



Automotive & Embedded Info

Never Forget Basics Whether its Life or Anything Else ... Basics are Cores. While seeing a Tree how can we forget Seed...

C

There are many good books of C language available in market/internet. Here intention of below information is to provide quick revision of C. You can provide your feedback to improve the quality of content.

C :

Data Types:

Basic Data Types: – INT, CHAR, FLOAT & DOUBLE

Derived Data Types: –Data types that are derived from fundamental data types are called derived data types. Derived data types don't create a new data type but, instead they add some functionality to the basic data types. Array, structure, union, pointer & function types are derived data types.

Enumeration Data Types: –They are used to define variables that can only be assigned certain discrete integer values throughout the program.

Void Data Types: –The data type void actually refers to an object that does not have a value of any type

User Defined Data Types: – Using **typedef** we create user defined data types.

Size of Data Types: –

DATATYPE	DEV C	TURBO C	KEIL C51	KEIL MDK	KEIL ARM
CHAR	1	1	1	1	1
INT	4	2	2	4	4
FLOAT	4	4	4	4	4
LONG	4	4	4	4	4
DOUBLE	8	8	4	8	8
—	—	—	—	—	—

Some C Keyword:

Modifier: –The modifiers define the amount of storage allocated to the variable. The standard c modifier are: SHORT, LONG, SIGNED, UNSIGNED.

Qualifiers: –It is used to refine the declaration of a variable, function and parameter. Standard c qualifier:

Qualifiers for Data type: –CONST, VOLATILE, STATIC, AUTO, EXTERN, REGISTER.

Qualifiers for Function: –STATIC, EXTERN, INLINE

Typecasting: – IMPLICIT & EXPLICIT

Constant: – via MACRO, ENUM & CONST.

DOUBLE, ELSE, ENUM, EXTERN, FLOAT, FOR, GOTO, IF, INT, LONG, REGISTER, INT, SHORT, SIGNED, SIZEOF, STATIC, STRUCT, SWITCH, TYPEDEF, UNION, UNSIGNED, VOID, VOLATILE, WHILE.

Decision, Loop and Case structure:

Decision Control Structure: – IF, IF-ELSE, IF-ELSEIF, GOTO

Loop Control Structure: – WHILE, DOWHILE, FOR

Case Control Structure: – SWITCH, CONTINUE, BREAK

FUNCTIONS:-

Definition: –A function is a named block of code that performs a task and return control to a caller.

Types of Function: – VOID ABC(VOID);, VOID ABC(INT A);, INT ABC(VOID);, INT ABC(INT A);,

Inline Function: –

The programmer has requested that the compiler insert the complete body of the function in every place that the function is called, rather than generating code to call the function in the one place it is defined.

Inline expansion is used to eliminate the time overhead (excess time) when a function is called. It is typically used for functions that execute frequently. It also has a space benefit for very small functions, and is an enabling transformation for other [optimizations](#).

inline. The programmer has little or no control over which functions are inlined and which are not. Giving this degree of control to the programmer allows for the use of application-specific knowledge in choosing which functions to inline.

Note: –Inline functions are faster because you don't need to push and pop activity to on/off the stack like parameters and the return address.

Function Arguments: – FORMAL & ACTUAL.

MEMORY& STORAGE CLASS:

To understand storage classes of it is quite necessary to have some understanding of memory.

MEMO RY	RAM/DA TA	STACK	Execute current execution of program.
		HEAP	Used in dynamic memory allocation. It manages by realloc, calloc, malloc & free.
		BSS (UNINITIALIZED DATA)	Contain uninitialized global & static variables
		DATA (INITIALIZED DATA)	Initialize by programmer. Classified into initialized read only area and initialized read/write area. char s[] = "hello world" and int debug=1 outside the main would be

		<p>stored in initialized read-write area.</p> <p>const char* string = "hello world" makes the string literal "hello world" to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.</p>
	CPU	SFR & I/O Registers
ROM/CODE	TEXT SEGMENT	<p>Contain executable instructions.</p> <p>Fixed size and read only.</p>

Storage Classes & Their Lifetime: –

AUTO: – Default storage class of variable declare inside any function. Scope of an auto variable is within that block. Lifetime- when program begin execution the block of code that contain auto var and exist when the program leaves the block of code. Storage- stack.

STATIC: – Variable/function define with this class have access to within that file and if define inside any function then with in that function. Static variable initializes only once. Lifetime – throughout the program execution. Storage- data segment.

REGISTER: – Variable define with this class store in CPU register to the fast access of that variable.

A register declaration is equivalent to an auto declaration, but hints that the

CPU registers, not in memory.

if a variable is declared register, the unary & (address of) operator may not be applied to it, explicitly or implicitly. Register variables are also given no initial value by the compiler.

EXTERN: – Default storage class of variable declare outside of any function. Scope of an extern variable with extern, is throughout all the program. Lifetime – throughout the program execution. Storage- data segment

LIFETIME: – Three lifetimes. 1. For static and extern variable, before main () is called until the program exits. 2. For function arguments and automatics, from the time function is called until it returns. 3. For dynamically allocated data.

Static Memory Allocation: –

In static memory process of allocating memory happens at compile time before the associated program is executed.

Dynamic Memory Allocation: –

In dynamic memory process of allocating memory happens any time even when program is running. Dynamic memory allocation function are: – malloc, realloc, calloc, & free.

Malloc malloc () function is used to allocate space in memory. It does not initialize memory. It carries garbage value. Returns null pointer if could not able to allocate requested amount of memory. Syntax: – malloc (number *sizeof (int)); malloc takes a single argument.

Calloc calloc () function is also like malloc () function. But calloc () initializes the allocated memory to zero. But malloc () doesn't. Syntax: – calloc

(number, sizeof (int)); calloc takes a two arguments.

The important difference between malloc and calloc function is that calloc initializes all bytes in the allocation block to zero and the allocated memory may/may not be contiguous.

Realloc realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size. If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block. Realloc (pointer__name, number * sizeof (int));

Free free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system. Syntax: – free (ptr);

How to free a block of memory previously allocated without using free?
Using Realloc.

ARRAY & STRINGS:

Why Array can't Pass by Value: –

When we define any array, then it occupy contiguous location of memory. Array member are access using pointer arithmetic like arr[i] is treated as *(arr+i) by the compiler. Now suppose we have an array X[3], when we pass array X, its equivalent to *(arr+i)/*(arr+0)/*(arr)/&X[0] i.e. pointer to the first element. Its type is therefore, INT*, and a called function use this pointer to indirectly access the element of the array.

Array: – Collection of similar data elements. Types single & multidimensional.

In an array removal of duplicate number. Array reverse order, ascending

order and descending order.

String: – Sequence (array) of character terminated by null character. String reversal, reverse by word only.

STRING FUNCTIONS

```
INT STRCMP (*STR1, *STR2), INT STRNCMP (*STR1, *STR2, LENGTH);
```

```
INT STRLEN (*STR1);
```

```
CHAR * STRCAT (*STR1, *STR2), STRNCAT (*STR1, *STR2, LENGTH);
```

```
STRCPY (*STR1, *STR2), STRNCPY (*STR1, *STR2, LENGTH);
```

```
CHAR *STRCHR (*STR1, CHAR);
```

```
INT STRCSPN, (*STR1, *STR2);
```

```
INT STRXFRM (*STR1, *STR2, LENGTH);
```

STRUCTURE & UNIONS:

STRUCTURE: – Collection of variable of different type under a single name.

UNION: – Collection of variable of different type under a single name.

NOTE: – Only difference between structure and *unions* are the way memory is allocated.

Union real time example: –

```
union time{ float time_in_MS; Double time_in_NS;};
```

```
union TcpIp_In4_Addr { unsigned int32; unsigned char[4] };
```



```
union HW_Reg { uint32 HWPortReg1, uint32 HWPortReg1Bit1:1,
..... uint32 HWPortReg1Bit32:1}'
```

Bit Fields:

A bit field is used within structure to address a single bit, multiple bit together etc.

Padding:

In order to align data in memory, one or more empty bytes (addresses) are inserted (or left empty) between the memory address which are allocated for other structure members while memory allocation. This concept is called structure padding.

Note: – Architecture of computer in such a way that it can read one word from memory at a time. One word is equal to 2 bytes for 16 bit processor 4 bytes for 32 bit processor and 8 byte for 64 bit processor. In c int & float takes 4 byte each and char takes 1 byte.

EX: – STRUCT STUDENT {INT ID1, INT ID2, CHAR A, CHAR B, FLOAT PERCENTAGE}.

Structure will occupy 14 byte but actually it will occupy 16 bytes.

EX: – STRUCT STUDENT {INT ID1, CHAR A, INT ID2}

Structure will occupy 9 byte but actually it will occupy 12 bytes. Here **#PRAGMA PACK (1)** can be used for arranging memory for structure member very next to the end of other structure member.

Packing:

Packing, on the other hand prevents compiler from doing padding – this has to be explicitly requested – under GCC it's `__attribute__((__packed__))`, so the following:

```
struct __attribute__((__packed__)) mystruct_A {  
  
    char a;  
  
    int b;  
  
    char c;};
```

Generally packed structures will be used:

- to save space
- to format a data structure to transmit over network using some protocol (this is not a good practice of course because you need to deal with endianness)

Slack Bytes: –

To store any type of data in structure there is minimum fixed byte which must be reserved by memory. This minimum byte is known as slack byte. Word boundary depends on machine.

```
#pragma align 1
```

```
#pragma pack 1
```

Architecture of computer in such a way that it can read one word from memory at a time. One word is equal to 2 bytes for 16 bit processor 4 bytes for 32 bit processor and 8 byte for 64 bit processor. By using this above pragma align one word will become equal to 1 byte.

EX: – STRUCT {CHAR X; INT I; CHAR}

If #pragma align is assign to 1 then now structure will occupy 5 bytes.

Diff B/W Structure & Union: –

We can access all member of structure at any time but in union only one member can access at any time. In structure memory is allocated to all members but in union memory is allocated to variable which requires more memory. All members of structure can be initialized but in union only the first member of can be initialized.

What is a self-referential structure?

A structure containing the same structure pointer variable as its element is called as self-referential structure.

POINTERS:

A pointer is a variable that stores address of (point to) another variable of same data type.

Differencing: – Obtain the address of a data item held in another location from (a pointer). Asterisk (*) indirection operator is used along with pointer variable while *Dereferencing* the pointer variable.

Void/Generic Pointer: – A void pointer can store address of (point to) another variable of any data type.

```
void *ptr; int j = 10; ptr = &j;
```

Null Pointer: – Null pointer is a pointer which points a NULL value or base address of segment.

(SYNTAX `INT *PTR= (INT *) 0; CHAR *PTR= (CHAR *) 0; FLOAT*PTR= (FLOAT *) 0; CHAR*PTR='\0'; INT *PTR=NULL`).

Constant Pointer: – Once a constant pointer points to a variable then it cannot point to any other variable.

(SYNTAX `INT *CONST PTR` 😊)

Pointer to Constant: – A pointer through which one cannot change the value of variable it points is known as pointer to constant.

(SYNTAX `CONST INT* PTR` 😊)

Pointer to Data Types: –

(SYNTAX `INT I,*J; CHAR C,*D; FLOAT F,*G || J=&I; D=&C; || G=&F`).

Pointer to Array: –

(SYNTAX `int i [10],*j; || j=&i[0]; j=i` 😊)

Pointer to array of definite size: `int (*ptr)[3];`

Pointer to array for two dimensional array:- `int tab1[100][280];`

`int (*pointer)[280]; // pointer creation`

`pointer = tab1; //assignment`

`int (*pointer)[100][280]; // pointer creation`

`pointer = &tab1; //assignment`

```
pointer[5][12] = 517; // use

int myint = pointer[5][12]; // use
```

```
(*pointer)[5][12] = 517; // use

int myint = (*pointer)[5][12]; // use
```

Array of Pointer: –

```
(SYNTAX      INT      (*PTR)      [MAX],      CHAR      *[]      =
{"ZARA", "HINA", "SARA", "ZUNA"})
```

Pointer to Structure: –

```
(SYNTAX STRUCT EMP__DATA {CHAR NAME []; INT AGE; CHAR GEN ;} EMP
[33], *PTR; PTR=&EMP [0] 🤔)
```

Pointer to Function/Function Pointer: –

A function pointer points to executable code in memory.

```
int sum(int a, int b);
```

```
int (*ptr) (int x, int y);
```

```
ptr = &sum;
```

```
result = (*ptr) (10, 2);
```

Array of Function Pointer: –

```
int sum(int a, int b);
```

```
int subtract(int a, int b);
```

```
int mul(int a, int b);
```

```
int div(int a, int b);
```

```
int (*p[4]) (int x, int y) = {&sum, &subtract, &mul, &div};
```

```
result = (*p[index]) (10, 2);
```

Dangling Pointer: –

If any pointer is pointing the memory address of any variable but after some variable has deleted from that memory location while pointer is still pointing such memory location. Such pointer is known as dangling pointer and this problem is known as dangling pointer problem.

Difference between Dangling Pointer and Memory Leak:-

When you free an area of memory, but still keep a pointer to it, that pointer is dangling:

```
char *c = malloc(16);
```

```
free(c);
```

```
c[1] = 'a'; //invalid access through dangling pointer!
```

Uninitialized/Wild Pointer: –

An uninitialized pointer points to an unknown memory location.



Forward reference w.r.t pointer in C?

Forward Referencing with respect to pointers is used when a pointer is declared and compiler reserves the memory for the pointer, but the variable or data type is not defined to which the pointer points to. For example:

```
Struct A *p;
```

```
Struct A{
```

```
...members};
```

Can math operation be performed on a void pointer?

No. Pointer addition and subtraction means advancing the pointer by a number of elements. But in case of a void pointer, we don't know for sure what it's pointing to, so we don't know the size of what it's pointing to. That is why pointer arithmetic cannot be used on void pointers.

Pointer Arithmetic: –

	OPERATIONS		OPERATIONS
1	ADDRESS + NUMBER=ADDRESS	8	ADDRESS & ADDRESS = ILLEGAL
2	ADDRESS + NUMBER=ADDRESS	9	ADDRESS ADDRESS = ILLEGAL
3	ADDRESS – ADDRESS=ADDRESS	10	ADDRESS ^ ADDRESS = ILLEGAL
4	ADDRESS + ADDRESS=ILLEGAL	11	~ADDRESS = ILLEGAL

5	ADDRESS + ADDRESS=ILLEGAL	12	ADDRESS>= /<= /!= /</> ADDRESS – LEGAL
6	ADDRESS / ADDRESS = ILLEGAL	13	
7	ADDRESS % ADDRESS = ILLEGAL	14	

Assigning A Specific Memory Location To A Pointer: –

```
VOID* P =(VOID*) 0X28FF44;
```

```
CHAR* P =(CHAR*) 0X28FF44;
```

Near, Far & Huge Pointer: –

Four registers are used to refer to four segments on the 16-bit x86 segmented memory architecture. Ds ([data](#) segment), cs ([code](#) segment), ss ([stack](#) segment), and es (extra segment). Another 16-bit register can act as an offset into a given segment, and so a logical address on this platform is written *segment: offset*.

NEAR→Near pointers refer to the current segment. They are the fastest pointers, but are limited to point to 64 kb of memory (the current segment).

Far→the pointer which can point or access whole the residence memory of ram i.e. which can access all 16 segments is known as far pointer. Size of far pointer is 4 byte or 32 bit. (Syntax→int far *ptr;)

huge→Huge pointers are essentially far pointers, but are normalized every time they are modified so that they have the highest possible segment for that address. This is very slow but allows the pointer to point to multiple segments, and allows for accurate pointer comparisons

C OPERATORS & OPEARTIONS:

Bit Wise Operators & Operations: –

OPERATORSà OR(|), AND(&), X-OR (^), LEFT SHIFT(<<), RIGHT SHIFT(>>), INVERT(~)

BIT OPERATIONS

EX:

1. SET A BIT NUM! = (1<<BIT). **#define SETBIT(X, Y) ((X) |= ((1) << (Y)))**
2. CLEAR A BIT NUM &= ~ (1<<BIT). **#define CLRBIT(X, Y) ((X) &= ~ ((1) << (Y)))**
3. TOOGLE A BIT NUM ^= (1<<BIT). **#define TGLBIT(X, Y) ((X) ^= ((1) << (Y)))**

Logical Operators: – OPERATORSàOR (| |), AND (&&), NOT (!)

Comparison Operators:- OPERATORSàEQUAL TO (==), NOT EQUAL TO (! =), GREATER THAN (>), GREATER THAN OR EQUALA TO (>=), LESS THAN (<), LESS THAN OR EQUAL TO (<=).

Operator and Precedence Rank

Operator	Precedence Rank
Pre Increment	1
Post Increment	2
Arithmetic Operator	3
Assignment Operator	4

Arithmetic Operators: –

OPERATORS → PLUS (+), MINUS (-), ASSIGN (=), MULTIPLICATION (*), DIVIDE (/), PERCENTAGE (%), INCREMENT (++), DECREMENT (--).

Condition Operator / Ternary Operator: –

Conditional operators return one value if condition is true and returns another value if condition is false

```
#include<stdio.h>

Int main(void)

{

Int x=1, y;

Y = (x==1 ? 2:0)

Printf(“%d %d”, x, y);

Getch();

Return 0;

}
```

Member & Pointer Operator: –

OPERATORS → ARRAY (ARR [A]), POINTER VALUE/DEFINITION (*A),

SOME KEY POINTS:

Enumeration Size: –An enum is only guaranteed to be large enough to hold int values. The compiler is free to choose the actual type used based on the enumeration constants defined so it can choose a smaller type if it can represent the values you define. If you need enumeration constants that don't fit into an int you will need to use compiler-specific extensions to do so.

Preprocessor: –

[illegible]

#pragma are all preprocessing directives. (A line containing only # is also a preprocessing directive, but it has no effect.)

Constant Vs Macro: –

Macro informs the preprocessor to do a textual substitution and constant declare as a c variable. Pointer can be used for constant but not for macros. Macros are local only and constant can be access globally. Macros can use for textual substitution but constant not.

NOTE:-

If the scope of 'constant' is small or isolated prefer a const variable.

If the constant is used to specify the size for an array in automatic storage, choose a macros.

Where there is need of textual substitution use macros.

Macro Vs Function: –

Macro is preprocessed and function is compiled. No type checking in macro but in function is type checking. By using macros code length increases but using function remains same. By using macros speed of execution is faster but using function speed gets slow. Before compilation macro name is replaced by macro value and in function during function call control transfer takes place. Macro do not extended beyond one line (to extend beyond one line we have to use / at end of line) and function can extend n no of lines. Useful where small code appear many time and function is useful where large code appear many times. Macro cannot return but function can return.

NOTE: – macro are error-prone because they rely on textual substitution and do not perform type checking ForEX #define SQUARE (X) X*X now

when i call like this `SQUARE (1+2)` then result is 5 that is wrong. EX. `#define SWAP(X,Y) (T=X; X=Y; Y=T;)` now if when we will call it like this then `if(X<Y) SWAP(X,Y)` then only `T=X;` will execute that is wrong.

Function Vs Inline Function: –

Inline function get copied in the code where ever they called, so during function call function overhead (push & pop of stack is not there so it saves time and improve execution speed.) Is not there, it causes speedup of the program. Inline function increase the code size. Function that called frequently and have small size should defined as an inline function. Function defined without inline has function overhead during function call, it causes slow execution of code as compare to inline. Normal function decrease the size of code.

Optimization: –

Optimization is process of transforming a piece of code to make more efficient without changing its o/p or side effects. The only difference visible to code users should be that it's run faster and/or consume less memory.

Volatile: –

Volatile tells the compiler not to optimize anything that has to do with the volatile variable. (SYNTAX `VOLATILE INT I=0;`)

If any variable that is not define with volatile keyword in code, and during optimization, if compiler found that value of that variable is not getting change throughout the code then compiler treat that variable as constant and optimize the code related to that variable. If variable define with volatile keyword compiler does not anything with that variable.

So if any variable value is getting change externally that variable should define with volatile keyword.

Typedef: –

To assign alternative name to existing type. (SYNTAXà `TYPEDEF INT NUMERIC; NUMERIC I=0` 😊)

Masking: –

A mask is data that is used for [bitwise operations](#), particularly in a [bit field](#). Using a mask, multiple bits in a [byte](#), [nibble](#), [word](#) (etc.) Can be set either on, off or inverted from on to off (or vice versa) in a single bitwise operation.

Endianness: –

It is a term that describe the order/sequence in which bytes of data stores in computer memory.

Big Endian & Little Endianà

Big-endian is an order/sequence in which the “big end” (most significant value in the sequence) is stores first (at the lowest storage address).

Little-endian is an order in which the “little end” (least significant value in the sequence) is stores first.

For example: suppose we have data bytes 0x4f52 needs to read/write in a computer. In a big-endian computer, if 4f is stored at storage address 1000, for example, 52 will be at address 1001. In a little-endian system, it would be stored as 524f. 52 at address 1000, 4f at 1001.

How to check endianness?

<pre> int i=0x01; char *ptr; ptr= (char *) &i; if(*ptr==0) printf("big endian");, else printf("lit endian");, </pre>	<pre> union checkendianess { Int i; char c; } Checkendianess.i = 0x0001; If(Checkendianess.c = 0x00) printf("big endian");, else printf("lit endian");, </pre>
---	---

Callback: –

A callback is a piece of executable code that is passed as an argument to other code and which is expected to call back the argument at some convenient time. The invocation may be synchronous (blocking) and asynchronous (deferred).

Memory Leak: –

A memory leak occurs when a program incorrectly manages memory location. **Ex.** During dynamic memory allocation developer should take care after using dynamic allocated function it should be free.

Linkage: –

EXTERNAL LINKAGE → global variable and function have ext. Linkage.

INTERNAL LINKAGE → static variable and function have int. Linkage.

NO LINKAGE → local variable, function parameter, typedef names, label names, structure tag name have no linkage.

Diff B/W Definition & Declaration: –

Definition means where a variable or function is defined in reality and actual memory is allocated for variable or function. Declaration means just giving a reference of a variable or function.

Diff B/W Reentrant & Recursive: –

Reentrant function is a function which guaranteed that it can be work well under multithread environment i.e. if function is access by one thread, another thread can call it, means there is a separate execution stack and handling for each.

If one thread tries to change the value of shared data at the same time as another tries to read the value, the result is not predictable. We call it as a race condition.

So function should not contain any static or shared variable which can harm or disturb the execution. Mean function which can be called by thread, while

running from another thread safely and properly.

A reentrant function shall satisfy the following condition:-

Should not call another non reentrant function.

Should not access static life time variable (static/extern).

Should not include self-modifying code.

A recursive function is one which calls itself.

Ex. Recursion and re-entrant

```
int rec(int x)
```

```
{
```

```
if(x==1)
```

```
return 0;
```

```
else
```

```
{
```

```
x--;
```

```
return x*rec(x);
```

```
}
```

```
}
```

```
int main()
```

```
{
```

```
int c, n, fact = 1;
```

```
printf("Enter a number\n");
```

```
scanf("%d", &n);
```

```
for (c = 1; c <= n; c++)
```

```
fact = fact * c;
```

```
printf("Fact of %d = %d\n", n, fact);
```

```
return 0;
```

```
}
```

```
int t; void swap(int *x, int *y){ t =
*x; *x = *y; /* hardware interrupt
might invoke isr() here! */ *y =
t;} void isr(){ int x = 1, y = 2;
swap(&x, &y);}
```

```
int t; void swap(int *x, int *y){ int s;
s = t; /*save global variable*/ t = *x;
*x = *y; /* hardware interrupt
might invoke isr() here! */ *y = t; t
= s; /*restore global variable*/} void
isr(){ int x = 1, y = 2; swap(&x, &y);}
```

Diff B/W Pass by Value & Pass by Reference: –

In pass by value, value of a variable is passed. Change made to the formal will not affect to actual parameter. Different memory location will be created for both the variable. In pass by reference address of a variable is passed to function. Whatever changes made to the formal parameter will not affect actual parameter. Same memory location is used for both variables. It is required when required more than one value.

Diff B/W Normal & Cross Compiler: –

In normal compiler user can run the program and can see the output on the same machine at which user built the program. Ex. Dev C++. In cross compiler user can run the program and cannot see the output on the same machine at which user built the program. To see the output of the program user needs to use some other machine. Ex. Keil.

Correlation B/W Array & Pointer: –

An array can be thought of as a constant pointer. The array itself can be treated as a pointer and used in pointer arithmetic. When a pointer points to the beginning of an array, adding an offset to the pointer indicates which element of array should be referenced and offset value is identical to the array subscript. This is referred to as pointer/offset notation. Pointer can be

Overloading: –

In c++ we can give special meaning to operator when they are used with user-defined classes. This is called operator overloading. Operator overloading is generally defined by the language, the programmer or both. C does not support operator overloading.

Hashing: –

Hashing is a process of storing large amount of data and retrieving element from it in optimal time. Hashing is the solution when we want to search an element from large collection of data. Two important thing of hashing is hashing table & hashing function.

LINKED LIST:

Data structure in c. Dynamic data structure whose length can be increased or decreased at run time. Linked list basically consists of memory blocks that are located at random memory locations. Usually a block in a linked list is represented through a structure like this:

```
STRUCT TEST_STRUCT {INT VAL; STRUCT TEST_STRUCT * NEXT};
```

This structure contains an integer 'val' and a pointer to a structure of same type. The integer 'val' can be any value (depending upon the data that the linked list is holding) while the pointer 'next' contains the address of next block of this linked list. So linked list traversal is made possible through these 'next' pointers that contain address of the next node. The 'next' pointer of the last node (or for a single node linked list) would contain a null.

Node creation a node is created by allocating memory to a structure:

```
STRUCT TEST_STRUCT *PTR = (STRUCT TEST_STRUCT*) MALLOC
```

```
(sizeof (STRUCT TEST_STRUCT));
```

Assigning nodeà once a node is created, then it can be assigned the value (that it is created to hold) and its next pointer is assigned the address of next node. If no next node exists (or if it's the last node) then as already discussed, a null is assigned.

```
ptr->val = 6; ptr->next = NULL;
```

Searching nodeàone just needs to start with the first node and then compare the value which is being searched with the value contained in this node. If the value does not match then through the 'next' pointer (which contains the address of next node) the next node is accessed and same value comparison is done there. The search goes on until last node is accessed or node is found whose value is equal to the value being searched.

Deletion of node A node is deleted by first finding it in the linked list and then calling free () on the pointer containing its address. If the deleted node is any node other than the first and last node then the 'next' pointer of the node previous to the deleted node needs to be pointed to the address of the node that is just after the deleted node. It's just like if a person breaks away from a human chain then the two persons (between whom the person was) needs to join hand together to maintain the chain.

Diff B/W Array & Linked List: –

1. An array is a static data structure. This means the length of array cannot be altered at run time. While, a linked list is a dynamic data structure and can altered at runtime.
2. In an array, all the elements are kept at consecutive memory locations while in a linked list the elements (or nodes) may be kept at any location but still connected to each other.
3. Linked lists are preferred mostly when you don't know the volume of data

4. For example, in an employee management system, one cannot use arrays as they are of fixed length while any number of new employees can join. In scenarios like these, linked lists (or other dynamic data structures) are used as their capacity can be increased (or decreased) at run time (whenever required).

OPTIMIZATION OF C CODE/PROGRAM:

1. Use unsigned int instead of int if u know value will not go negative.
2. Use shift operations >> and << instead of integer multiplication and division, where possible.
3. Use bitwise operator, where possible.
4. Use the prefix operator (++obj) instead of the postfix operator (obj++), where possible.
5. Try to avoid casting where possible.
6. If you do not need a return value from a function, do not define one.
7. Reduce the number of function parameters & local variables.
8. Jumps/branches are expensive. Minimize their use whenever possible.
9. Loop unrolling: avoid calculation inside loop, where possible. Avoid small loops.
10. Fast mathematics: avoid unnecessary division. Try to use bitwise operator for * & /.
11. Try to write simplified expression.
12. Prefer int instead of char.
13. Better way of calling function: reduce no of parameter or argument for a function. Pass structure & big array to function by reference.
14. Use of register variable: if any variable called several times e.g. in loop, try to use register class for it.
15. With switch: write most frequently executed cases on top.
16. Try to use array instead of dynamic memory allocation.
17. If function will not be used global, it has to be declare static otherwise

compiler will assume that it is global and take extra effort for its external

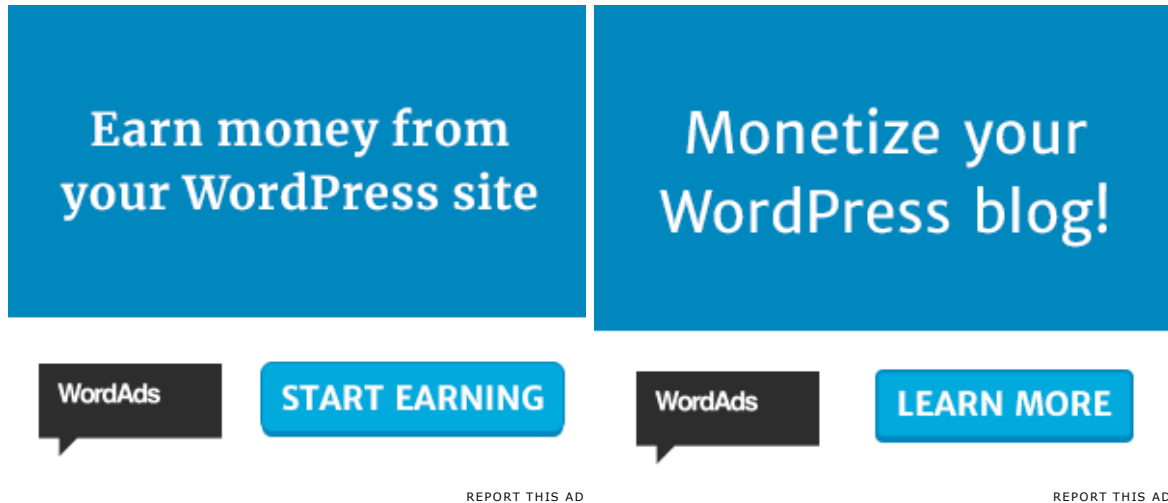
linkage.

18. Prefer integer comparison than other data type.

19. Avoid string comparison.

20. Declare variable depending on its size order always largest to smaller.

Advertisements



The advertisement banner consists of two blue rectangular boxes side-by-side. The left box contains the text "Earn money from your WordPress site" in white. The right box contains the text "Monetize your WordPress blog!" in white. Below the left box is a black speech bubble containing the text "WordAds". Below the right box is a black speech bubble containing the text "WordAds". Between the two boxes are two blue buttons: "START EARNING" on the left and "LEARN MORE" on the right. Below each "WordAds" speech bubble is a small link that says "REPORT THIS AD".

Share this:



Be the first to like this.

Automotive & Embedded Info / Powered by WordPress.com.

