

# Math, Logic, n' Bits

James Oswald

# The Basic C Program ([link](#))

```
// Type your code here, or load an example.

#include<stdio.h> //Gives us the ability to use printf()

//declare the entry point to the program, all our code goes between the braces { }
int main() {
    printf("Hello World!"); //prints hello world to stdout (the terminal)
}
```

# Basic Output

Printf is a function, you can view this as a special black box that lets us put text in and have it print text to the screen:

```
Printf("Any text in these quotes gets printed to the  
screen");
```

# Statements

- C programs are made out of ***statements***,
- statements are code that tell the computer what to do
- Statements end with a semicolon (;).
- Statements do not need to be on a new line but often are

Three types of statements we care about now:

- Declarations      Create boxes to put values in
- Assignments      Put values in those boxes
- Calls      Run code somewhere else

# Declarations (Variables)

**Variables** are named boxes in which we can store values

To create a variable we need to ***declare it***

Declarations take the form:

(OptionalCVQualifiers) ***Type*** VariableName;

# Variable Names

## RULES FOR NAMING C VARIABLE:

1. Variable name must begin with letter or underscore.
2. Variables are case sensitive
3. They can be constructed with digits, letters.
4. No special symbols are allowed other than underscore.
5. sum, height, \_value are some examples for variable name

# Variable Types

## Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges –

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

## Floating-Point Types

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision –

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

# CV Qualifiers

## Const and Volatile

Prefixing (or postfixing) a declaration with `const` or `volatile` applies special modifiers to the box that give it certain properties.

`Const` means that you can not edit what's in the box ever after an initial assignment. The compiler will give you an error if you try editing what's in the box.

`Volatile` means that this box may be changed by magic outside forces or is needed by some outside thing the compiler has no idea about. Hence In no circumstances can the compiler optimize it out of existence.



# Sample Declarations

LEGAL:

```
char myByte;
```

```
unsigned int age;
```

```
int moneyInMyBankAccount;
```

```
float height;
```

```
const float pi;    //west const
```

```
float const pi;    //east const
```

```
volatile unsigned int hardwareFlags;
```

```
const volatile unsigned char flags;
```

# Assignments

Assignments take the form of: *Lvalue = Rvalue or Lvalue*

An lvalue (locator value) represents an object that occupies some identifiable location in memory (i.e. has an address).

rvalues are defined by exclusion. Every expression is either an lvalue or an rvalue, so, an rvalue is an expression that does not represent an object occupying some identifiable location in memory.

Ex, constants like 10 and 20 are rvalues, they do not refer to a memory location (“a box”) where as the variable “int x;” referenced as x is an lvalue, its the name of a location in memory.

# Complex rvalues

The results of complex computations with operators are rvalues

An Rvalue combined with an Lvalue is an rvalue, an LValue combined with an lvalue is an rvalue.

IE: (assuming `int x;`)

`x` is an lvalue

`10` is an rvalue

`x + 10` is an rvalue

`x + x` is an rvalue

# Operators

Take value(s) and return a new value!

The results may depend on the types.

Different classes of operators:

Arithmetic operators: take numbers and do math with them to produce an new rvalue

Relational operators: take numbers and return 1 or 0 depending if the relation holds

Logical operators: takes 1 or 0 and returns 1 or 0 based on logic rules

Bitwise operators: manipulates numbers at the bit level.

Assignment operators: shorthand operators modify the provided lvalue with some other operator.

# Arithmetic Operators


The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then –

Show Examples [↗](#)

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.	$A-- = 9$

## Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then –

Show Examples 

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.


## Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then –

Show Examples [↗](#)

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.

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then –

Show Examples 

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 One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e., -0111101
<<	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



## Assignment Operators

The following table lists the assignment operators supported by the C language –

Show Examples [↗](#)

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	$C = A + B$ 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 $C = C * A$
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	$C /= A$ is equivalent to $C = C / A$
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	$C \% A$ is equivalent to $C = C \% A$
<<=	Left shift AND assignment operator.	$C <<= 2$ is same as $C = C << 2$
>>=	Right shift AND assignment operator.	$C >>= 2$ is same as $C = C >> 2$
&=	Bitwise AND assignment operator.	$C \&= 2$ is same as $C = C \& 2$
^=	Bitwise exclusive OR and assignment operator.	$C \wedge= 2$ is same as $C = C \wedge 2$
=	Bitwise inclusive OR and assignment operator.	$C  = 2$ is same as $C = C   2$