EEEE1042 - Lecture 2
Tokens, keyword, identifiers
Operators, Literals, Separators
Autumn Semester 2022.

Dr. Chan, Kheong Sann
kheongsann.chan@nottingham.edu.my
University of Nottingham Malaysia
Department of Electrical and Electronic Engineering

EEEE1042 C Programming: Scheduled classes

EEEE1042: for EE students, EEEE1032: for Mecha students.

Week	Dates	Lecture	EEEE1042 Practical	EEEE1032 Practical	Assessment
4	Sep 26 – 30	Thu2-4pm			
5	Oct 3 – 09	Thu2-4pm	Mon3-6pm	Wed3-6pm	
6	Oct 10 – 14	Thu2-4pm	P.H.	Wed3-6pm	
7	Oct 17 – 21	Thu2-4pm	Mon3-6pm	Wed3-6pm	PT1 5%
8	Oct 24 – 28	Thu2-4pm	P.H.	Wed3-6pm	
9	Oct 31 –Nov 04	Project Week 1			
10	Nov 07 – 11	Thu2-4pm	Mon3-6pm	Wed3-6pm	PT2 5%
11	Nov 14 – 18	Thu2-4pm	Mon3-6pm	Wed3-6pm	CW1 10%
12	Nov 21 – 25	Project Week 2			
13	Nov 28 –Dec 04	Thu2-4pm	Mon3-6pm	Wed3-6pm	PT3 5%
14	Dec 05 – 09	Project Week 3			
15	Dec 12 – 16	Thu2-4pm	P.H.	Wed3-6pm	PT4 5%
16	Dec 19 – 23	Study Week			CW2 30%
17-18	Dec 26 – Jan 06	Study Weeks			
19-20	Jan 09 – 21	Final Exam (40%)			

Outline EEEE1042 C Lecture 2:

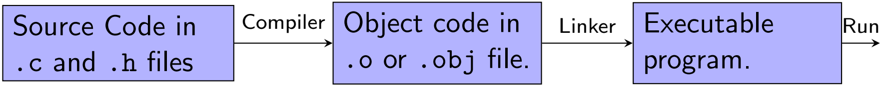
C Hello World Program

```c
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv) {
  printf("Hello World\n");
  return(0);
}
```

Programming practices for modularity, maintainability and reuseability:

- Always comment your code.
- Any code that performs a non-trivial self-contained task, separate into its own subfunction and comment.
- Put a set or family of subfunctions into their own file.



C Hello World Program

```c
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv){
   printf("Hello World\n");
   return(0);
}
```

The `main` function is the entry point where the compiler starts creating the executable. `int argc` is the number of parameters passed to the program (an integer), `char **argv` is a list of `argc` strings, each string is one of the input arguments.

Programming practices for modularity, maintainability and reuseability:

- Always comment your code.
- Any code that performs a non-trivial self-contained task, separate into its own subfunction and comment.
- Put a set or family of subfunctions into their own file.

Source Code in `.c` and `.h` files $\xrightarrow{\text{Compiler}}$ Object code in `.o` or `.obj` file. $\xrightarrow{\text{Linker}}$ Executable program. $\xrightarrow{\text{Run}}$

C Hello World Program

```
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv) {
   printf("Hello World\n");
   return(0);
}
```

The `printf()` function is C's main way of outputing text to stdout (the screen). The compiler knows what to do with this command because it has been **declared** in the header file `stdio.h`. This header file declares all the standard input output functions that come together with C.

Programming practices for modularity, maintainability and reuseability:

- Always comment your code.
- Any code that performs a non-trivial self-contained task, separate into its own subfunction and comment.
- Put a set or family of subfunctions into their own file.

Source Code in `.c` and `.h` files $\xrightarrow{\text{Compiler}}$ Object code in `.o` or `.obj` file. $\xrightarrow{\text{Linker}}$ Executable program. $\xrightarrow{\text{Run}}$

Tokens

A **token** in C or C++ is the smallest chunk of the program code that has a meaning to the compiler. They are of 6 types:

Token	Description/Purpose	Example
Keywords	Special reserved words that the compiler recognizes	`int`, `double`, `char`, `for`,`auto`
Identifiers	Names of things that aren't hard-coded into the language via the compiler	`cout`, `printf`, `std`, `x`, `myFunction`
Literals	Constants whose values are specified directly in the source code	`"Hello World"`, `24.3`, `0`, `'c'`
Operators	Mathematical or logical operations	+, -, *, /, ++, &&, %, <<
Punctuation/ Separators	Punctuation defining the structure of the source code	{ } ( ) , ;
Whitespace	Spaces of various sorts; ignored by compiler	Spaces, tabs, newlines, comments

C Hello World, Keywords

```
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv) {
   printf("Hello World\n");
   return(0);
}
```

Keywords, also known as reserved words, are inherently known to the compiler. They can't be used as variable or function names. The most common keywords are the known C/C++ data types as shown here.

Types of Tokens:

Keywords	`int`, `double`, `char`, `for`
Identifiers	`cout`,`std`, `x`,`myFunction`
Literals	`"Hello World"`, `24.3`, `0`, `'c'`
Operators	+, −, ∗, /, ++, &&, %
Punctuation/Separators	{ } ( ) , ;
White Space	Spaces, tabs, newlines, comments

C Hello World, Data types

```
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv) {
   printf("Hello World\n");
   return(0);
}
```

Keywords, also known as reserved words, are inherently known to the compiler. They can't be used as variable or function names. The most common keywords are the known C/C++ data types as shown here.

C/C++ is a typed language: all variables need to be **declared** to be of some known type. Common C/C++ data types:

Type	Memory size	Range
char	1 byte	0 to 255
int	4 bytes	-2147483648 to $+2147483647$
short int	2 bytes	-32768 to 32767
long int	8 bytes	-2^{63} to $2^{63}-1$
float	4 bytes	
double	8 bytes	
long double	16 bytes	

C Hello World, Data types

```
#include<stdio.h>

/* Check the size of data types in C */
int main(int argc, char **argv) {
   printf ("sizeof(char) = %d\n",sizeof(char));
   printf ("sizeof(int) = %d\n",sizeof(int));
   printf ("sizeof(long int) = %d\n",sizeof(long int));
   printf ("sizeof(short int) = %d\n",sizeof(short int));
   short int x=32767, y=x+1;
   printf ("x = %d, y = %d\n",x,y);
   return(0);
}
```

C Hello World, Data types

```c
#include<stdio.h>

/* Check the size of data types in C */
int main(int argc, char **argv) {
   printf ("sizeof(char) = %d\n",sizeof(char));
   printf ("sizeof(int) = %d\n",sizeof(int));
   printf ("sizeof(long int) = %d\n",sizeof(long int));
   printf ("sizeof(short int) = %d\n",sizeof(short int));
   short int x=32767, y=x+1;
   printf ("x = %d, y = %d\n",x,y);
   return(0);
}
```

Output:

```
   sizeof(char) = 1
   sizeof(int) = 4
   sizeof(long int) = 8
   sizeof(short int) = 2
   x = 32767, y = -32768
```

Quick refresher on base N

16s ones

Base 16: | F | B |

$15 \times 16 + 11 \times 1 = 251$

100s tens ones

Base 10: | 2 | 5 | 1 |

$2 \times 100 + 5 \times 10 + 1 \times 1 = 251$

64s 8's ones

Base 8: | 3 | 7 | 3 |

$3 \times 64 + 7 \times 8 + 3 \times 1 = 251$

128 64 32 16 8 4 2 1

Base 2: | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

$128 + 64 + 32 + 16 + 8 + 2 + 1 = 251$

C Hello World, Data types, Double vs Float

Double precision is 64 bits while float is single precision consuming 32 bits.

```
#include<stdio.h>

/* Check the accuracy of float and double */
int main(int argc, char **argv) {
   float x=3.14159265358979323846;
   double y=3.14159265358979323846;
   printf ("x = %.20f \n",x);
   printf ("y = %.20lf\n",y);
   return(0);
}
```

C Hello World, Data types, Double vs Float

Double precision is 64 bits while float is single precision consuming 32 bits.

```
#include<stdio.h>

/* Check the accuracy of float and double */
int main(int argc, char **argv) {
   float x=3.14159265358979323846;
   double y=3.14159265358979323846;
   printf ("x = %.20f \n",x);
   printf ("y = %.20lf\n",y);
   return(0);
}
```

Output:

```
x = 3.14159274101257324219
y = 3.14159265358979311600
```

Data types, Declarations

Before using a variable `x`, you must tell C/C++ what `x` is by declaring:

```
int x; // int is the keyword, x is the identifier
```

- After declaration, compiler knows what is `x` (it's identified).
- Compiler grabs 4 bytes from the stack and assigns it to `x`.
- Memory freed when variable goes out of scope (function ends)

Functions also need declarations:

```
int mymax(int x, int y);
```

- Declares function `mymax` as returning an `int`
- `int x` and `int y` are inputs to `mymax`
- After function declaration, compilers knows what is mymax:
  - A function that takes two ints as input and returns an int as output
- All variables/functions need to be declared before first use
- Standard practice is to put function declarations into a **header file**
- Others can `#include` the header file to use the functions there.

Function definitions

- **Function declarations** declare how the compiler is to call functions
- **Function definitions** define what the function does with its inputs and how it computes its outputs when it enters the function.
- Function definitions look similiar to function declarations at first

```
int mymax(int x, int y) {
   int z;
   if (x>y) z=x;
   else     z=y;
   return(z);
}
```

- This functions takes two integers as inputs, computes their maximum and returns it to the calling environment
- Function declarations **must** occur before the first use of the function
- Function definitions can be stored anywhere: in the same file, in another file, or in another library (tell compiler which file or library)
- Declaration must match definition, or compilation will fail.

Function Declarations and Definitions

```
#include<stdio.h>
#include<math.h>

int main(int argc, char **argv) {
    double x = cos(5);
    return(0);
}
```

```
$ gcc -o hello hello.c -lm
```

- Function declaration of `cos` is stored in `math.h`
- Declaration occurs before first use of the function
  → Compiler knows how to call `cos`
- During compilation phase the compiler puts a stub in the object file for the function `cos` based on the known function declaration.
- During linking phase, the compiler gets the function definition. In this case, it is in the math library linked with `-lm`.

Qualifiers

In C/C++ **qualifiers** modify a property of the data type being declared.

1. **Sign qualifiers**
   - `signed`, `unsigned`. For example:
     `signed int` can take a value from -2^{31} to $+2^{31}-1$
     `unsigned int` can take a value from 0 to $+2^{32}-1$
2. **Size qualifiers**
   - `short`, `long`. For example:
     `short int` could be 2 bytes
     `int` could be 4 bytes
     `long int` could be 8 bytes
3. **Type qualifiers**
   - `const`: the declared type cannot change within the scope of its declaration.
   - `volatile`: the declared type could change outside the program's control within the scope of its declaration.

C Hello World, Identifiers

```c
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv) {
   printf("Hello World\n");
   return(0);
}
```

You can think of the identifiers as the variables and functions in the source code. These are not known a-priori to the compiler, but the compiler knows how to handle them as they are "declared" before they are used.

Types of Tokens:

Keywords	`int`, `double`, `char`, `for`
Identifiers	`cout`,`std`, `x`,`myFunction`
Literals	`"Hello World"`, 24.3, 0, ’c’
Operators	+, −, ∗, /, ++, &&, %
Punctuation/Separators	{ } ( ) , ;
White Space	Spaces, tabs, newlines, comments

C Hello World, Identifiers

```
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv) {
  printf("Hello World\n");
  return(0);
}
```

You can think of the identifiers as the variables and functions in the source code. These are not known a-priori to the compiler, but the compiler knows how to handle them as they are "declared" before they are used.

Rules for identifiers:

- Must consist of upper and lower case alpha-numeric characters and the underscore _ symbol (they are case-sensitive)
- Identifiers must be unique
- The first character must be an alphabet or underscore
- Keywords are not valid as identifiers
- Only the first 31 characters are used.

C Hello World, Literals

```
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv
   printf("Hello World\n");
   return(0);
}
```

Literals are the values assigned to variables typed in literally in the source code. For literal strings, there are several non-printable characters that have special escape sequences. Meanwhile returning 0 to the calling environment signals a successful completion to the caller.

Types of Tokens:

Keywords	int, double, char, for
Identifiers	cout,std, x,myFunction
Literals	"Hello World", 24.3, 0, 'c'
Operators	+, -, *, /, ++, &&, %
Punctuation/Separators	{ } ( ) , ;
White Space	Spaces, tabs, newlines, comments

C Hello World, Literals

```c
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv
  printf("Hello World\n");
  return(0);
}
```

Literals are the values assigned to variables typed in literally in the source code. For literal strings, there are several non-printable characters that have special escape sequences. Meanwhile returning 0 to the calling environment signals a successful completion to the caller.

Escape sequences:

Escape Seq	Represented Char	Escape Seq	Represented Char
\a	System Bell	\b	Backspace
\f	Formfeed (pagebreak)	\n	Newline (line break)
\r	carriage return	\t	Tab
\\	A single backslash	\'	Single quote char
\“	Double quote char	%%	% percent character

University of Nottingham UK | CHINA | MALAYSIA

`printf()` formatting

The `printf()` function sends bytes to the output stream defined by its arguments. There are 2 types of arguments to `printf()`:

`printf(`*`formatString`*`, `*`variables`*`...);`

- *`formatString`* a string determining how the variables are printed
- *`variables`* is a list of declared variables to be printed according to the format string.

Conversion character	Corresponding argument is printed	Conversion character	Corresponding argument is printed
c	as single character	g	e or f format
d,i	as decimal integer		whichever shorter
u	as unsigned integer	s	as a string
o	as octal number	p	as a pointer address
x,X	as hexadecimal number	f	floating point format
e	exponential format		

Examples: printf()

```
#include<stdio.h>
int main(int argc, char **argv) {
   int x=-23;
   /* Print x in different formats */
   printf("x=%d, x=%c x=%u, x=%f, x=%e, x=%g\n", x,x,x,x,x,x);
   x=65;
   printf("x=%d, x=%c x=%u, x=%f, x=%e, x=%g\n", x,x,x,x,x,x);
   float y=3.1415;
   printf("y=%d, y=%c y=%u, y=%f, y=%e, y=%g\n", y,y,y,y,y,y);
   return(0);
}
```

Output:

```
x=-23, x=\351 x=4294967273, x=0.000000, x=0.000000e+00, x=0
x=65, x=A x=65, x=0.000000, x=0.000000e+00, x=0
y=1768764064, y=@ y=0, y=3.141500, y=3.141500e+00, y=3.1415
```

printf() formatting

The printf() function formatting string gives more control over the way the number is printed.

- A number before the conversion character indicates how much space the rendered text will take up:

```
printf("'%8d'",123);
```
→
```
'     123'
```

- To zero-pad an integer put a zero before the number:

```
printf("'%08d'",123);
```
→
```
'00000123'
```

- To control number of decimal places in a float use:

```
printf("'%8.3f'",20.0/3.0);
```
→
```
'   6.667'
```

- To left align instead of right align, use a negative number:

```
printf("'%-8.3f'",20.0/3.0);
```
→
```
'6.667   '
```

- Works for strings too:

```
printf("'%-8s'","abc");
```
→
```
'abc     '
```

Strings in C

- A string in C is just an array of characters, terminated by the end-of-string character '\0'.

```
char s[]="Hello";
```

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]
H	e	l	l	o	\0

6 bytes alloc for 5 chars

- The end-of-string character tells functions like `strcpy` `strcmp` and `printf` where the string ends.
- A string is a **pointer** to an **array of chars**, denoted by `*`.
- Pointer-to-array-of-chars is just the memory address of the array.
- Some C string commands:

```
char *strcat(char *s1, const char *s2);//Concatenate s2 onto s1.
char *strcpy(char *s1, const char *s2);//Copy s2 into s1.
int strcmp(const char *s1, const char *s2);//Compare s2 with s1.
unsigned int strlen(const char *s);//Length of s
```

Note: You must `#include<string.h>` to use the above string functions.

Strings in C Example:

```
#include <stdio.h>
#include <string.h>
int main () {
    char str1[12] = "Hello";// End-of-string char is automatically included
    char str2[12] = "World";
    char str3[12];
    int len ;
    /* copy str1 into str3 */
    strcpy(str3, str1);
    printf("strcpy( str3, str1) : %s\n", str3 );
    /* concatenates str3 and str2 */
    strcat( str3, str2);
    printf("strcat( str3, str2): %s\n", str3 );
    /* total lenghth of str3 after concatenation */
    len = strlen(str3);
    printf("strlen(str3) : %d\n", len );
    return 0;
}
```

Strings in C Example:

```
#include <stdio.h>
#include <string.h>
int main () {
    char str1[12] = "Hello";// End-of-string char is automatically included
    char str2[12] = "World";
    char str3[12];
    int len ;
    /* copy str1 into str3 */
    strcpy(str3, str1);
    printf("strcpy( str3, str1) : %s\n", str3 );
    /* concatenates str3 and str2 */
    strcat( str3, str2);
    printf("strcat( str3, str2): %s\n", str3 );
    /* total lenghth of str3 after concatenation */
    len = strlen(str3);
    printf("strlen(str3) : %d\n", len );
    return 0;
}
```

Output:

```
strcpy( str3, str1) : Hello
strcat( str3, str2): HelloWorld
strlen(str3) : 10
```

C Hello World, Operators

```
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv
   printf("Hello World\n");
   return(0);
}
```

Operators operate on the variables.
There are **unary** operators that operate on 1 variable, such as `x++` and `x--`.
There are **binary** operators that operate on 2 variables such as `x+y` and `x/y`.
There is 1 **ternary** operator that operates on 3 variables (`x?y:z`).
Operators can both modify the value of the variable and also "return" a value to the caller.

Types of Tokens:

Keywords	`int`, `double`, `char`, `for`
Identifiers	`cout`,`std`, `x`,`myFunction`
Literals	`"Hello World"`, `24.3`, `0`, `'c'`
Operators	+, −, ∗, /, ++, &&, %
Punctuation/Separators	{ } ( ) , ;
White Space	Spaces, tabs, newlines, comments

Assignment, Increment, Math Operators

Operator	Name	Function	Example	Returns
$=$	Assignment	Assigns to var	$x = 12;$	The value assigned.
$++$	Increment	Increments var	$x++;$	x
$--$	Decrement	Decrements var	$x--;$	x
Alternate form of increment/decrement operators				
$++$	Increment	Increments var	$++x;$	$x+1$
$--$	Decrement	Decrements var	$--x;$	$x-1$
$+=$	Increment	Increments var	$x+=5;$	The final value
$-=$	Decrement	Decrements var	$x-=3;$	The final value
$*=$		Multiplies var	$x*=3;$	The final value
$/=$		Divides var	$x/=2;$	The final value
$+$	Addition	Adds 2 vars	$x+y;$	$x+y$
$-$	Subtraction	Subtracts 2 vars	$x-y;$	$x-y$
$*$	Multiplication	Multiplies 2 vars	$x*y;$	$x*y$
$/$	Division	Divides 2 vars	$x/y;$	x/y
$\%$	Remainder	Remainder modulo y	$x\%y;$	$x\%y$

Examples Operators

```
#include<stdio.h>
/* Testing of Assignment Increment and Math operators*/
int main(int argc, char **argv) {
   int x=0,y=0,z=0;/* Declare x, y and z. Initialize values to 0 */
   /* Each assignment operator returns the value it assigned */
   x=y=z=12;
   /* Difference between pre and post increment operators. */
   printf("(x++)=%d. (++y)=%d\n",(x++),(++y));
   /* Test binary mathematical operator. */
   printf("x=%d, y=%d. Binary operator x+y=%d\n",x,y,x+y);
   //* Test *= and /= operators. */
   printf("(x*=2)=%d, (y/=2)=%d\n",(x*=2),(y/=2));
   printf("x+y=%d, x%%y=%d\n",(x+y),(x%y));
   return(0);
}
```

Examples Operators

```
#include<stdio.h>
/* Testing of Assignment Increment and Math operators*/
int main(int argc, char **argv) {
   int x=0,y=0,z=0;/* Declare x, y and z. Initialize values to 0 */
   /* Each assignment operator returns the value it assigned */
   x=y=z=12;
   /* Difference between pre and post increment operators. */
   printf("(x++)=%d. (++y)=%d\n",(x++),(++y));
   /* Test binary mathematical operator. */
   printf("x=%d, y=%d. Binary operator x+y=%d\n",x,y,x+y);
   //* Test *= and /= operators. */
   printf("(x*=2)=%d, (y/=2)=%d\n",(x*=2),(y/=2));
   printf("x+y=%d, x%%y=%d\n",(x+y),(x%y));
   return(0);
}
```

```
(x++)=12. (++y)=13
x=13,y=13. Binary operator x+y=26
(x*=2)=26,(y/=2)=6
x+y=32, x%y=2
```

Relational, Logical and Bitwise Operators

Operator	Name	Function	Example	Returns
$==$	Comparison	True if x=y	$x == y$	
$!=$	Not equal	True if x≠y	$x! = y$	1 if True.
$<, <=$	Less than/equal	True if $x < y$	$x < y$	0 if False.
$>, >=$	Greater than/equal	True if $x \geq y$	$x >= y$	
$\&\&$	Logical AND		$x\&\&y$	
$\vert\vert$	Logical OR		$x\vert\vert y$	
!	Logical NOT		$!x$	
$\&$	Bitwise AND		$x\&y$	
$\vert$	Bitwise OR		$x\vert y$	
^	Bitwise XOR		x ^ y	
$<<$	Bitwise left shift operator		$x << y$	
$>>$	Bitwise right shift operator		$x >> y$	
~	Bitwise Not operator		~x	

Example: Relational, Logical, Bitwise Operators

```c
#include<stdio.h>
/* Testing of Relational, Logical and Bitwise Operators */
int main(int argc, char **argv) {
   int x=2,y=0;/* Declare x and y and initialize their values */
   /* Relational operators */
   printf("(x==x)=%d, (x==y)=%d\n",(x==x),(x==y));
   /* Logical operators */
   printf("(x&&y)=%d, (x||y) =%d\n", (x&&y),(x||y));
   /* Bitwise operators */
   printf("(x&y)=%d, (x|y)=%d, (x<<2)=%d, (x>>1)=%d\n",
          (x&y),(x|y),(x<<2),(x>>1));
   return(0);
}
```

Example: Relational, Logical, Bitwise Operators

```c
#include<stdio.h>
/* Testing of Relational, Logical and Bitwise Operators */
int main(int argc, char **argv) {
   int x=2,y=0;/* Declare x and y and initialize their values */
   /* Relational operators */
   printf("(x==x)=%d, (x==y)=%d\n",(x==x),(x==y));
   /* Logical operators */
   printf("(x&&y)=%d, (x||y) =%d\n", (x&&y),(x||y));
   /* Bitwise operators */
   printf("(x&y)=%d, (x|y)=%d, (x<<2)=%d, (x>>1)=%d\n",
          (x&y),(x|y),(x<<2),(x>>1));
   return(0);
}
```

Output:

```
(x==x)=1, (x==y) =0
(x&&y)=0, (x||y) =1
(x&y)=0,(x|y)=2, (x<<2)=8, (x>>1)=1
```

Example: Relational, Logical, Bitwise Operators

```
#include<stdio.h>
/* Testing of Relational, Logical and Bitwise Operators */
int main(int argc, char **argv) {
   int x=2,y=0;/* Declare x and y and initialize their values */
   /* Relational operators */
   printf("(x==x)=%d, (x==y)=%d\n",(x==x),(x==y));
   /* Logical operators */
   printf("(x&&y)=%d, (x||y) =%d\n", (x&&y),(x||y));
   /* Bitwise operators */
   printf("(x&y)=%d, (x|y)=%d, (x<<2)=%d, (x>>1)=%d\n",
          (x&y),(x|y),(x<<2),(x>>1));
   return(0);
}
```

Output:

```
(x==x)=1, (x==y) =0
(x&&y)=0, (x||y) =1
(x&y)=0, (x|y)=2, (x<<2)=8, (x>>1)=1
```

(x&y):bitwise ANDs x and y:

0	0	0	0	0	0	1	0	=2
0	0	0	0	0	0	0	0	=0
0	0	0	0	0	0	0	0	=0

Example: Relational, Logical, Bitwise Operators

```c
#include<stdio.h>
/* Testing of Relational, Logical and Bitwise Operators */
int main(int argc, char **argv) {
   int x=2,y=0;/* Declare x and y and initialize their values */
   /* Relational operators */
   printf("(x==x)=%d, (x==y)=%d\n",(x==x),(x==y));
   /* Logical operators */
   printf("(x&&y)=%d, (x||y) =%d\n", (x&&y),(x||y));
   /* Bitwise operators */
   printf("(x&y)=%d, (x|y)=%d, (x<<2)=%d, (x>>1)=%d\n",
          (x&y),(x|y),(x<<2),(x>>1));
   return(0);
}
```

Output:

```
(x==x)=1, (x==y) =0
(x&&y)=0, (x||y) =1
(x&y)=0, (x|y)=2, (x<<2)=8, (x>>1)=1
```

(x|y):bitwise ORs x and y:

0	0	0	0	0	0	1	0	=2
0	0	0	0	0	0	0	0	=0
0	0	0	0	0	0	1	0	=2

Example: Relational, Logical, Bitwise Operators

```
#include<stdio.h>
/* Testing of Relational, Logical and Bitwise Operators */
int main(int argc, char **argv) {
   int x=2,y=0;/* Declare x and y and initialize their values */
   /* Relational operators */
   printf("(x==x)=%d, (x==y)=%d\n",(x==x),(x==y));
   /* Logical operators */
   printf("(x&&y)=%d, (x||y) =%d\n", (x&&y),(x||y));
   /* Bitwise operators */
   printf("(x&y)=%d, (x|y)=%d, (x<<2)=%d, (x>>1)=%d\n",
          (x&y),(x|y),(x<<2),(x>>1));
   return(0);
}
```

(x<<2):bitwise shift left x:

0	0	0	0	0	0	1	0	=2
0	0	0	0	1	0	0	0	=8

Output:

```
(x==x)=1, (x==y) =0
(x&&y)=0, (x||y) =1
(x&y)=0,(x|y)=2, (x<<2)=8, (x>>1)=1
```

Example: Relational, Logical, Bitwise Operators

```c
#include<stdio.h>
/* Testing of Relational, Logical and Bitwise Operators */
int main(int argc, char **argv) {
   int x=2,y=0;/* Declare x and y and initialize their values */
   /* Relational operators */
   printf("(x==x)=%d, (x==y)=%d\n",(x==x),(x==y));
   /* Logical operators */
   printf("(x&&y)=%d, (x||y) =%d\n", (x&&y),(x||y));
   /* Bitwise operators */
   printf("(x&y)=%d, (x|y)=%d, (x<<2)=%d, (x>>1)=%d\n",
          (x&y),(x|y),(x<<2),(x>>1));
   return(0);
}
```

Output:

```
(x==x)=1, (x==y) =0
(x&&y)=0, (x||y) =1
(x&y)=0,(x|y)=2, (x<<2)=8, (x>>1)=1
```

(x>>1):bitwise shift right x:

0	0	0	0	0	0	1	0	=2
0	0	0	0	0	0	0	1	=1

Example: The ternary operator (:?)

The ternary operator has the following form
(*condition* ? *value1* : *value2*)

The ternary operator returns *value1* if *condition* is true
and returns *value2* if *condition* is false

```
#include<stdio.h>
// Testing of ternary operator
int main(int argc, char **argv) {
   int x=3,y=4;
   // Ternary operator replicates the max function
   printf("(x>y?y:x)=%d\n",(x>y?x:y));
   return(0);
}
```

Example: The ternary operator (:?)

The ternary operator has the following form
(*condition* ? *value1* : *value2*)

The ternary operator returns *value1* if *condition* is true
and returns *value2* if *condition* is false

```
#include<stdio.h>
// Testing of ternary operator
int main(int argc, char **argv) {
   int x=3,y=4;
   // Ternary operator replicates the max function
   printf("(x>y?y:x)=%d\n",(x>y?x:y));
   return(0);
}
```

Output:

```
(x>y?x:y) =4
```

C Hello World, Separators

```
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv
   printf("Hello World\n");
   return(0);
}
```

Separators define the structure of the C/C++ program. The semicolon ; indicates to the compiler the end of a command-line which can span several physical lines in the source code file. Commas , delineates different inputs in a function call or declarations when declaring variables. Parentheses () and braces {} delineate the end of inputs and functional and control blocks.

Types of Tokens:

Keywords	`int`, `double`, `char`, `for`
Identifiers	`cout`,`std`, `x`,`myFunction`
Literals	`"Hello World"`, `24.3`, `0`, `'c'`
Operators	+, −, ∗, /, ++, &&, %
Punctuation/Separators	{ } ( ) , ;
White Space	Spaces, tabs, newlines, comments

C Hello World, Separators

```
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv
   printf("Hello World\n");
   return(0);
}
```

Separators define the structure of the C/C++ program. The semicolon ; indicates to the compiler the end of a command-line which can span several physical lines in the source code file. Commas , delineates different inputs in a function call or declarations when declaring variables. Parentheses () and braces {} delineate the end of inputs and functional and control blocks.

C Hello World, white-space

```
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv) {
   printf("Hello World\n");
   return(0);
}
```

In general, the compiler ignores white-space and comments. These are put there primarily for the humans rather than for the compiler.

Types of Tokens:

Keywords	int, double, char, for
Identifiers	cout,std, x,myFunction
Literals	"Hello World", 24.3, 0, 'c'
Operators	+, −, ∗, /, ++, &&, %
Punctuation/Separators	{ } ( ) , ;
White Space	Spaces, tabs, newlines, comments

C Hello World, white-space

```
#include<stdio.h>

/* Hello world program in C */
int main(int argc, char **argv) {
   printf("Hello World\n");
   return(0);
}
```

In general, the compiler ignores white-space and comments. These are put there primarily for the humans rather than for the compiler.

The above program could have been written as below, and the compiler wouldn't care, but it'll become buggy and unmaintainable for humans.

```
#include<stdio.h>
int main(int argc, char **argv) {printf("Hello world!\n");return(0);
}
```

Please make proper use of comments and whitespace. You will be penalized if you write the second code above.

C Hello World, comments

The standard C-commenting structure uses /* to begin a comment and */ to end it. Comments in the C-comment structure can span multiple lines. It is typical and helpful to code maintenance to have comments such as:

```
/****************************************
 * It is very useful to to write multiline comments
 * that describe what an entire section of code does
 * using the C commenting structure.  This way you
 * immediately know what the code does and which part
 * of the code you are looking at.
 ****************************************/
```

C++ uses a // comment structure which extends to the end of the line

```
// swap x and y
tmp = x; // assign x to a temporary variable
x = y; // copy y into x variable
y = tmp; // copy tmp variable into y.
```

Good programming practices: comments + whitespa

Make sure your code is always well commented and visually well structured.

- When the code is trivial (eg: hello world) commenting your code may not seem important.
- Don't fall into the trap of thinking that commenting code is not important in general: it is critical to debugging, maintaining and reusing non-trivial code.
- Every function needs to be documented with comments stating:
  - What the function does
  - What is each input to the function
  - What is each output the function modifies or returns.
- If you ever find your self repeating code with cut-n-paste, put that code into its own function with appropriate inputs and outputs and call the function instead.
- Functions → greater modularity → more maintainable code.