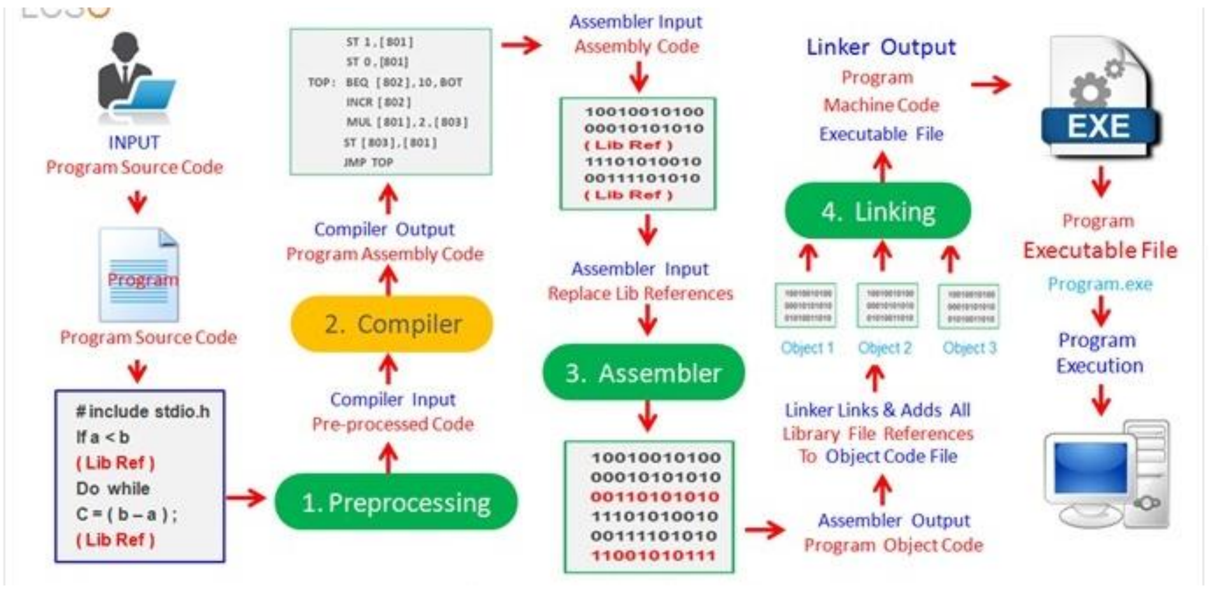
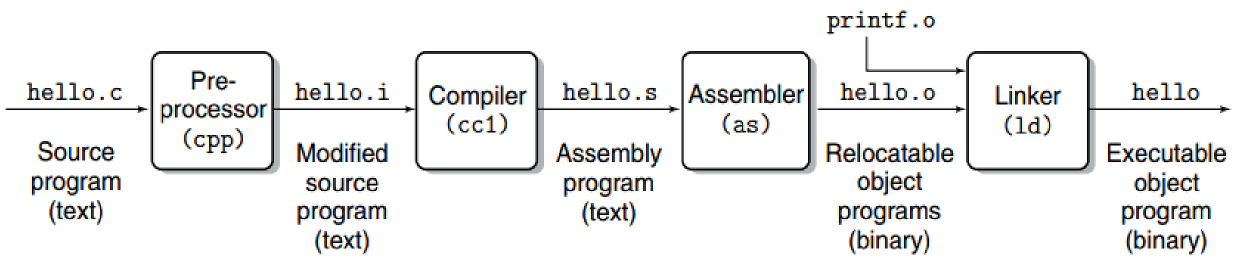
1)





hello.ciso

2) Pointer and array :

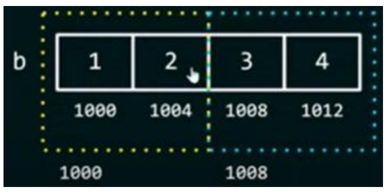
Int a[5] = {1, 2, 3};

Int *p = a; pointer p has address of a. so when we put * (dereference operator) we go inside the array so we will get first element of array a.

Cout<<*p; (output = 1)

Now, int (*p)[3]; how to read this statement ? *p means p is pointer. Statement means p is pointer to the whole 3 element array which contains integer element.

Int (*p) [5] = &a; Why &a



Consider this 2D array b now *b = pointer to 1st element of 1st 1D array. So *b contains address of first element of first array. So *b = 1000. Now, if we want to go outside the 1D we will add &(reference operator) so, &*b = b. Both cancel each other. Similarly, in green statement a is address of first element of array but p is pointer to whole array so we want to go out of address we use &. So that is why &a is written.

Let's say a = 1000; address of a

Now, *(a+1) = 20; this means a contains address of first element of array. We are adding 1 into address now size of integer is 4 so this actually represents *(1000 + 1x4) = *(1004) = 20. So *(a+i) = a[i];

You cannot do a++; Here we trying to assigning new address to array a which will result in error.

Now, again consider b array in black diagram. B = 1000 (from diagram). If ptr* = b; then ptr = b; if we do ptr++; This means ptr = ptr + 1 = (b + 2x4) because the size of 1D array is get added as you can see from diagram.

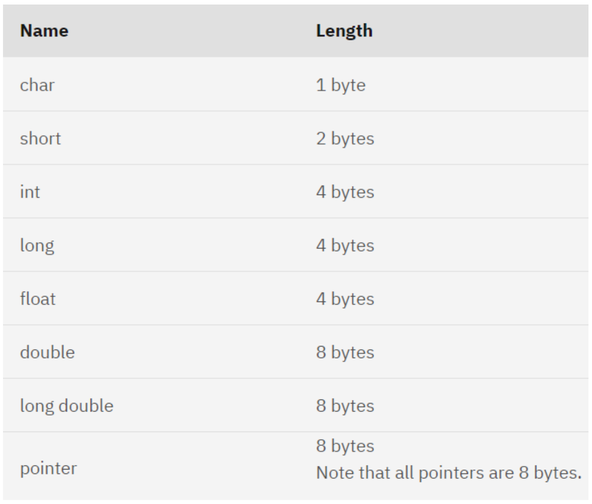
Int *a[3][5]; This is array a of pointers. But a contains address of first array of pointer. So size of *a = total pointer x size of each pointer.

Int (*a)[3][5]; This is pointer pointing to 2D array of type int. sizeof *a = total element of whole 2D array because it points to whole 2D array multiplied by size of int.

Int *(a[3][5]); this is 2D array of pointers. Which is same as first case. So, it is synonymous.
Int (*a[3])[5]; This is array of pointers of size 3 points to array of size 5 which contains integer element. So basically it is pointers of array. So size of a is 3 x size of pointers because there are 3 pointers. And if we add * then sizeof(*a) = size of pointer. because * means inside so it represents address of first element of a.

3) *Dangling pointer and memory leak problem*

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    int arr[5] = {2, 5, 3, 10, 9};
    int *p = (int *) malloc (sizeof (int));
    *p = 5;
    if (–1)
    {
        p = (int *) malloc (sizeof (int));
        *p = 6;
    }
    arr[0] = *p;
    free(p);
    for (int i = 1; i < 5, i++)
        arr[i] += arr[0];
    return 0;
}
```



Name	Length
char	1 byte
short	2 bytes
int	4 bytes
long	4 bytes
float	4 bytes
double	8 bytes
long double	8 bytes
pointer	8 bytes Note that all pointers are 8 bytes.

Here After “free(p);” line, pointer p become **dangling pointer**. Because now p is pointing to deallocated memory. In ‘if’ condition, the value is –1 and it is not zero, so if condition is true and program will execute statements inside ‘if’. Here malloc( ) function allocate new memory to pointer p and program will lost the old allocated memory to p. So, it is **memory leak**.

4) Const ptr means pointer which points to memory which read only so if we try to write some value it will show error.
5) “**extern**” keyword is used to extend the visibility of function or variable. By default, the functions are visible throughout the program, there is no need to declare or define extern functions. It just increases the redundancy.
6) **Upper** and **lower triangular matrix** in data structure is different from mathematics here for 0 entry we do not assign memory.
7) When a variable is declared as **static**, space for it gets allocated for the lifetime of the same program. When static keyword is used, variable or data members or functions cannot be modified again. If same function is called more than twice then the values of static variable remains same.
8)

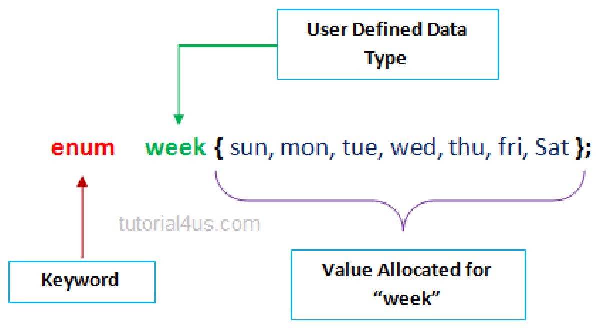
```
unsigned int x = 0xDEADBEEF;
unsigned short y = 0xFFFF;
signed int z = -1;
if (x > (signed short) y)
    printf("Hello");
if (x > z)
    printf("World");
```

Prints nothing. first y is type casted to signed short, now one operand is unsigned int and other operand is signed short, so signed short is promoted to unsigned int, sign extension will happen on y, so we’re actually checking (DEADBEEF > FFFFFFFF). This equates to False.

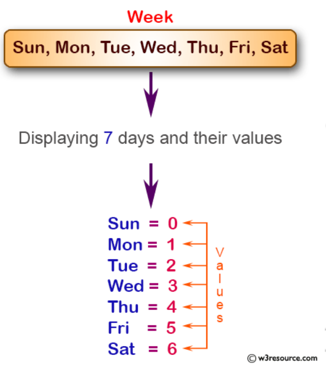
9) Ox means hexadecimal example, Ox10 means 16 in decimal, and 0 means octal example, 010 means 8 in decimal. Which means if int i = Ox10 + 010 + 10 = 34. Note that int means decimal. Now just like %d %x is used for hexadecimal.

10) Static int means consider it value only at first declaration after that consider updated value just like global variable but static variable is used in function. So static variable has scope.

11) Enum is data structure

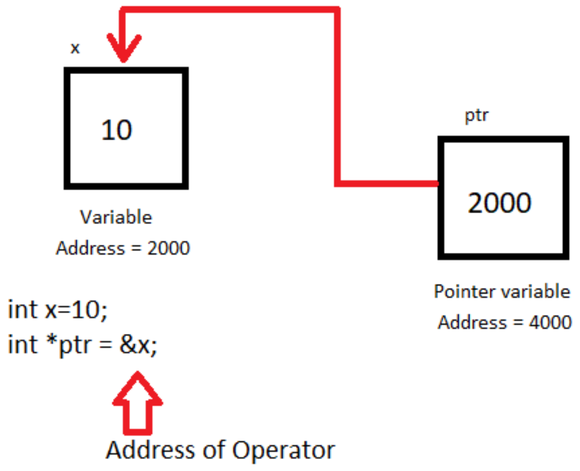


User Defined Data Type
enum
week { sun, mon, tue, wed, thu, fri, Sat };
tutorial4us.com
Keyword
Value Allocated for "week"



```
enum state{
 Mp,
 Goa=3,
 Up,
 Himachal
};
```

If the values are not assigned consider 0, 1, 2, 3,… . In above image goa has value 3 so up = 4, and himachal = 5 and so on. But Mp has 0 value.



$$(char^*)\ (\&arr+1) - (char^*)\ \&arr$$

$$\Rightarrow \frac{char*(\&arr+1)-(char*)\&arr}{size\ of\ (char*(\&arr))} = 210$$

$$= \frac{210-0}{1} = 210$$

12)

While subtracting addresses and pointers do not forget to divide it by respective size of data structure.

13) There is a no such thing as function is not defined in c. Function is not declared is the correct error.

14)

In the following program, which of the identifiers possess the lvalue (An expression that designates an object)? (An "lvalue" is a value that can be the target of an assignment. The "l" stands for "left", as in the left hand side of the equals sign.)

```
int a[20];
int b = 99;
int f( int c[20], int e )
{
    int d;
}
```

(a) a, b, c, d, e, f
(b) b, c, d, e
(c) a, b, c, d, e
(d) a, b, d

Answer: B
Solution:
Identifier possess the lvalue, in which a value is assigned .
In the given program, a[20] is an array, where 'a' is constant pointer, therefore no value is assigned to identifier 'a'.
'd','b','e' can possess lvalue, and c is a pointer and it is a parameter of function f, therefore 'c' can possess the lvalue attribute, while 'f' is a function , we can't assign any value.
∴b, c, d, e possess lvalue attribute.
∴Correct Answer is (B)

15) Addition in pointer is not allowed.

16) Int a = 1, b = -1; then value of (a)! is 0 and that of (b)! is 0. Because -1 is considered as nonzero number.

17) Inside the switch, consider case x : y =0; then the value of x should be constant case - - x : y = 0; this gives error as - - x is not constant.

```
#include <stdio.h>
void dynamic(int s, ...)
{
    printf("%d", s);
}
int main( )
{
    dynamic(3, 4, 6, 7);
    return 0;
}
```

18) In C, (…) is operator also known as ellipsis. Which is variable number of argument of function. So output is still 3. It does not change the function of program.

19) If(a=f(x))print("%d", 1);else print("%d", 0). In this case, first f(x) will run then it will give value to the a then if consider current value of a not old value and if a is non-zero then it will print 1 or else 0.

```
int a[ ] = {1, 2, 3, 4, 5, 6};
int *ptr = (int*)(&a + 1);
```

20) This means ptr points to end of the array means *(ptr – 1) is 6. And also *((int *)(ptr -1)) is 1.

Consider the following C code:

```
#include<stdio.h>
struct Point
{
   int x ;
   int y ;
};
struct Point foo (struct Point p1 , struct Point *p2)
{
   p1.x += p2->x;
   p1.y += p2->y;
   p2->x += p1.x;
   p2->y += p1.y;
   return p1;
}
int main ()
{
   struct Point a = {1, 2};
   struct Point b = {3, 4};
   struct Point c = foo(a, &b);
   printf ("%d, %d, %d, %d, %d, %d ", a.x, a.y, b.x, b.y, c.x, c.y);
   return 0;
}
```

The sum of value printed by above code is________

21) In above code content of a does not change because it's not been called by reference. But content of b change because it's been called with reference in foo function. So P1 in foo is just a variable with exact replica of a but it is not a whereas P2 is b itself.

22) You cannot apply modulus function with double or float.

23) Consider the following initialization of an array and pointer.

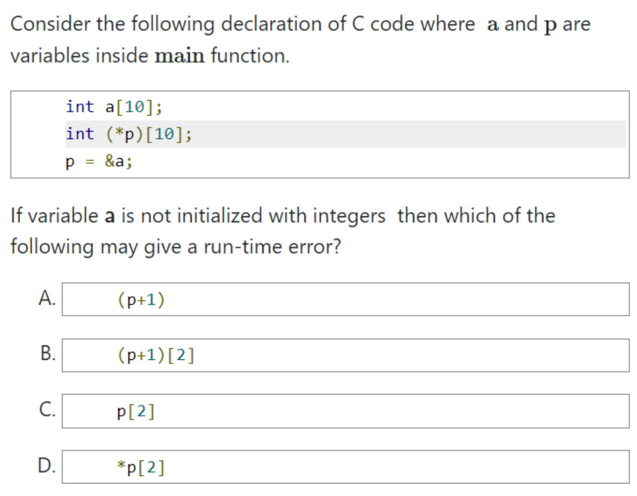
char a[] = "abc\0de";

char *p = a ;

The size of int is 4 bytes and the size of char is 1 byte. Remember p carry address of a which is int.

Then sizeof(a) = 7, sizeof(p) = 4 , sizeof(*p) = 1, strlen(p) = 3.

24)



Consider the following declaration of C code where a and p are variables inside main function.

```
int a[10];
int (*p)[10];
p = &a;
```

If variable a is not initialized with integers then which of the following may give a run-time error?

A. `(p+1)`

B. `(p+1)[2]`

C. `p[2]`

D. `*p[2]`

doing **int (*p)[10]** means p is pointing to an array of size 10.

Now, Option A → (p+1) = (p + 1***sizeof**(*p)) = (p + 10*4) → address

Option B → (p+1)[2] = *(p+1 + 2) = *(p+3) = *(p+3*10*4) = *(p + 120) → address of cell which contains some value.(how ?)

Option C → p[2] = *(p+2) = *(p + 2*10*4) = *(p + 80) → address of cell which further contains some value.(how ?)

Option D → *p[2] = *(*(p+2)) = *(address same as option C) → **integer**

25)

```
struct x {
int i;
struct x *ptr;
};
struct x arr[4] = {   6, arr + 1,
                      7, arr,
                      8, arr + 3,
                      9, arr + 2
                   };
struct x *aptr[ ] = {arr, arr + 2, arr + 1, arr + 3};
struct x **aa = aptr;
int main
{
    print("%d", aptr[aa[0] → ptr → i%2] → i++);
}
```

aptr :	arr	arr + 2	arr + 1	arr + 3

aa[0] = arr

(arr) → ptr = (arr + 1)

(arr + 1) → i = 7

7 % 2 = 1

aptr[1] = (arr + 2)

(arr + 2) → i = 8

8++

∴ Printed value = 8