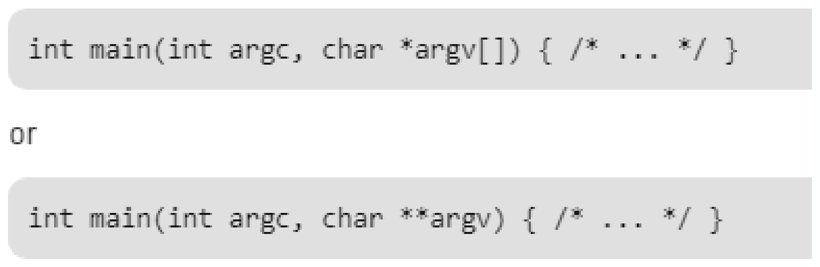
Generic Programming in C

argc, *argv

Command line arguments: argv, argc

- The main() function is mostly defined with a return type of int and without parameters.
- We can also give command-line arguments.
- Command-line arguments are given after the name of the program (e.g. linux command line).
- To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments (argc) and second is list of command-line arguments (argv).



```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

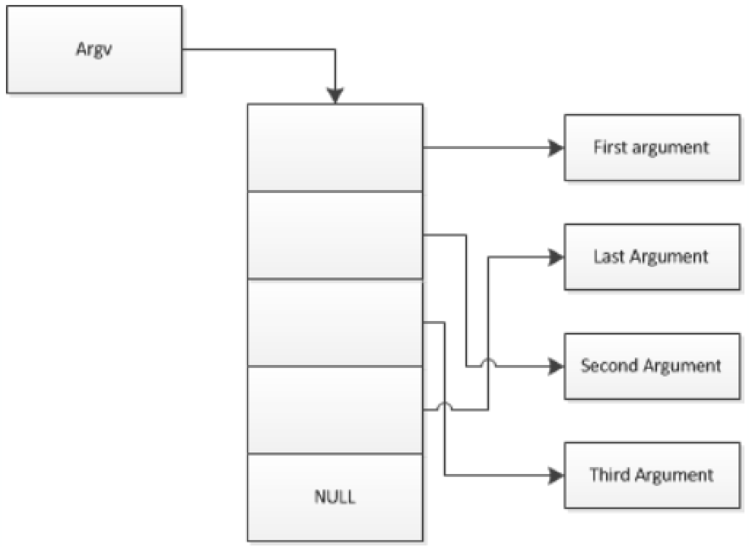
Command line arguments: argv, argc

Properties of Command Line Arguments:

- It is possible to pass some values from the command line to your C programs when they are executed.
- These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.
- The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program.
- argv[LAST] is a NULL pointer.
- argv[0] holds the name of the program.
- argv[1] points to the first command line argument and argv[n] points last argument.

argv

- Argv contains the address of an array of pointers.
- Each of those points to some place in memory containing the text of the corresponding argument that was passed on the command line.
- Then, at the end of that array there's guaranteed to be a null pointer.
- Note that the actual storage for the individual arguments are at least potentially allocated separately from each other, so their addresses in memory might be arranged fairly randomly (but depending on how things happen to be written, they could also be in a single contiguous block of memory--you simply don't know and shouldn't care).



Command line arguments: argv, argc

- To pass command line arguments to our program we should use the following **main** declaration:

```
main(int argc, char* argv[]) { ...
```

✓char** argv

✗char argv[][]

argv & argc: example

```
$ prog1 –u danny –p 1234
```

argc == 5

argv[0] == “prog1”

argv[1] == “-u”

...

argv[4] == “1234”

Always: argv[argc] == NULL

(redundant since we are also given argc)

Command line arguments - example

```
#include <stdio.h>

int main( int argc, char *argv[] )  {

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

Code Generalizability

Code Generalizability

- Can we write code once – code that works on a variety of types / sizes?
- Solutions
  - Pointers – not limited to a specific size
  - void* - not limited to a specific type
  - Templates (“pure” Generic programming)

Void* (Pointers)

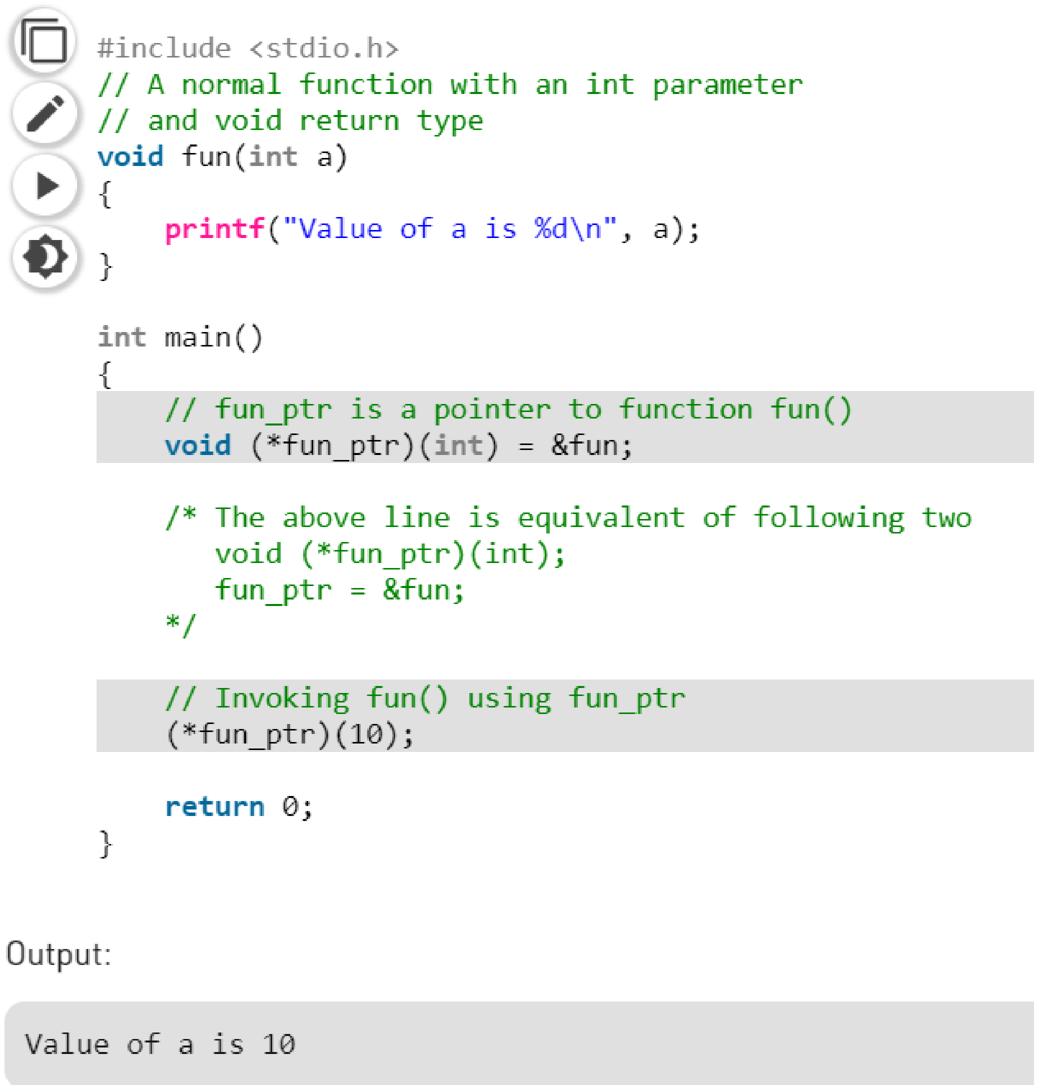
- A way to pass data of an arbitrary type.
- void* p - is a generic pointer capable of representing any pointer type.
- void* pointer cannot be dereferenced.
- "Dereferencing a pointer" means using the pointer to access the object that it points to.
- Even though void pointer points to memory allocated to hold an int, the pointer can't be used directly to access that memory. It must be converted to a pointer to int first.

```
int i= 5;
double f= 3.14;

void* p;
p= &i; // ok
p= &f; // ok
printf("%f\n",*p)); // error
printf("%f\n",*((double*)p)); // ok
```

Pointers to Functions

- Assuming the function f is defined, &f and f are pointers to the function - the address where the function's definition begins in memory.
- Most common-use – for callback function (any exe code that is passed as an argument to other code; that other code is expected to *call back* (execute) the argument at a given time)
- A pointer to function will be declared only (e.g. function prototype), without function definition of the pointer (only definition for the function itself)

Pointers to Functions - example

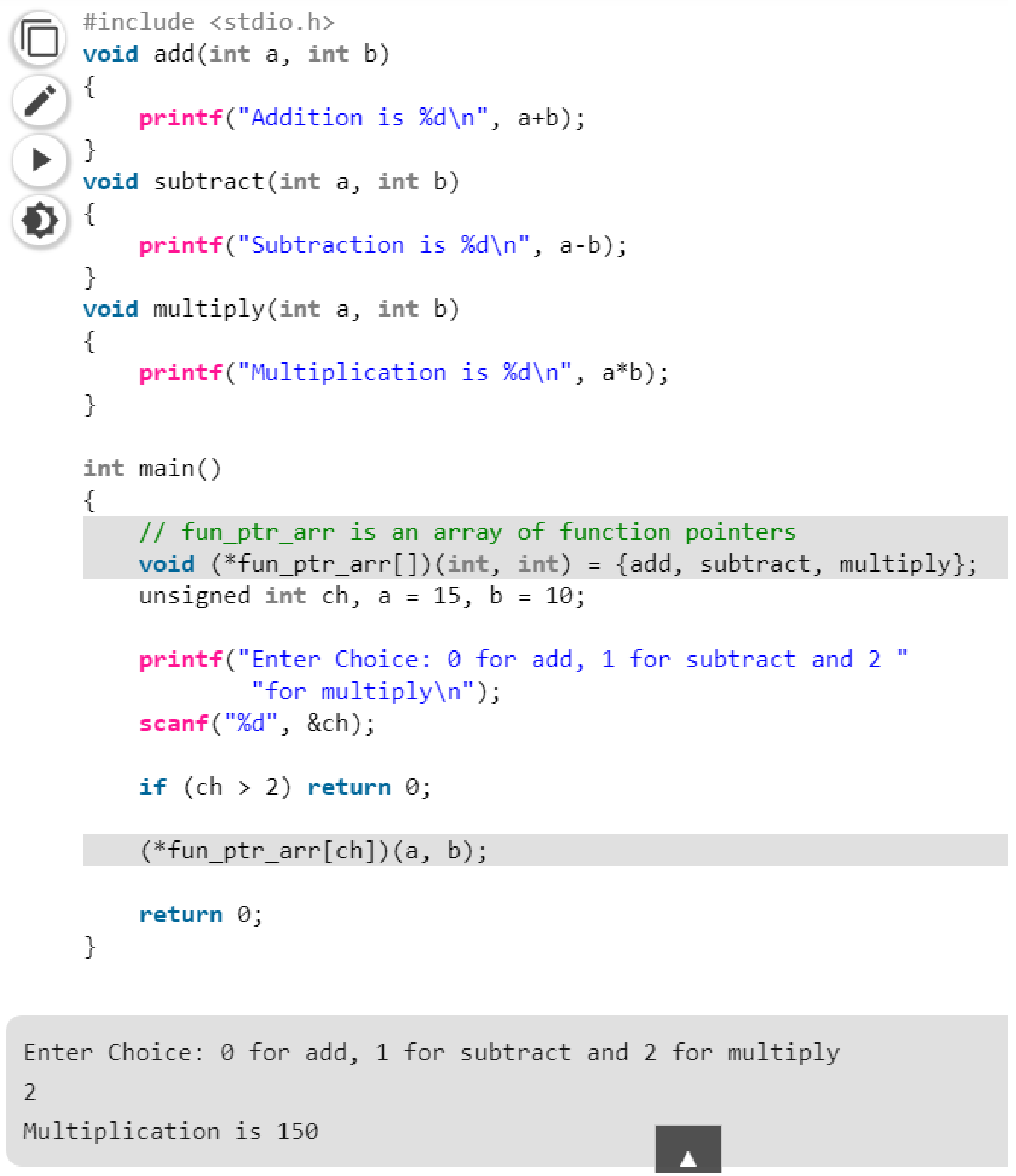


```c
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
       void (*fun_ptr)(int);
       fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

Output:

```
Value of a is 10
```

Pointers to Functions - example



```
#include <stdio.h>
void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 "
            "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```

```
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Multiplication is 150
```

Functions typedef

- Generic programming - pass a function name to another function as a parameter.
- Without the typedef word, the declaration would declare a variable TwoIntsFunc of type pointer to function of 2 input arguments, returning void.
- With the typedef it instead defines TwoIntsFunc as a name for that type.

```
typedef int (* TwoIntsFunc) (int, int);
TwoIntsFunc f1;
f1 = &avg;
f1 = &sum;
```

qsort_e.c (code example in moodle)

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

Array to be sorted

# elements in array

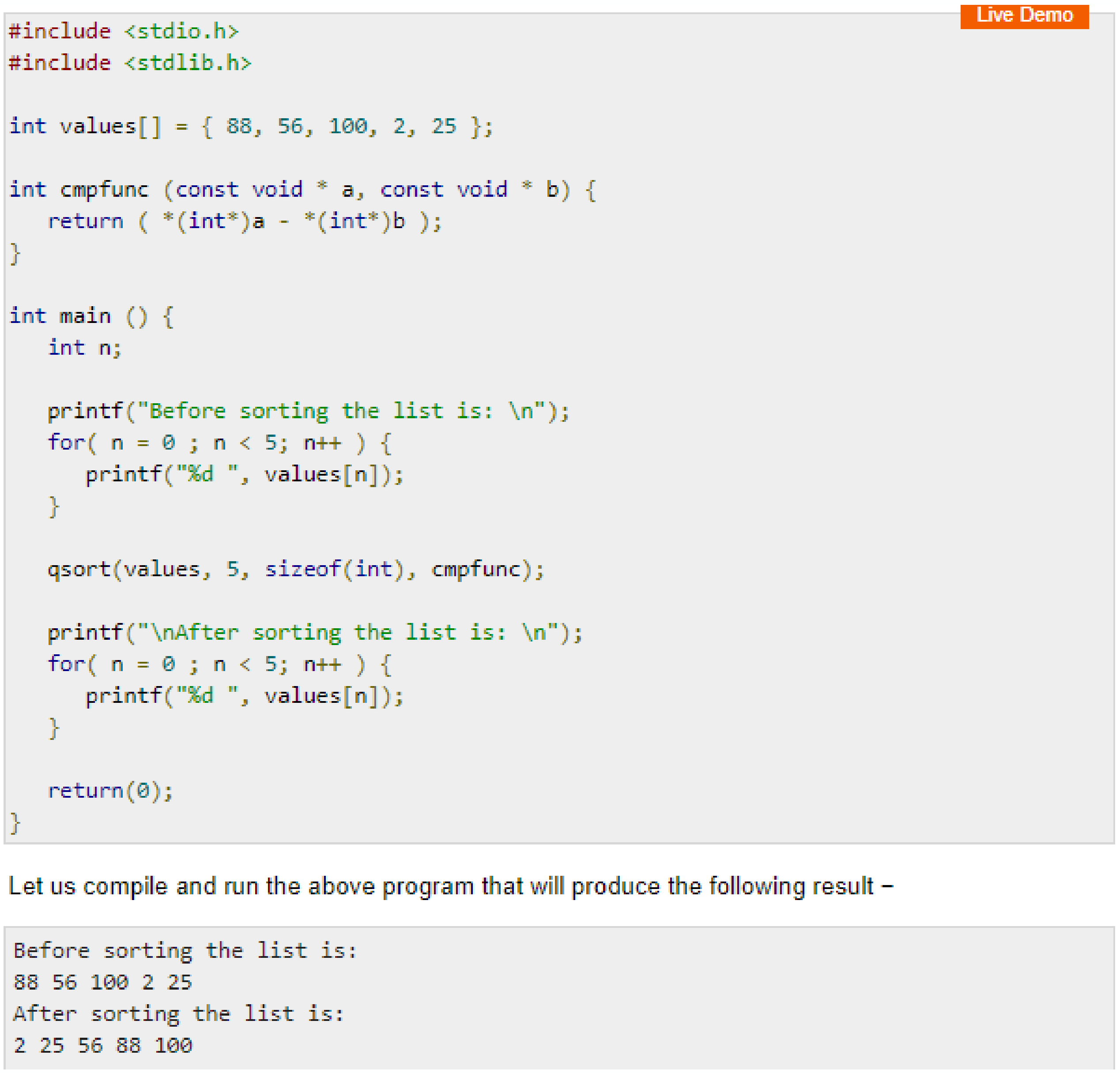
sizeof each element in array

Pointer to the comparator function.

Return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

- qsort problems -
    - Not efficient – casting, calls to functions.
    - Not user friendly.
    - Type safety problems.

qsort example



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

int values[] = { 88, 56, 100, 2, 25 };

int cmpfunc (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}

int main () {
   int n;

   printf("Before sorting the list is: \n");
   for( n = 0 ; n < 5; n++ ) {
      printf("%d ", values[n]);
   }

   qsort(values, 5, sizeof(int), cmpfunc);

   printf("\nAfter sorting the list is: \n");
   for( n = 0 ; n < 5; n++ ) {
      printf("%d ", values[n]);
   }

   return(0);
}
```

Let us compile and run the above program that will produce the following result –

```
Before sorting the list is:
88 56 100 2 25
After sorting the list is:
2 25 56 88 100
```

Array optimization, register, volatile, union

Optimization

- Code optimization is any method of code modification to improve code quality and efficiency.
- A program may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer input/output operations.
- Check if you need to optimize
- Remember to "turn off" automatic debugging (#define NDEBUG) to prevent premature optimization
- Check where to optimize
- Use common techniques such as cache,...

Memory Levels

- Level 1 or Register (CPU) –

  It is a type of memory in which data is stored and accepted that are immediately stored in CPU.

- Level 2 or Cache memory –

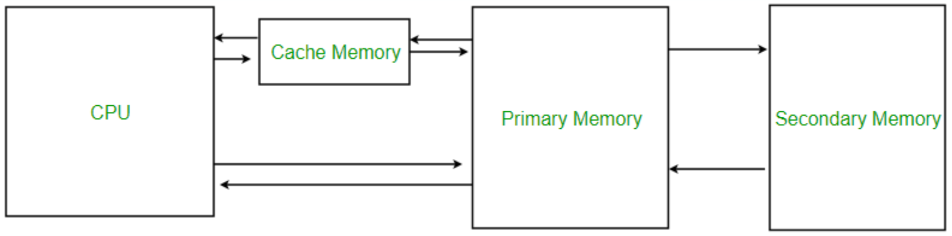
  It is the fastest memory which has faster access time where data is temporarily stored for faster access.

- Level 3 or Main Memory –

  It is memory on which computer works currently. It is small in size and once power is off data no longer stays in this memory.

- Level 4 or Secondary Memory (hard-disk) –

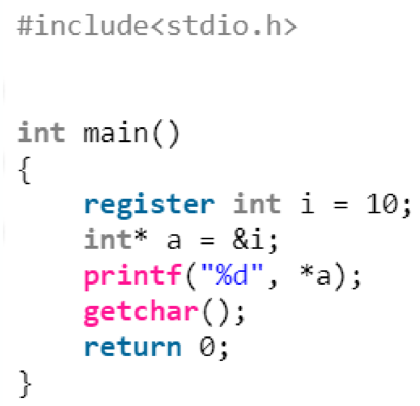
  It is external memory which is not as fast as main memory but data stays permanently in this memory.



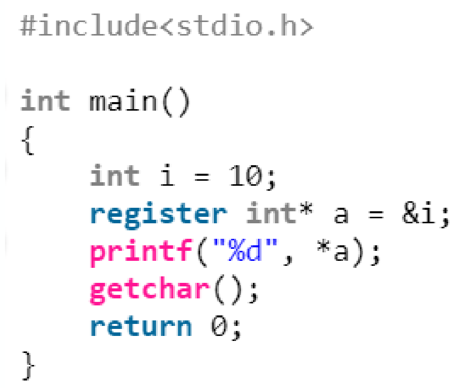
Register

- A small amount of very fast memory.
- Registers are faster than the main memory to access, so the variables which are most frequently used in a C program can be put in registers using register keyword.
- Provides quick access to commonly used values.
- The register keyword specifies that the variable is to be stored in a machine register, if possible.
- Experience has shown that the compiler usually knows much better than humans what should go into registers and when.
- Generally, compilers themselves do optimizations and put the variables in register.

Register - examples



```c
#include<stdio.h>

int main()
{
    register int i = 10;
    int* a = &i;
    printf("%d", *a);
    getchar();
    return 0;
}
```



```c
#include<stdio.h>

int main()
{
    int i = 10;
    register int* a = &i;
    printf("%d", *a);
    getchar();
    return 0;
}
```

Cache Memory

- Very high-speed memory.
- It is used to speed up and synchronizing with high-speed CPU.
- Cache memory is more expensive than main memory or disk memory but economical than CPU registers.
- Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU.
- It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.
- Cache memory is used to reduce the average time to access data from the Main memory.
- The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations.
- It is done automatically by the compiler. No special manual keyword like register

Cache Memory

- Application

  The cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory.

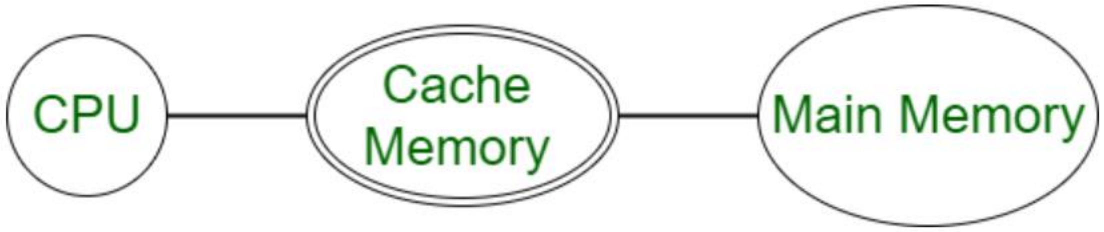
- Types of Cache
  - Primary Cache –

    A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers.
  - Secondary Cache –

    Secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the level 2 (L2) cache. Often, the Level 2 cache is also housed on the processor chip.

Cache Hit / Miss

- When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.
- If the processor finds that the memory location is in the cache, a **cache hit** has occurred and data is read from cache
- If the processor **does not** find the memory location in the cache, a **cache miss** has occurred.
- In case of cache miss, it retrieves the data from the **main memory**



Volatile

- Variables that may be modified externally at any time point (such as being shared by multiple tasks).
- Variables declared to be volatile will not be optimized by the compiler because their value can be changed at any time.
- The volatile keyword is intended to prevent the compiler from applying any optimizations on objects.
- Volatile exists for the purpose of not caching the values of the variable automatically.
- It will tell the compiler not to cache the value of this variable. So it will generate code to take the value of the given volatile variable from the main memory every time it encounters it. This mechanism is used because at any time the value can be modified by the OS or any interrupt. So using volatile will help us accessing the value afresh every time.

Volatile example

```
void foo(void)
{
    volatile int* addr;
    addr= INPUT_ADDRESS;

    *addr = 0;
    while (*addr != 255)
        ;
 }
```

union

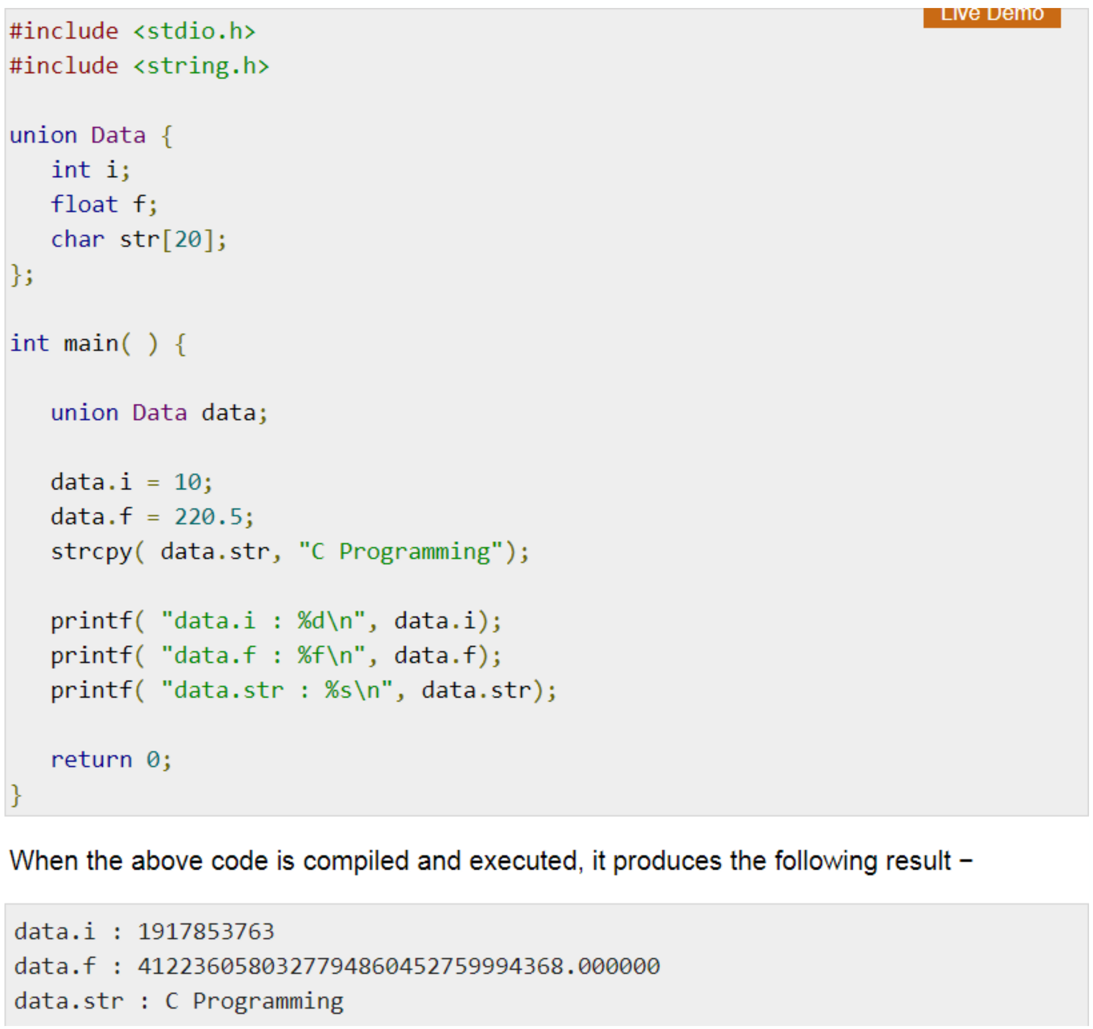
Union

- A type that keeps (in different times) different types
- When initializing a union, only the 1st component is initialized.

```
typedef union MyUnion
{
        int  i_val;
         double  d_val;
} UnExmp;
```

```
UnExmp u;

u.i_val= 3;
printf("%d\n", u.i_val);

u.d_val= 3.22;
printf("%f\n", u.d_val);
```

Union – bad example



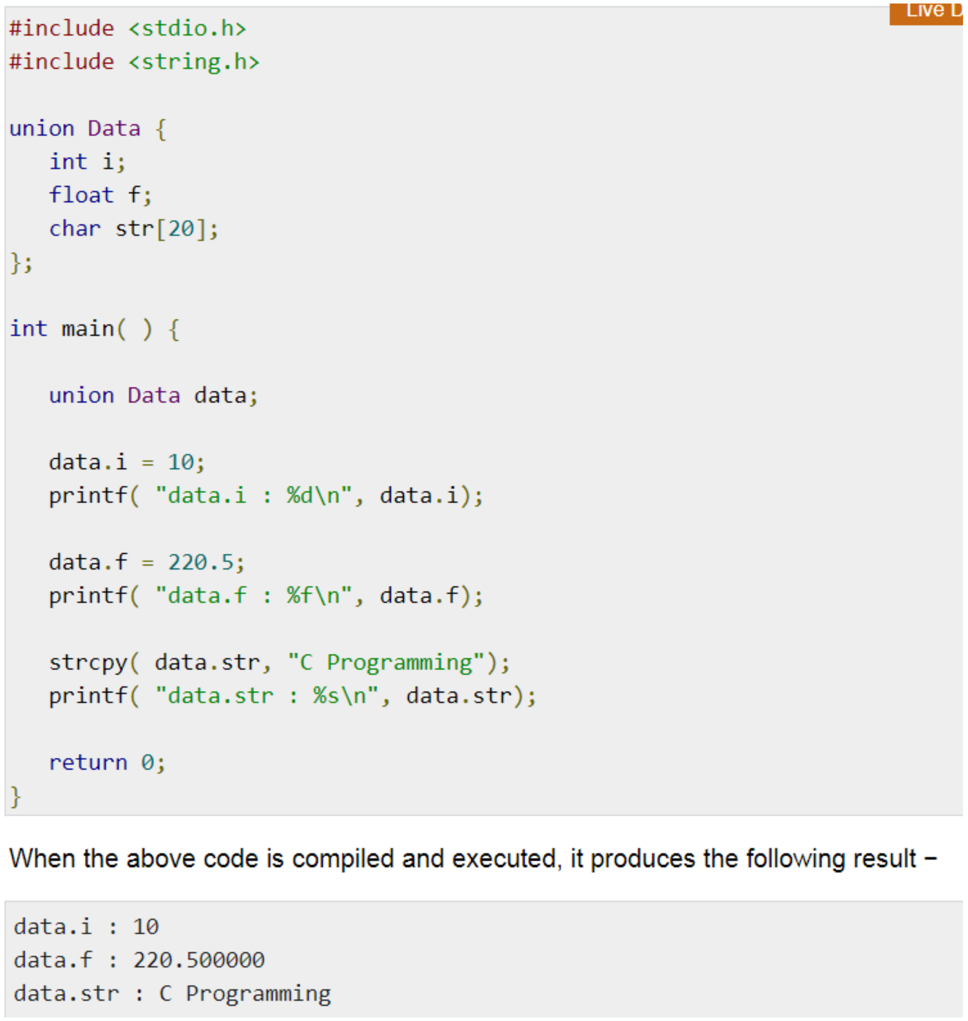
```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

- The values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Union – good example



```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

- Here, all the members are getting printed very well because one member is being used at a time.

Union Vs. Struct

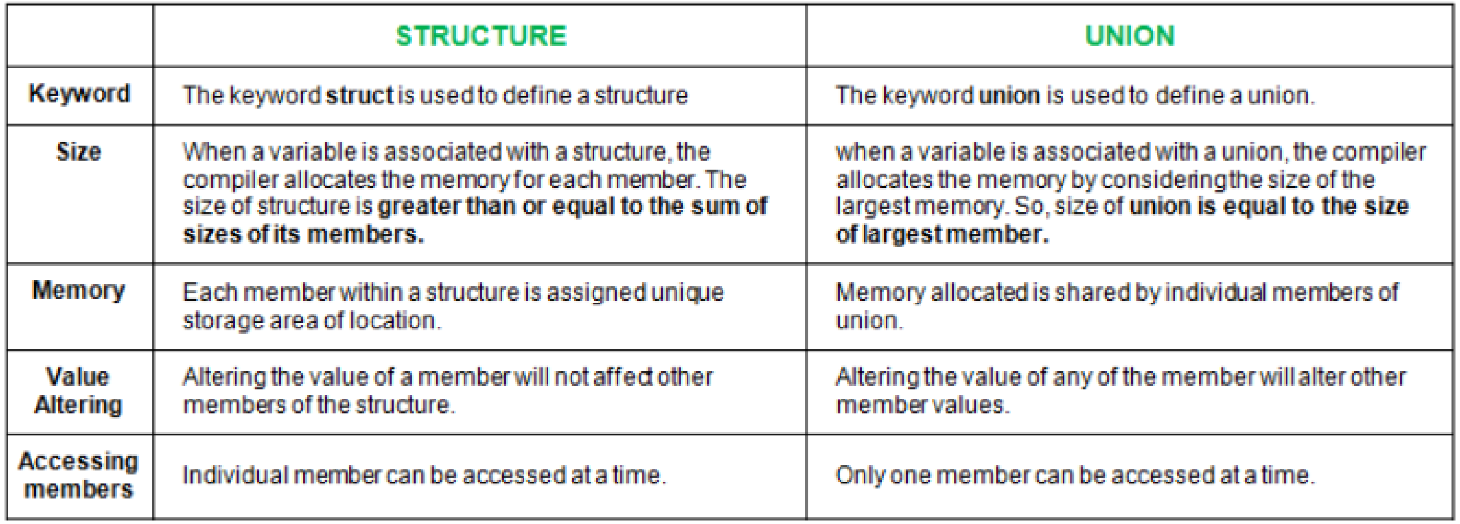
- A structure is a user-defined data type available in C that allows to combining data items of different kinds, located in different memory slots.

- A union is a special data type available in C that allows storing different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purposes.

Union Vs. Struct – Similarity

- Both are user-defined data types used to store data of different types as a single unit.
- Their members can be objects of any type, including other structures and unions or arrays.
- Both structures and unions support assignment = operator. Both structures or unions in the assignment must have the same members and member types.
- A structure or a union can be passed by value to functions and returned by value by functions.
- '.' operator is used for accessing members.

Union Vs. Struct – Comparison



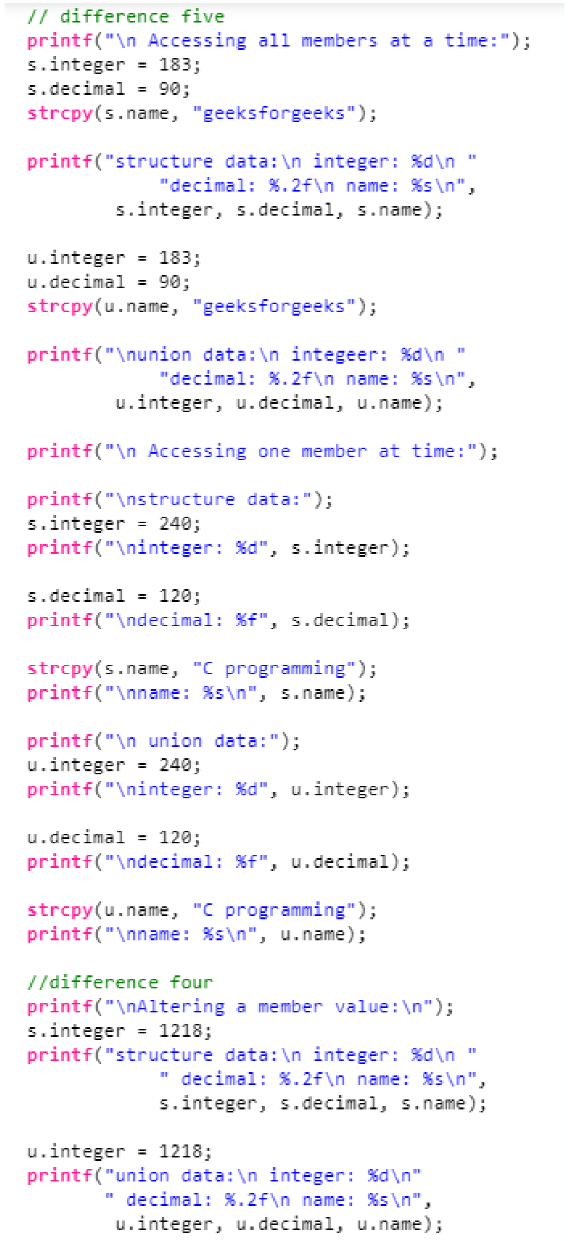
	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.

Union Vs. Struct – Example

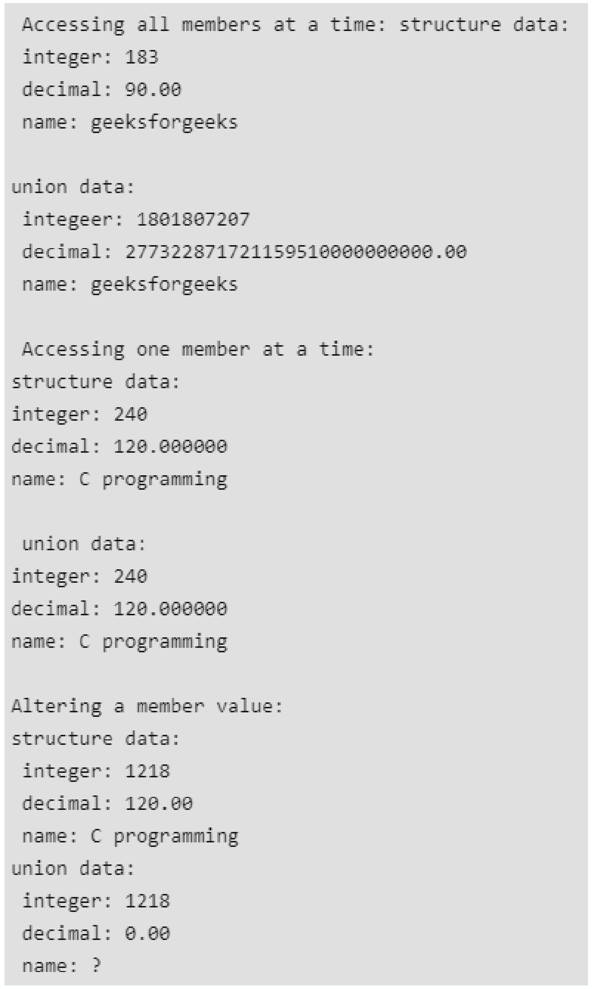
```c
// C program to illustrate differences
// between structure and Union
#include <stdio.h>
#include <string.h>

// declaring structure
struct struct_example
{
    int integer;
    float decimal;
    char name[20];
};

// declaraing union

union union_example
{
    int integer;
    float decimal;
    char name[20];
};

void main()
{
    // creating variable for structure
    // and initializing values difference
    // six
    struct struct_example s={18,38,"geeksforgeeks"};

    // creating variable for union
    // and initializing values
    union union_example u={18,38,"geeksforgeeks"};


    printf("structure data:\n integer: %d\n"
            "decimal: %.2f\n name: %s\n",
            s.integer, s.decimal, s.name);
    printf("\nunion data:\n integeer: %d\n"
             "decimal: %.2f\n name: %s\n",
            u.integer, u.decimal, u.name);


    // difference two and three
    printf("\nsizeof structure : %d\n", sizeof(s));
    printf("sizeof union : %d\n", sizeof(u));
```

```
structure data:
 integer: 18
 decimal: 38.00
 name: geeksforgeeks

union data:
 integeer: 18
 decimal: 0.00
 name: ?

sizeof structure: 28
sizeof union: 20
```

Union Vs. Struct – Example cont.



```
// difference five
printf("\n Accessing all members at a time:");
s.integer = 183;
s.decimal = 90;
strcpy(s.name, "geeksforgeeks");

printf("structure data:\n integer: %d\n "
            "decimal: %.2f\n name: %s\n",
        s.integer, s.decimal, s.name);

u.integer = 183;
u.decimal = 90;
strcpy(u.name, "geeksforgeeks");

printf("\nunion data:\n integeer: %d\n "
            "decimal: %.2f\n name: %s\n",
        u.integer, u.decimal, u.name);

printf("\n Accessing one member at time:");

printf("\nstructure data:");
s.integer = 240;
printf("\ninteger: %d", s.integer);

s.decimal = 120;
printf("\ndecimal: %f", s.decimal);

strcpy(s.name, "C programming");
printf("\nname: %s\n", s.name);

printf("\n union data:");
u.integer = 240;
printf("\ninteger: %d", u.integer);

u.decimal = 120;
printf("\ndecimal: %f", u.decimal);

strcpy(u.name, "C programming");
printf("\nname: %s\n", u.name);

//difference four
printf("\nAltering a member value:\n");
s.integer = 1218;
printf("structure data:\n integer: %d\n "
            " decimal: %.2f\n name: %s\n",
            s.integer, s.decimal, s.name);

u.integer = 1218;
printf("union data:\n integer: %d\n"
       " decimal: %.2f\n name: %s\n",
        u.integer, u.decimal, u.name);
```

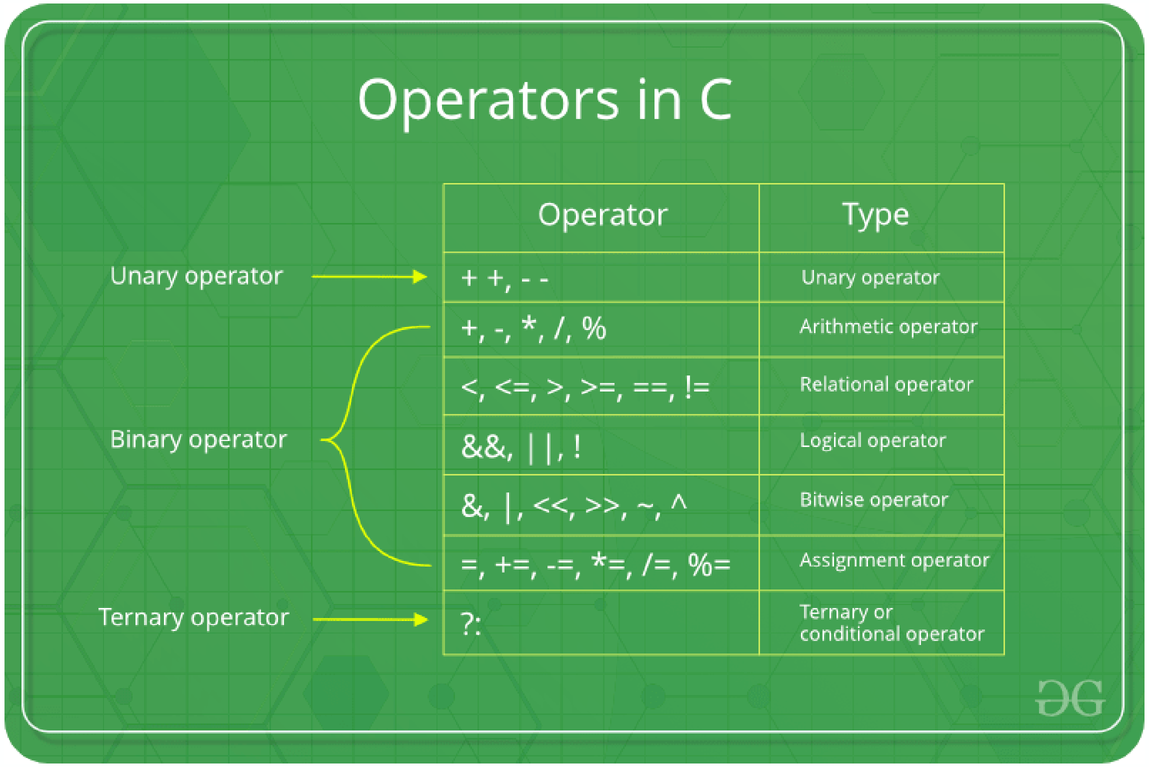


```
 Accessing all members at a time: structure data:
 integer: 183
 decimal: 90.00
 name: geeksforgeeks

union data:
 integeer: 1801807207
 decimal: 277322871721159510000000000.00
 name: geeksforgeeks

 Accessing one member at a time:
structure data:
integer: 240
decimal: 120.000000
name: C programming

 union data:
integer: 240
decimal: 120.000000
name: C programming

Altering a member value:
structure data:
 integer: 1218
 decimal: 120.00
 name: C programming
union data:
 integer: 1218
 decimal: 0.00
 name: ?
```

Bitwise Operations

Bitwise Operators

- Bitwise operations give the language the power of a “low level language”.
- For example, several flags can be kept within one byte.
- Accessing bits directly is fast and efficient, especially if you are writing a real-time application.


Unary operator
Binary operator
Ternary operator
ƏG

Operators in C

Operator	Type
+ +, - -	Unary operator
+, -, *, /, %	Arithmetic operator
<, <=, >, >=, ==, !=	Relational operator
&&, | |, !	Logical operator
&, |, <<, >>, ~, ^	Bitwise operator
=, +=, -=, *=, /=, %=	Assignment operator
?:	Ternary or conditional operator

Bitwise Operations

- The language introduces the following bitwise operators, which support manipulating a single bit or groups of bits.
- Bitwise operators may be used on integers only (**unsigned int** type is preferable).

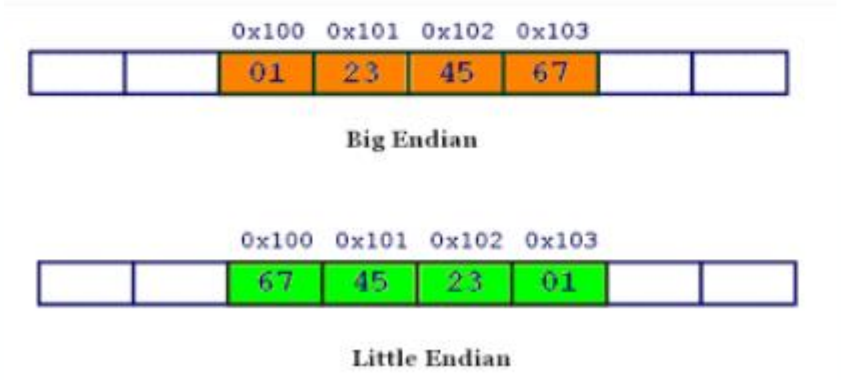
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	1's complement
<<	left shift
>>	right shift

Bitwise Operators

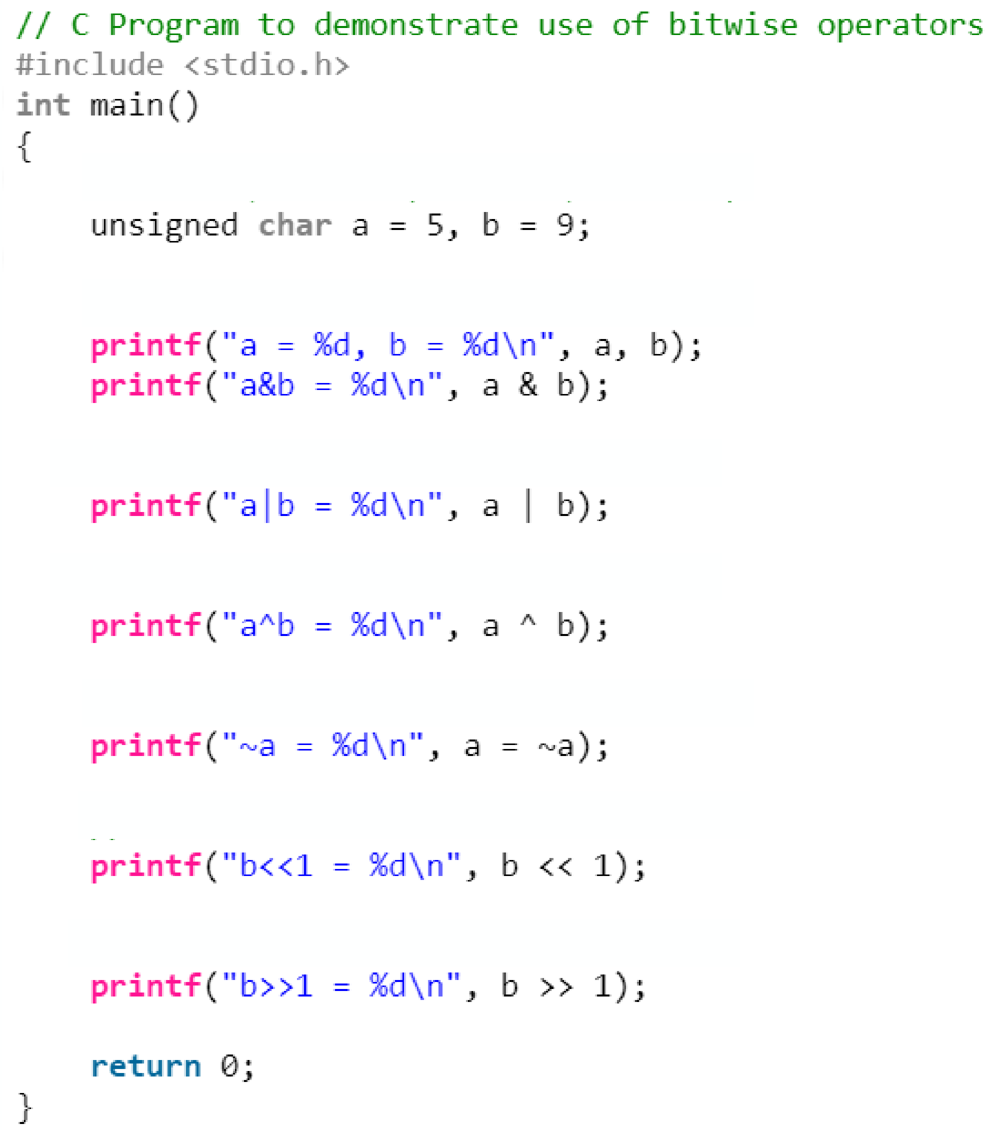
- **& (bitwise AND)** takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
- **| (bitwise OR)** takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
- **^ (bitwise XOR)** takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
- **<< (left shift)** takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
- **>> (right shift)** takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
- **~ (bitwise NOT)** takes one number and inverts all bits of it

Big/Little Endian - machine depended

- Working with bits is machine-dependent.
- Little and big endian are two ways of storing multibyte data-types (int, float, etc). In little endian machines, last byte of binary representation of the multi-byte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

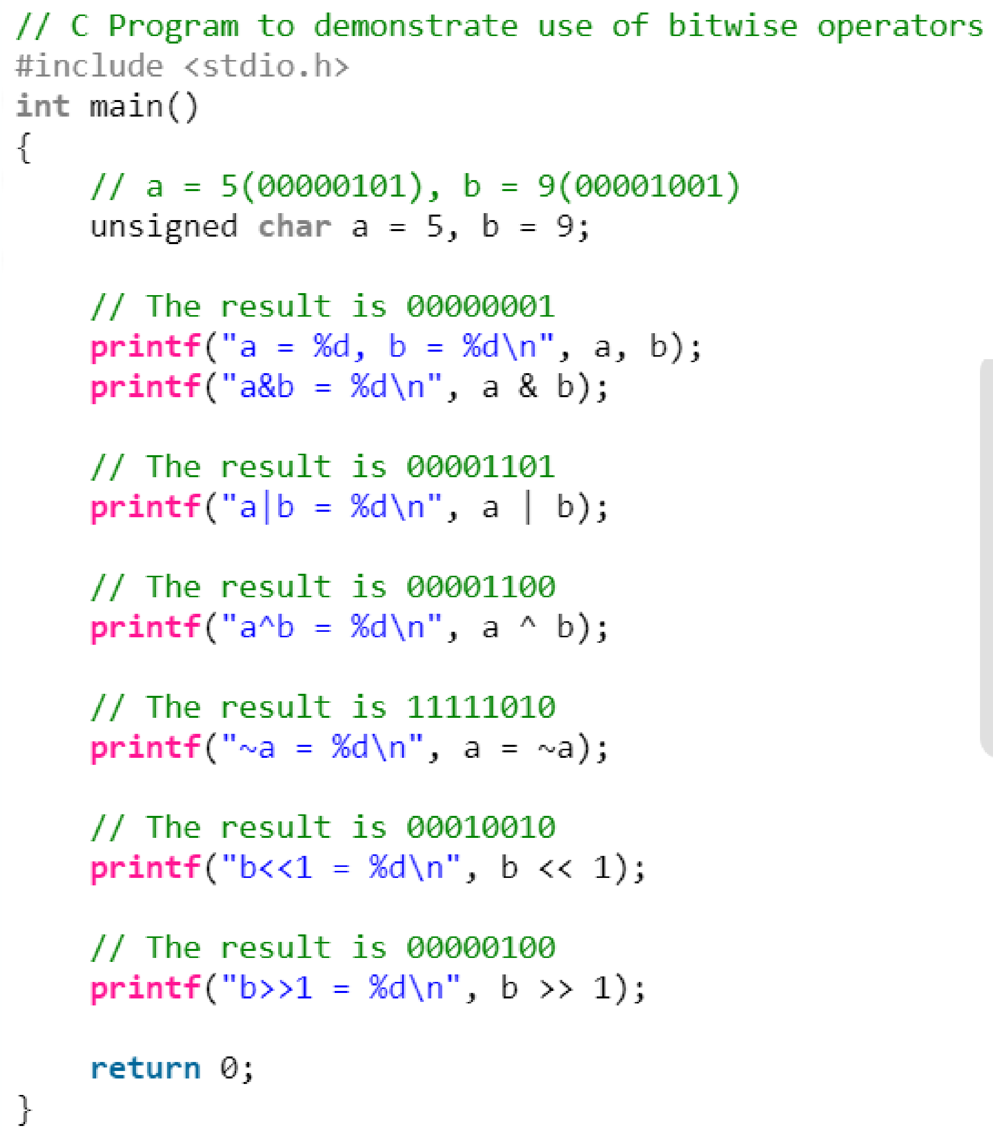


Bitwise Operators - example

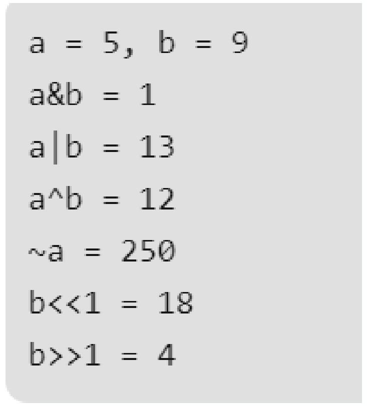


```
// C Program to demonstrate use of bitwise operators
#include <stdio.h>
int main()
{

    unsigned char a = 5, b = 9;


    printf("a = %d, b = %d\n", a, b);
    printf("a&b = %d\n", a & b);


    printf("a|b = %d\n", a | b);


    printf("a^b = %d\n", a ^ b);


    printf("~a = %d\n", a = ~a);


    printf("b<<1 = %d\n", b << 1);


    printf("b>>1 = %d\n", b >> 1);

    return 0;
}
```

Bitwise Operators - example



```c
// C Program to demonstrate use of bitwise operators
#include <stdio.h>
int main()
{
    // a = 5(00000101), b = 9(00001001)
    unsigned char a = 5, b = 9;

    // The result is 00000001
    printf("a = %d, b = %d\n", a, b);
    printf("a&b = %d\n", a & b);

    // The result is 00001101
    printf("a|b = %d\n", a | b);

    // The result is 00001100
    printf("a^b = %d\n", a ^ b);

    // The result is 11111010
    printf("~a = %d\n", a = ~a);

    // The result is 00010010
    printf("b<<1 = %d\n", b << 1);

    // The result is 00000100
    printf("b>>1 = %d\n", b >> 1);

    return 0;
}
```



```
a = 5, b = 9
a&b = 1
a|b = 13
a^b = 12
~a = 250
b<<1 = 18
b>>1 = 4
```

Bitwise Logical Operations

Operations on logical TRUE or FALSE

- two states -- takes one bit to represent: TRUE=1, FALSE=0

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XOR will be 1 if either A or B will be 1, but not both of them

Bitwise Operations in Integers

AND

- Useful for clearing bits
- Priority to 0's

$$\begin{array}{lr} & 11000101 \\ \text{AND} & \underline{00001111} \\ & 00000101 \end{array}$$

OR

- Useful for setting bits
- Priority to 1's

$$\begin{array}{lr} & 11000101 \\ \text{OR} & \underline{00001111} \\ & 11001111 \end{array}$$

NOT

- Unary operation -- one argument
- Flips every bit

$$\begin{array}{lr} \text{NOT} & \underline{11000101} \\ & 00111010 \end{array}$$

NAND

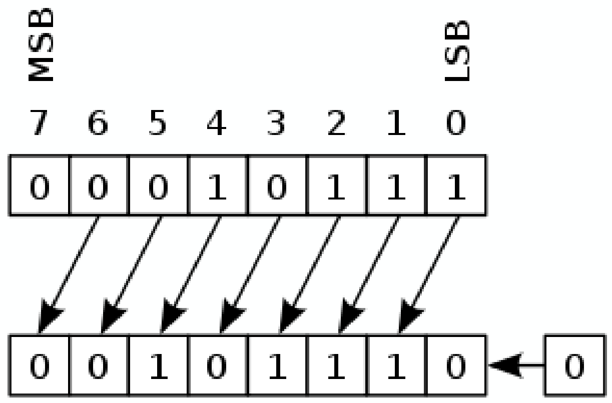
$$\begin{array}{lr} & 11000101 \\ \text{NAND} & \underline{00001111} \\ & 11111010 \end{array}$$

XOR

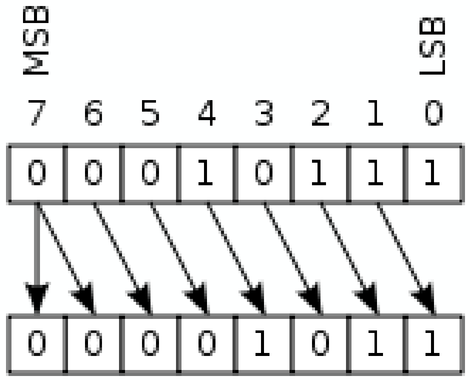
- Checking odd or even

$$\begin{array}{lr} & 11000101 \\ \text{XOR} & \underline{00001111} \\ & 11001010 \end{array}$$

Shifts Operations



Left Logical Shift

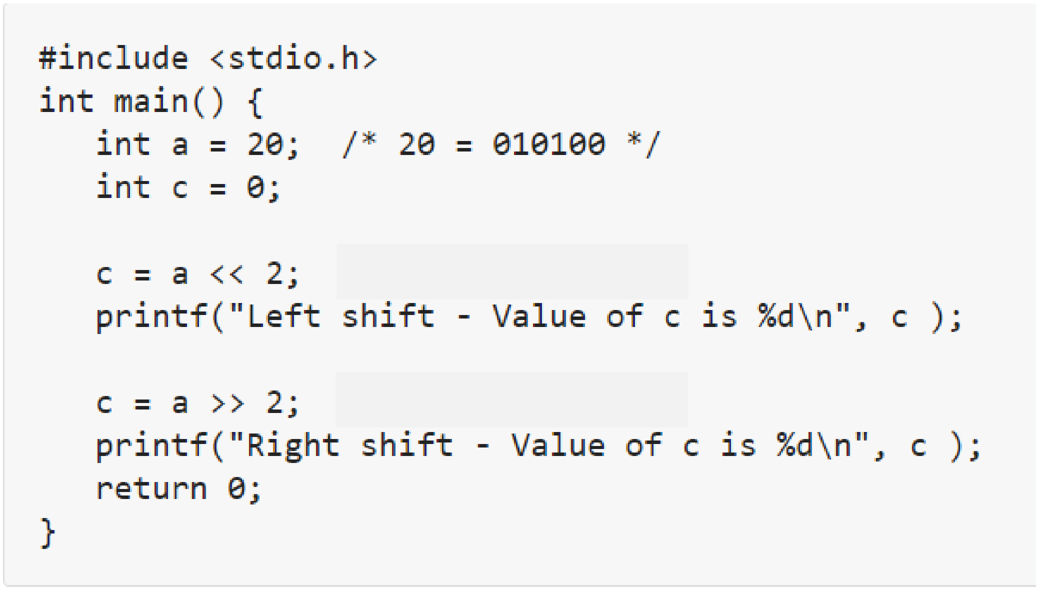


Right Arithmetic Shift

- << - Shift left - Similar to multiply by 2
- >> - Shift right - Similar to divide by 2
- Similar but not equivalent since there might be differences due to rounding strategies.

Shifts Operations - example

What are the prints of this function ?



```
#include <stdio.h>
int main() {
   int a = 20;  /* 20 = 010100 */
   int c = 0;

   c = a << 2;
   printf("Left shift - Value of c is %d\n", c );

   c = a >> 2;
   printf("Right shift - Value of c is %d\n", c );
   return 0;
}
```

Shifts Operations - solution

```c
#include <stdio.h>
int main() {
   int a = 20;  /* 20 = 010100 */
   int c = 0;

   c = a << 2;  /* 80 = 101000 */
   printf("Left shift - Value of c is %d\n", c );

   c = a >> 2;  /*05 = 000101 */
   printf("Right shift - Value of c is %d\n", c );
   return 0;
}
```

Output:

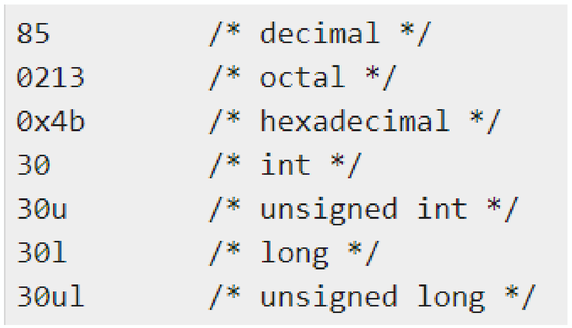
```
Left shift - Value of c is 80
Right shift - Value of c is 5
```

- a<<n == fast multiply the variable a by 2^n

C Literals

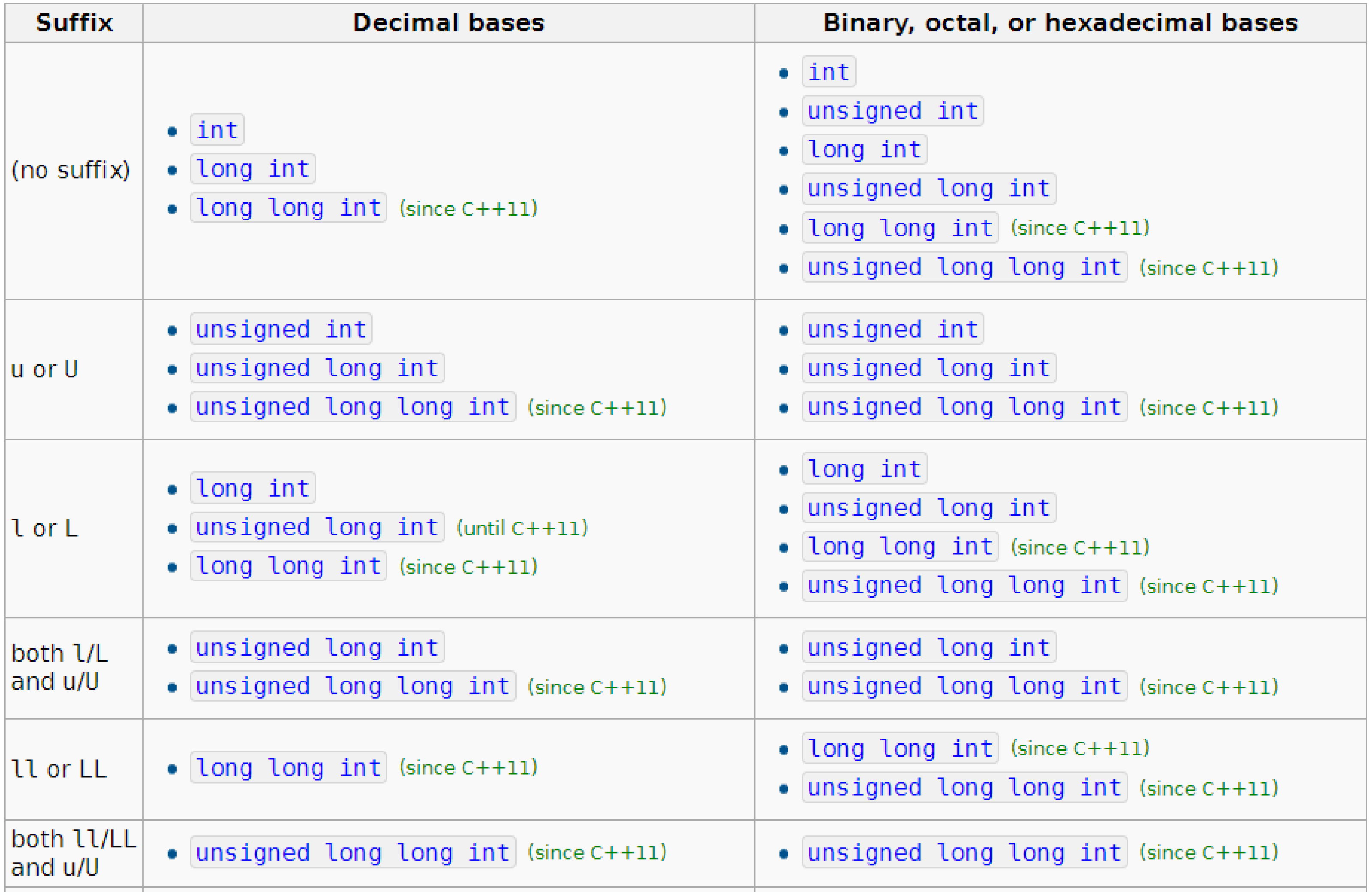
Integer Literals Rules

- An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.
- An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.
- *binary-literal* is the character sequence 0b or the character sequence 0B followed by one or more binary digits (0, 1)
- Examples:
  - 42 → decimal int
  - **042u** → **octal** (34 in decimal) **unsigned** int
  - **0x**42**L** → **hexadecimal** (66 in decimal) **long** int



```
85         /* decimal */
0213       /* octal */
0x4b       /* hexadecimal */
30         /* int */
30u        /* unsigned int */
30l        /* long */
30ul       /* unsigned long */
```

Integer Literals Rules

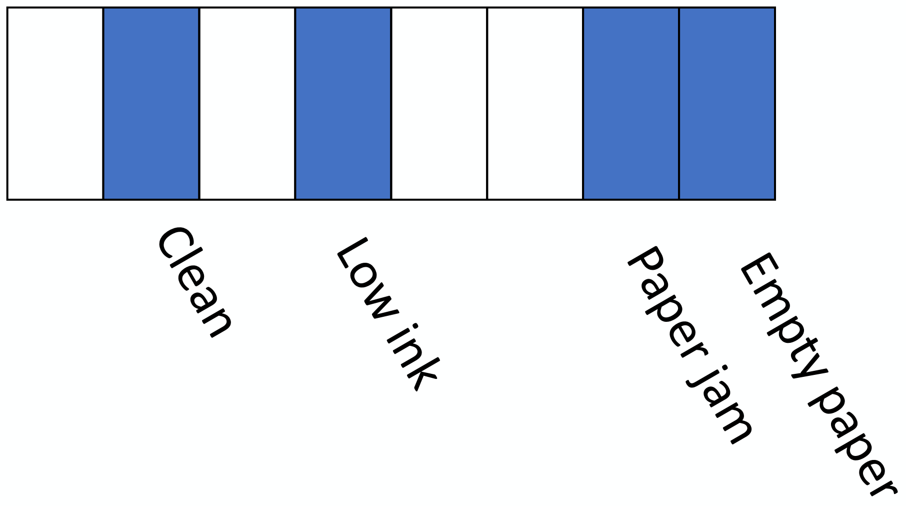


Suffix	Decimal bases	Binary, octal, or hexadecimal bases
(no suffix)	• `int` • `long int` • `long long int` (since C++11)	• `int` • `unsigned int` • `long int` • `unsigned long int` • `long long int` (since C++11) • `unsigned long long int` (since C++11)
`u` or `U`	• `unsigned int` • `unsigned long int` • `unsigned long long int` (since C++11)	• `unsigned int` • `unsigned long int` • `unsigned long long int` (since C++11)
`l` or `L`	• `long int` • `unsigned long int` (until C++11) • `long long int` (since C++11)	• `long int` • `unsigned long int` • `long long int` (since C++11) • `unsigned long long int` (since C++11)
both `l`/`L` and `u`/`U`	• `unsigned long int` • `unsigned long long int` (since C++11)	• `unsigned long int` • `unsigned long long int` (since C++11)
`ll` or `LL`	• `long long int` (since C++11)	• `long long int` (since C++11) • `unsigned long long int` (since C++11)
both `ll`/`LL` and `u`/`U`	• `unsigned long long int` (since C++11)	• `unsigned long long int` (since C++11)

Working with bits - two Approaches

- Traditional C – bit masks - #define and bitwise operations
- Modern approach - use bit fields

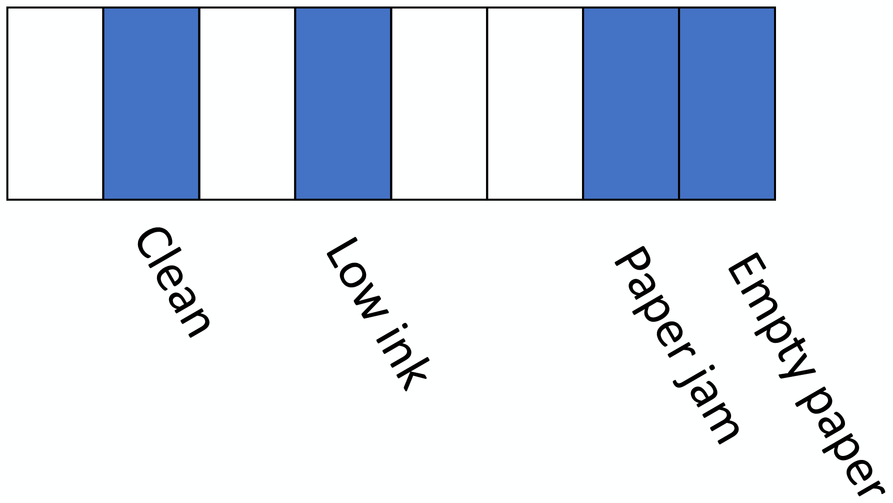
Printer Status Register - example



Definition of bit masks

```
#define EMPTY   1
#define JAM     2
#define LOW_INK 16
#define CLEAN   64
```

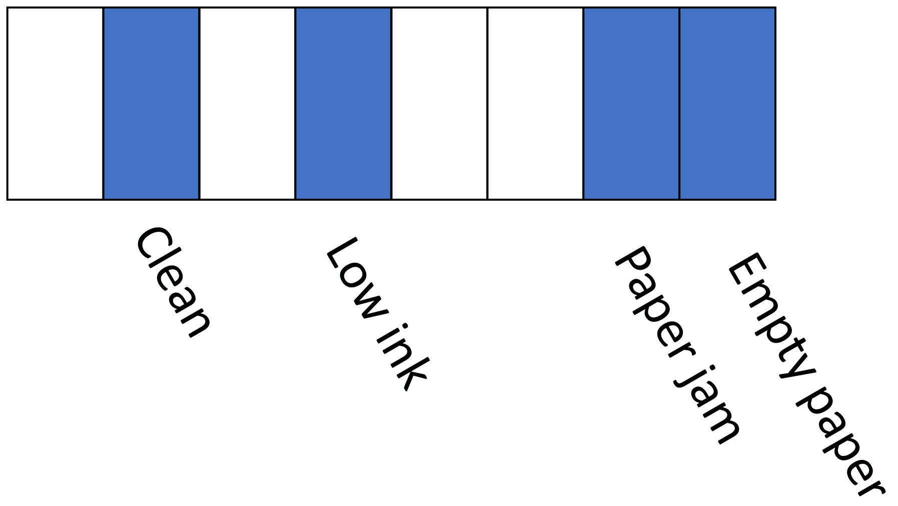
Printer Status Register - example



Definition of bit masks in octal

```
#define EMPTY   01
#define JAM     02
#define LOW_INK 020
#define CLEAN   0100
```

Printer Status Register - example



```
char status;

...

while (!(status & LOW_INK)) ...;

status|= CLEAN;   /* turns on CLEAN bit */

status&= ~JAM;    /* turns off JAM bit */
```

→ status=0, low_ink=0

Bit Fields

Bit-Fields

- Bit Fields allow the packing of data in a structure.
- Bit fields declaration –

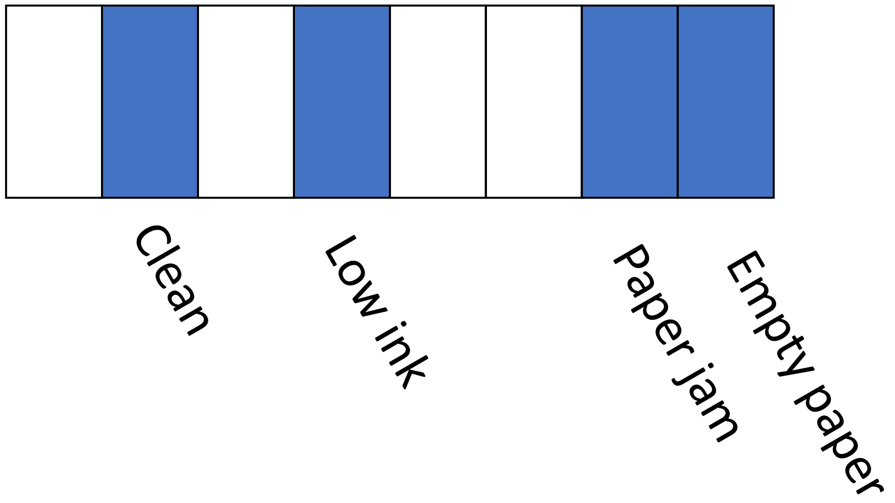
```
struct {
   type [member_name] : width ;
};
```

Sr.No.	Element & Description
1	**type** An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
2	**member_name** The name of the bit-field.
3	**width** The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

- The variables defined with a predefined width are called **bit fields**.
- A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows –

```
struct {
   unsigned int age : 3;
} Age;
```

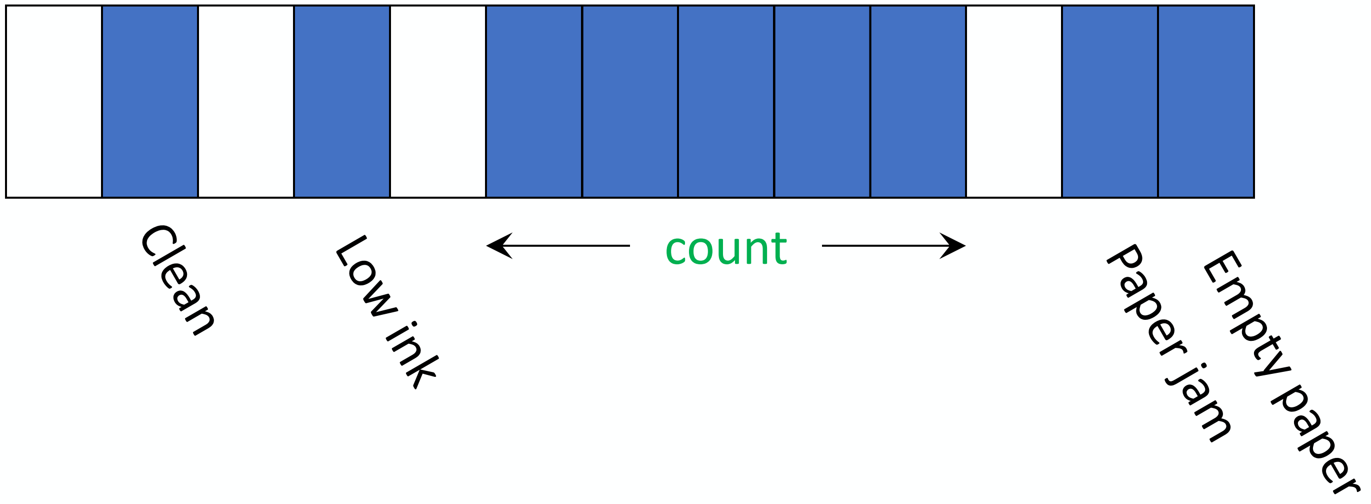
Bit-Fields



- Bit Fields allow the packing of data in a structure.
- Here the statusReg contains 4 members: 1 bit for each field.

```
struct statusReg {

        unsigned int emptyPaper     :1;
        unsigned int paperJam       :1;
        unsigned int lowInk         :1;
        unsigned int needsCleaning  :1;
};
```

Bit-Fields



```
struct statusReg {
  unsigned int emptyPaperTray        :1;
  unsigned int paperJam              :1;
  unsigned int count                 :5;
  unsigned int lowInk                :1;
  unsigned int needsCleaning         :1;
};
```

Bit-Fields

```
struct statusReg s;

if (s.emptyPaperTray || s.paperJam) ...;
while(!s.lowInk) ...;

s.needsCleaning = 1;
s.paperJam = 0;

int c = s.count;
s.count = 2;
```

https://www.geeksforgeeks.org/bit-fields-c/

Examples

Example 1

```
#include <stdio.h>
void main(void)
{
    unsigned char and, or, xor, comp, shiftL,
shiftR;
    unsigned char mask1 = 0X78, mask2 = 0XB2;

    and    = mask1 & mask2;
    or     = mask1 | mask2;
    xor    = mask1 ^ mask2;
    comp   = ~mask1;
    shiftL = mask1 << 3;
    shiftR = mask1 >> 3;

    printf("and    = %X HEXA\n", and);
    printf("or     = %X HEXA\n", or);
    printf("xor    = %X HEXA\n", xor);
    printf("comp   = %X HEXA\n", comp);
    printf("shiftL = %X HEXA\n", shiftL);
    printf("shiftR = %X HEXA\n", shiftR);
}
```

Example 2

Setting bits (1):
Performing a bitwise OR operation with 1 always results in a set bit (1)

This is how we set a bit (1)

Examples :

-
  ```
  unsigned int n, n_set;
  n_set = n |  1; /* now the LSB  of n_set is  1 */
  n_set = n | ~0; /* now ALL bits of n_set are 1 */
  ```

-
  ```
  unsigned int n = 4;                    100
  unsigned int mask = 1;                 001
  n |= mask; /* now n is 5 */            101
  ```

-
  ```
  unsigned int n = 4;                    100
  unsigned int mask = 1<<1;              010
  n |= mask; /* now n is 6 */            110
  ```

Toggling
Bitwise XOR of a bit with 1 toggles the bit.
Bitwise XOR of a bit with itself unsets the bit (0).
Bitwise XOR of a bit with its compliment sets the bit (1).

Example 3

```
This program reads an unsigned int and prints the 16
( or other size - depending on sizeof(int) )  bits that
represent it. */

#include <stdio.h>
#include <limits.h>

void b_uns(unsigned num)
{
    int i;
    int mask = 1 << (8 * sizeof(int) -1);

    for (i = 0 ; i < 8*sizeof(int) ; ++i)
    {
          if ( num &  mask)
                printf("1");
          else
                printf("0");
          num <<= 1;
    }
}
```

Example 3

```
void main(void)
{
    unsigned num;

    printf("Enter an unsigned integer (0 to %u) => ",
           UINT_MAX);
    scanf("%u" ,&num);
    printf("The number %u is represented by the following"
           "bits:\n", num);
    b_uns(num);
}
```