elements start from 0th.

-Arrays and functions: When we pass arrays or strings to functions, we dont need an ampersand because arrays and strings hold an address in the first place.

void show\_elements(int array[])

{

int i;

for (i = 0; i < 5; i++)

printf("%d\n", array[i]);

}

int main(void)

{

int array[5] = { 55, 414, 7, 210, 15 };

show\_elements(array);

return 0;

}

**-Multi dimension array**

int table[3][4] = { 8, 16, 9, 52, 3, 15, 27, 6, 14, 25, 2, 10 };  
int table[3][4] = { {8, 16, 9, 52}, {3, 15, 27, 6}, {14, 25, 2, 10} };

Row 0 : 8 16 9 52

Row 1 : 3 15 27 6

Row 2 : 14 25 2 10  
  
-big[big[2][2]][big[2][2]] = 177; // This means big[5][5] = 177;

-When you are initializing an array, if you enter less values than the number of elements, extra elements will be set to 0.

int table[3][4] = { {8, 16}, {3, 15, 27} };

Row 0 : 8 16 0 0

Row 1 : 3 15 27 0

Row 2 : 0 0 0 0

-If a multi dimensional array is to be passed to a function, the parameter declaration in the function must include sizes of all dimensions other than first one  
  
-You can assign address of a multiple dimension array to a pointer that points to the same data type and use that pointer as a single dimension array. This is because elements are kept one after another in memory.  
  
 int \*p = table[0];

* **Struct**

-An optional name called a structure tag may follow the word struct The variables names in a structure are called members.

-Struct must be defined above main function. But structs can be made local or global.

-Unlike arrays and strings, when we are passing structs and enums to functions, we need to use & because these two dont hold the addresses.

void push(struct stack \*stack1, value1); // Prototype of push function

push(&stack, value); // push function is being called in main function.

-To copy your data to your brothers data, all you have to do is, "brother = you".

struct personal\_information  
{

char name[40];

int height;

};

struct personal\_information person1;  
struct personal\_information person2;

strcpy(person1.name, "Michael");

person1.height = 174;  
  
person2 = { “David”, 171 };

-Alternative declaration and initialization in structs

struct personal\_information  
{

char name[40];

int height;

} person1 = { "David", 171 }, person2, person3;

### For a struct with date of birth, instead of using { "Charles", 164, 23, 3, 1980 }, you can use { "Charles", 160, {23, 3, 1980} }. Which is more readable. -Nested structs struct rect { struct point pt1; struct point pt2; }; struct rect screen = { { 0, 0 }, { 2, 2 } }; screen.pt1.x // Refers to the x coordinate of the pt1 member of screen. -Structs and Functions

struct personal\_information take\_information(void)

{

struct personal\_information person;

printf("Name> ");

gets(person.name);

printf("Height> ");

scanf("%d", &person.height);

return person;

}

void show\_information(struct personal\_information person)

{

printf("Name: %s\n", person.name);

printf("Height: %d\n", person.height);

}

int main()

{

struct personal\_information person = take\_information();

show\_information(person);

return 0;

}

-A structure member or a tag and an ordinary (i.e. non-member) variable can have the same name without conflict. Also, as you can see structs are pass by value like normal variables and unlike arrays and strings.  
  
 struct point

{

int x;

int y;

};

struct point makepoint(int x, int y)

{

struct point temp;

temp.x = x;

temp.y = y;

return temp;

}  
  
 struct point addpoint(struct point p1, struct point p2)

{

p1.x += p2.x;

p1.y += p2.y;

return p1;

}

int main()

{

struct point point1 = makepoint(1, 1);

struct point point2 = makepoint(2, 2);

struct point point3;

point3 = addpoint(point1, point2);

return 0;

}

-If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure.  
  
 struct point \*p;  
  
If p points to a point structure, \*p is the structure and (\*p).x and (\*p).y are the members.  
  
 struct point origin, \*p;  
  
 p = &origin;  
 printf(“origin is (%d,%d)\n”, (\*p).x, (\*p).y);  
  
The parantheses are necessary in (\*p).x because the precedence of the structure member operator . is higher than \*. The expression \*p.x means \*(p.x), which is illegal here because x is not a pointer.  
  
Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then  
  
 p->x  
  
refers to x member of the structure. So we could write instead  
  
 printf(“origin is (%d,%d)\n, p->x, p->y”);  
  
**-Arrays of Structures**

struct personal\_information person[3] = { "Michael", 170, { 17, 2, 1976 },   
 "David", 178, { 14, 4, 1980 },  
 "Charles", 176, { 4, 11, 1983 } };

for (i = 0; i < 3; i++)  
 {

printf("ID no.: %d\n", (i + 1));

printf("name: %s\n", person[i].name);

printf("Height: %d\n", person[i].height);

printf("Date of Birth: %d/%d/%d\n\n", person[i].date.day,

person[i].date.month, person[i].date.year);  
 }

struct key

{

char\* word;

int count;  
 } keytab[NKEYS];

struct key

{

char\* word;

int count;  
 };  
  
 struct key keytab[NKEYS];

-sizeof:   
  
 sizeof object  
  
 sizeof(type name)

Yield an integer equal to the size of the specified object or type in bytes.  
  
**-Accessing struct arrays with pointers**

### It is possible to access struct date from pointer.

struct personal\_information \*ptr;

for (i = 0, ptr = &person[0]; ptr <= &person[2]; ptr++, i++)   
{

printf("ID no.: %d\n", ( i + 1 ));

printf("Name: %s\n", ptr->name);

printf("Height: %d\n", ptr->height);

printf("Date of Birth: %d/%d/%d\n\n", ptr->date.day, ptr->date.month,

ptr->date.year);

}

### -Self-referential Structures -It is illegal for a structure to contain an instance of itself, but a pointer to an instance of itself is allowed. struct tnode { char \*word; int count; struct tnode \*left; struct tnode \*right; } -Dynamic Structs (stdlib.h)

struct personal\_information \*ptr;

ptr = calloc(1, sizeof(struct personal\_information));

free(ptr);

### -You shouldnt cast result of malloc/calloc when assigning it to a pointer(like the example below). Since the result is type void \*, it is automatically and safely promoted to any other pointer type. You need to do the cast if for C++ though. int \*ip; ip = (int \*) malloc(n, sizeof(int)); -It is recommended that you use “sizeof \*ip” instead of “sizeof(int)”. That way when you change types, you only need to change type of pointer. [dynamic storage allocation part2](#dynamic_storage_allocation_part2) -Typedef

### In order to use “dog dog1 = ...” instead of “struct dog dog1 = ...” we use one of the following. typedef struct dog dog; // 1 struct dog { char\* name; int age; }; typedef struct dog // 2 { char\* name; int age; } dog; -You can even create a short name for a pointer to a struct. If you use the code below, NODEPTR will be a pointer to a struct of type node. typdef struct node \*NODEPTR;

typedef int bool;

#define TRUE 1

#define FALSE 0

bool f = FALSE;

if (f) { ... }

typedef enum { FALSE, TRUE } boolean;

boolean b = FALSE;

if (b) { ... }

In effect, typedef is like #define, except that since it is interpreted by the compiler, it can cope with textual substitutions that are beyond the capabilities of the preprocessor. For example,   
  
 typedef int (\*PFI)(char \*, char \*);  
  
creates the type PFI, for “pointer to function (of two char \* arguments) returning int,” which can be used in contexts like   
  
 PFI strcmp, numcmp;  
  
 Besides purely aesthetic issues, there are two main reasons for using typedefs. The first is to parameterize a program against portability problems. If typedefs are used for data types that may be machine-dependent, only the typedefs need change when the program is moved. One common situation is to use typedef names for various integer quantities, then make an appropriate set of choices of short, int and long for each host machine. Types like size\_t and ptrdiff\_t from the standart library are examples.  
 The second purpose of typedefs is to provide better documentation for a program—a type called Treeptr may be easier to understand than one declared only as a pointer to a complicated structure.  
  
**-Enum**  
  
[enum part1](#enum_part1)  
  
 We define a new variable type and specify the values it can take.   
 Unlike arrays and strings, when we are passing structs and enums to functions, we have to use &.

enum boolean  
{  
 false = 0,  
 true = 1  
};

enum boolean isTrue;

isTrue = true;  
  
 // Unlike Java, we dont have to use boolean.true

if (isTrue == true)

printf("True\n");

enum boolean  
{  
 false = 0,  
 true = 1  
} isTrue;

isTrue = true;

if (isTrue == true)

printf("True\n");

**-Union:** A union is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage without embedding and machine-dependent information in the program.  
  
 union u\_tag  
 {  
 int ival;  
 float fval;  
 char \*sval;  
 } u;  
  
 Unions may occur within structures and arrays, and vice versa. The notation for accessing a member of a union in a structure (or vice versa) is identical to that for nested structures  
  
 struct  
 {  
 char \*name;  
 int flags’  
 int utype;  
 union  
 {  
 int ival;  
 float fval;  
 char \*sval;  
 } u;  
 } symtab[NSYM];  
  
the member ival is referred to as  
  
 symtab[i].u.ival  
  
and the first character of the string sval by either  
  
 \*symtab[i].u.sval  
 symtab[i].u.sval[0]  
  
 A union may only be initialized with a value of the type of its first member.  
  
**-­Bit-fields:** C offers the capability of defining and accessing fields within a word directly rather than by bitwise logical operators (page 149). A bit-field, or field for short, is a set of adjacent bits within a single implementation-defined storage unit that we will call a “word”.  
  
 struct  
 {  
 unsinged int is\_keyword : 1;  
 unsinged int is\_extern : 1;  
 unsinged int is\_static : 1;  
 } flags;  
  
This defines a variable called flags that contains three 1-bit fields. The number following the colon represents the field width in bits. The fields are declared unsigned int to ensure that they are unsigned quantities.  
  
 Individual fields are referenced in the same way as other structure members: flag.is\_keyword, flag.is\_extern, etc. Fields behave like small integers, and may participate in arithmetic expressions just like other integers. Thus the previous examples may be written more naturally as  
  
 flags.is\_extern = flags.is\_static = 1;  
  
to turn bits on;  
  
 flags.is\_extern = flags.is\_static = 0;  
  
to turn them off; and  
  
 if (flags.is\_extern == 0 && flags.is\_static == 0)  
  
to test them.

* **File Access:** A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices.

### -You can use the fopen() function to create a new file or to open an existing file. This call will initialize an object of the type FILE, which contains all the information necessary to control the stream.

FILE \*fp;  
 fp = fopen(“10letter.txt”, “w”);  
 filecopy(fp, stdout);  
 fclose(fp);  
  
 void filecopy(FILE \*ifp, FILE \*ofp)  
 {  
 int c;  
 while ((c = getc(ifp)) != EOF)  
 putc(c, ofp);  
 }  
  
-fclose is called automatically for each open file when a program terminates normally. You can close stdin and stdout if they are not needed. They can also be reassigned by the library function freopen.  
  
-Opening modes:  
  
 r**:** Opens an existing file to read.  
 w**:** Creates the file if it doesnt exist, resets if it does exist and starts writing from beginning.  
 a**:** Creates the file if it doesnt exist. If it exists, starts writing at the end of the file.  
 r+**:** Opensan existing file to read or write.  
 w+**:** Creates thefile if it doesnt exist, resets if it does exist and starts writing from beginning. For writing and reading.  
 a+**:** Creates file if it doesnt exist. If it exists, starts writing at the end of the file. For writing and reading.  
  
-If you are going to handle binary files, then you will use following access modes instead of the above mentioned ones

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

-The fclose(word) function returns zero on success, or EOF if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF(tag: End of file) is a constant defined in the header file stdio.h. EOF is equal to CTRL+D in UNIX/Linux/MacOS and CTRL+Z in windows. (tag: ctrl + d, ctrl + z)  
  
-If file does not open, fp will be NULL (0). If file opens, fp will be non-zero. fp = fopen("test.txt", "r");

if (fp)

{

printf("%c", getc(fp));

fclose(fp);  
 }-feof(fp): Returns 1(non-zero) if file pointer is at the end of the file, returns 0 if not.   
 rewind(fp); Moves file pointer position to the beginning.   
  
 In order to say do the following as long as we dont get EOF,  
  
 if (!feof(stdin))

printf("%c", getc(fp));  
  
 if (!feof(fp))

printf("%c", getc(fp));  
  
 while (!feof(fp))

printf("%c", getc(fp));   
  
 Or you could use the code below of immediately getting the next character is not a problem.  
  
 while (c = getc(fp) != EOF)  
 putc(c, fp);

-When a C program is started, the operating system environment is responsible for opening three files and providing file pointers for them. These files are the standard input, the standard output, and the standard error; the corresponding file pointers are called stdin, stdout, and stderr, and are declared in <stdio.h>. Normally stdin is connected to the keyboard and stdout and stderr are connected to the screen, but stdin and stdout may be redirected to file or pipes.  
  
-getc(fp)  
  
 putc(char, fp)  
  
 fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);  
  
 fprintf(fp, “x = %d”, x);  
  
 fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);  
  
 When fscanf reads from a file, file pointer goes to the right of the latest part it read. fscanf reads until it sees a space.

fscanf(fp1Trun, "%f,", &fp1Number);

fgets(string, maxLength, fp): In order to skip a line, you can read the line using fgets and ignore it.

fgets(fp1String, 77, fp1Trun);  
  
 [gets](#gets)  
  
 fputs(string, fp);  
  
 getw(fp): Writes an integer into a file.  
  
 putw(i, fp): Reads an integer value from a file.  
  
-Getting OS username, creating addresses  
  
 // address string gets the desktop directory of current computer.

sprintf(address, "%s\\Desktop\\", getenv("USERPROFILE"));

// We copy address of directory to address of our files

// and then add names of files.

sprintf(address1, "%s1.csv", address);

sprintf(address2, "%s2.csv", address);

sprintf(addressa, "%sa.csv", address);

printf(addressb, "%sb.csv", address);

sprintf(addressc, "%sc.csv", address);  
  
**-Error handling—Stderr and Exit**   
  
 char \*prog = argv[0];  
  
 if ((fp = fopen(\*++argv, “r”)) == NULL)

{  
 fprintf(stderr, “%s: can’t open %s\n”, prog, \*argv);  
 exit(1);  
 }  
  
 if (ferror(stdout))  
 {  
 fprintf(stderr, “%s: error writing stdout\n”, prog);  
 exit(2);  
 }  
  
 exit(0);  
  
 Output written on stderr normally appears on the screen even if the standard output is redirected.  
 The program signals errors two ways. First, the diagnostic output produced by fprintf goes onto stderr, so it finds its way to the screen instead of disappearing down a pipeline or into an output file. We included the program name, from argv[0], in the message, so if this program is used with others, the source of an error is identified.  
 Second, the program uses the standard library function exit, which terminates program execution when it is called. The argument of exit is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a sub-process. Conventionally, a return value of 0 signals that all is well; non-zero values usually signal abnormal situations. Exit calls fclose for each open output file, to flush out any buffered output.

return vs exit: return and exit both leave the program when used in main function. Using return in a non-main function causes the function to return to its caller. Using exit in a non-main function leaves the program.  
 The function ferror returns non-zero if an error occured on the stream fp.  
  
 int ferror(FILE \*fp)  
  
 Although output errors are rare, they do occur (for example, if a disk fills up), so a production program should check this as well. feof returns non-zero if end of file has occured on the specified file.  
  
 int feof(FILE \*fp)

* **Miscellaneous Functions**

**-String Operations** (string.h)

-A 30 character long char array can have a maximum of 29 character because the last character must be NULL ‘\0’.

-In C, you use apostrophe(‘’) for characters and quotes for strings(“”). Without these, we either take value of a variable with the same name or ascii value of the character.

-Just like arrays, when we are defining a string, we dont have to specify its length if we are initializing it at the same time.   
  
 char name[] = { 'M', 'I', 'C', 'H', 'A', 'E', 'L', '\0' };  
 char lastName[5] = { 'T', 'E', 'D', 'D', '\0' };  
 printf("%s %s\n", name, lastName);

-Just like arrays, we dont need to add an ampersand before string idenfitiers when we are passing them to a function since they are already pointers.

strlen(s) Returns lenght of string.

strcpy(s, t) Copy t to s.  
strncpy(s, t, n) Copy at most n characters of t to s.  
strcmp(s, t) Returns negative if s < t, zero if s == t, positive for s > t  
strncmp(s, t, n) Compares only first n characters.  
strcat(s, t) Concatenate t to end of s.  
strncat(s, t, n) Concatenate n characters of t to end of s.

strstr(s, t) Searching t in s or NULL if not present. Case sensitive.  
atoi(s) Extracts the integer value from a string.  
atof(s) Extracts the floating point value from a string.  
strchr(s, c) Return pointer to first c in s or NULL if not present  
strrchr(s, c) Return pointer to last c in s or NULL if not present

char adres[] = "Terry Avenue Seattle Washington";

char \*first\_point, \*last\_point;

first\_point = strchr(address, 'e');

last\_point = strrchr(address, 'e');

if (first\_point != NULL)  
 {

printf("First match: %d\n", first\_point - address);

printf("Last match: %d\n", last\_point - address);

else

printf("No match found.\n");  
  
sprintf(s, t, arg1, ...) String.format of C.

char str[80];  
 sprintf(str, "Value of Pi = %f", 3.14);  
 sprintf(hashString, "%d.txt", hashResult);  
  
**-Character Class Testing and Conversion** (ctype.h): In the following, c is an int that can be represented as an unsigned char, or EOF. The functions return int.  
  
isalpha(c) non-zero if c is alphabetic, 0 if not  
isupper(c) non-zero if c is upper case, 0 if not  
islower(c) non-zero if c is lower case, 0 if not  
isdigit(c) non-zero if c is digit, 0 if not  
isalnum(c) non-zero if isalpha(c) or isdigit(c), 0 if not  
isspace(c) non-zero if c is blank, tab, newline, return, formfeed, vertical tab  
toupper(c) return c converted to upper case  
tolower(c) return c converted to lower case  
  
**-ungetc**: Pushes the character c back onto file fp, and returns either c, or EOF for an error.  
  
 int ungetc(int c, FILE \*fp)  
  
**-Command Execution:** The function system(char \*s) executes the command contained in the character string s, then resumes execution of the current program.  
  
 system(“date”);  
  
**-Storage Management** (stdlib.h)  
  
[dynamic storage allocation part1](#dynamic_storage_allocation_part1)  
  
[Stack. Heap - stackoverflow](https://stackoverflow.com/questions/10200628/heap-memory-in-c-programming)  
[Stack, Heap - hawaii.edu](https://www2.hawaii.edu/~walbritt/ics212/examples/HeapStack.htm)  
  
-Stack vs Heap  
 If a function allocates local variables (on the stack) then when function returns, the on-stack variables are gone.   
 A function can allocate a bunch of heap memory and fill it with something, that memory will still be valid after function returns. A drawback of using the heap is that if you forget to release heap-allocated memory, you may exhaust it. The failure to release heap-allocated memory is sometimes called a memory leak.  
  
-The functions malloc and calloc obtain blocks of memory dynamically. Global and static variables are kept in heap too.  
  
-malloc and calloc return a pointer to enough space for an array of n objects of the specified size, or NULL if the request cannot be satisfied In calloc, the storage is initialized to zero.

pointerName = malloc(elementCount, sizeOfElement);

array = calloc(elementCounter, sizeof(int));

if (array == NULL)

printf("Insufficient memory!\n");

-free(p) frees the space pointed by p, where p was originaly obtained by a call to malloc or calloc. Keep in mind that all the space obtained by the program will be freed at the end of the program.

void free(void \*ptr)

A typical but incorrect piece of code is this loop that frees items from a list:  
  
 for (p = head; p != NULL; p = p->next) // Wrong  
 free(p);  
  
 for (p = head; p != NULL; p = q) // Right  
 {  
 q = p->next;  
 free(p);  
 }  
  
**-Mathematical Functions** (math.h):Here are some of the more frequently used functions of math.h. Each takes one or two double arguments and returns a double.  
  
sin(x) sine of x in radians  
cos(x) cosine of x in radians  
atan2(y, x) arctangent of y/x in radians  
exp(x) exponential function exlog(x) natural (base e) logarithm of x (x > 0)  
log10(x) common (base 10) logarithm of x (x > 0)  
pow(x, y) xysqrt(x) square root of x (x >= 0)  
fabs(x) absolute value of x **-Random number generation** (stdlib.h, time.h): rand() and srand() needs stdlib.h. time() needs time.h. Randomization is performed as a function that has a starting value, namely the seed. So, for the same seed, you will always get the same sequence of values. If you only have rand() in your program, it will use the same sequence of random numbers every time the program is executed.  
  
 x = rand();  
  
In order to put the number in the format we want, we use “rand() % n1” or “rand() % n1 + n2”.

Using the same sequence of numbers might be useful for testing. But if you want a random number every time, you need to set the seed either yourself or using time.

unsigned feed;  
  
 scanf("%d", &feed);  
 srand(feed);  
 x = rand();  
  
 or,  
  
 srand(time(NULL));  
 coinToss = rand() % 2;  
  
To produce random floating-point numbers greater than or equal to zero but less than one is   
  
 #define frand() ((double) rand() / (RAND\_MAX + 1.0))

* **The Unix System Inferface**