

# **C Programming Tips for Advanced Platform Independent Programming**

**- Tushar J. Anjaria**

C provides a rope – Either you can Swing or Hang!

This is a very common Mantra of C programmers!  
But they all know that C programming creates wonders with proper understanding of advanced technics.

I was an avid C programmer in late 1980s and early 1990s. I have written thousands of lines of C code. Those were days of limited memory availability – only 640K and with extended memory 1024K on DOS!! Software programs were developed to run on multiple platforms – OS like DOS, UNIX, WINDOWS and on various machines. C was at its best for memory optimization and platform portability.

I learned many aspects of C for platform portability and noted down points as and when I faced and solved issues. Then my focus was changed on other things and my C programming is stopped since 1994. Recently I found my hand written points so thought to share with advanced C programmers. As I have been away from C since long, I am not sure how many of these tips are still valid today in the latest versions. But this may help to know more on platform independent aspects of C.

Happy C Programming!

- Tushar J. Anjaria  
Ahmedabad

01/01/2019

Email: advait\_sys@hotmail.com  
Cell: +91 9825159024

**Here I have given**

- 1. Tips for Portability**
- 2. Some Interesting Points of C Programming**

## Tips for Portability

- Use #define instead of hardcoded items
- Avoid machine dependent features like segmentation, word length, byte order within words, bit field's evaluation orders etc
- Avoid some compiler specific features for file handling, memory management, file's pathname etc
- Use typedef as much as possible, specially for short, int, long so we can change it accordingly
- Use Sizeof () instead of directly using char=1, int=2, long=4 etc
- a [i] = i++; is non portable operation.
- Find what is the maximum number a machine can support by portable way like

```
#define MAXPOS ((int)((((unsigned)-1)>>1))
```

to find maximum positive integer value supported by machine

- The unions are portable only if they are used to place different data in the same place at different times.

- Any program that makes use of knowledge of the internal byte order in a word is not portable.
- For portability, no bit field should exceed 16 bits.
- For pointer assignments, don't use non portable features like assigning pointers of different types.
- Large programs or programs that require large data areas may have portability problems on small machines.
- The only character set requirements are that all chars must fit in the char data type and that all chars must have positive values.
- Use `if (isupper(b))` in place of `if ( (B > 'A') && (B < 'Z'))`
- On different compilers signed, unsigned char may have sign extension problems.
- The number and type of register variables in a function depend on the machine hardware and the compiler.
- When char is compared with int, it can have problem. For Type conversion, use type casting property. For `getchar()` etc, use int instead of char.

- Don't use system calls and system library functions specific to some OS
- Data files are non portable on different machines because structures, unions, and arrays have varying internal layout and padding requirements on different machines. The only way to achieve data file portability is to write and read data files as one dimensional character arrays.
- Pointers to an object of given type may be assigned (or converted) to a pointer to an object of smaller size and back again without change ,i.e. a pointer-to-long can be assigned to a pointer-to-char and then back like:

```
Long *l;  
Char *c;
```

```
c = l;  
l = c;
```

- Don't use compiler dependent features of C, i.e. the order of evaluation of function arguments and order of evaluation of operands are unspecified.

- Don't use machine dependent features.  
Don't depend on machine's word size. The size of int depends on the machine's word size which varies on different machines. So if you are not sure of the result of an integer operation, then use long to avoid potential overflow problems and use short.
- floating point numbers have different internal representations on different machines so results may differ on different machines. So try to avoid this dependency.
- If you convert a pointer of one type to a pointer of another type, your program may cause an addressing exception when the converted pointer is dereferenced. This is caused by restrictions on machine alignment. Use library function malloc to get suitable aligned pointer.
- Be careful for Signed pointer comparisons.
- Pointer arithmetic may cause overflow or underflow.
- Use values.h which contains system wide hardware constants
- Use header files to isolate data and environment dependent data such as filenames or options. Place such things which vary across the systems in the header file so we can easily modify them.

- Be sure to supply to functions correct number of arguments having the right type.
- Call the system via standard library functions when possible instead of using your own system calls.
- To enhance the maintainability and portability of program, keep all external variable definitions of a program in their own source file.
- The maximum number of characters for external variables and functions depends on the operating system environment.
- Binary data files are inherently non portable because different machines use different internal representations of data objects (because of byte order etc).
- On some compiler, the large functions can give segmentation violations.
- To avoid segmentation violations of float, doubles, long etc, use `char *` for type cast and then do whatever operation is required (initialization etc) using `char *`
- For some items, padding may be required because of difference in storage alignment supported by different machines.



- Conversion of Signed (to fro) Unsigned is machine dependent.
- For portability, specify signed or unsigned if non character data is to be stored in char variable.
- PDP 11 and M68000 require that data types longer than one byte be aligned on even boundaries. Others like 8086s and VAX-11 have no such restrictions but even with these machines, most compilers generate code that aligns words, structures, arrays and long words on even addresses or log addresses.

## Some Interesting Points of C Programming

- In C, arguments are passed to function as Call by Value. The called function gets only temporary copy of argument values (on stack) and cannot change them.
- In C, pointers can be used for Call by reference.
- For Array arguments, only location of the beginning of array is passed. Elements are not copied.
- Declaration doesn't allocate storage for variable but only the nature of variable is stated. Definition actually assigns storage to the variable.
- If the external definition occurs before its use in file, then extern declaration is not required.
- The % operator cannot be applied on float or double.
- $X * Y + 1$  means  $X * (Y + 1)$

- If a function returns non integer values, then it must be declared before using it (which type it returns). Otherwise, compiler can think, it returns int and so junk can come. If calling a function and the called function are in the same file, compiler may detect this but if they are in different files, then it may fail.
- External variables' advantages – Scope is lifetime, Initialized, Saves argument lists.
- Static variables provide permanent and private storage.
- A pointer is a variable that contains the address of another variable. The unary operator and gives the address of an object.
- `*px +=1;` and `(*px)++;` -> both are same.
- If `int a [10];` then

`int *pa;`  
`pa = &a[0];` or `pa = a;`  
`*(pa+i)` or `a[i]` are same.

An array and index expression can be written as a pointer and object and vice versa. However, `a = pa;` or `a++;` or `pa = &a;` are illegal.

- `f(a+2)` or `f (&a[2])` both pass the address of the 2<sup>nd</sup> element `a[2]`

- `p+=i` increments `p` to point `i` element beyond where it currently is.
- Pointers cannot be used in addition, multiplication, division or float, double cannot be added to them.
- `*++p`; increments `p` before fetching whatever it points to.
- `*p++`; first fetches whatever it points to, then increments `p`.
- `int *a[13]`; is array of 13 pointers to integer.  
`Int (*a)[13]`; is pointer to an array of 13 integers.
- `char *line[100]`; is an array of 100 elements, each element of which is a pointer to a char. `line[i]` is a char pointer and `*line[i]` accesses a character.
- `int a[10][10]`;  
`int *b[10]`;

The usage of `a` and `b` may be similar in that `a[5][5]` and `b[5][5]` are both legal references to a single int. But array `a` is the true array. All 100 storage cells have been allocated and for `b`, the declaration allocates only 10 pointers – each must be set to point to an array of integers. Assuming that each does point to a ten element array, there will be

100 storage cells set aside, plus 10 cells for the pointers. Thus the array of pointers uses more space and may require an explicit initialization. But it has two advantages: (1) accessing an element is done by indirection through a pointer rather than by multiplication and addition (2) Rows of the array may be of different length, i.e. each element of `b` may point to different length elements.

- `*++argv` is a pointer to an argument string. `(*++argv)[0]` is its first character. This is different from `*++(argv[0])`
- `int (*f1)()` is a pointer to a function returning an `int`
- `int *f1();` is a function returning pointer to an `int`. The use of `f1` is like `if ((*f1()) ...`
- Structures cannot be assigned or copied as a unit and they cannot be passed or returned from functions. Pointers to structure can be used for this.
- `p -> y` and `(*p).y` are same.
- `++p -> x` and `++(p -> x)` increment `x` not `p`  
  
`(++p) -> x` increments `p` before accessing `x`  
`(p++) -> x` increments `p` after accessing `x`  
`*p -> y` fetches whatever `y` points to

`*p -> y++` increments `y` after fetching whatever it points to  
`(*p -> y)++` increments whatever `y` points to  
`*p++ -> y` increments `p` after accessing whatever `y` points to.

- The `&` operator only applies to objects in memory, variables and array elements. It cannot be applied to expressions, constants or register variables.
- If one is sure that the element exists, then it is possible to index backwards in an array. `p[-1]`, `p[-2]` so on are legal and refer to the elements that immediately precedes `p[0]`.
- When difference of the two pointers are taken (i.e. `p - q`), this difference may be too large to store in `int`. The header `<std define>` defines a type `ptrdiff_t` that is large enough to hold the signed difference of two pointer values. However the type `size_t` can be used to match the standard library version. `size_t` is the unsigned integer type returned by the `sizeof` operator.
- In `char *name[] = {"a", "bcd", "efghijk"};`, each element occupies space equal to the actual requirement, i.e. 1<sup>st</sup> element "a" occupies one byte, 2<sup>nd</sup> element "bcd" occupies three bytes, 3<sup>rd</sup> element "efghijk" occupies seven bytes.

While in two dimensional array `char aname[][10] = {"a", "bcd", "efghijk"...}`, each element occupies 10 bytes irrespective of the actual requirement.

Thus two dimensional array occupies unnecessary space.

- Sometimes it may become necessary to pack several objects into a single machine word. A field is a set of adjacent bits within a single int. A field may not overlap an int boundary; if the width would cause this to happen, the field is aligned to the next int boundary. Fields need not be named. Unnamed fields are used for padding. The special width 0 may be used to force alignment at the next int boundary. Fields should not be used for portability.