TATA ELXSI
engineering creativity

TATA

C Programming

Presenter : Learning & Development Team

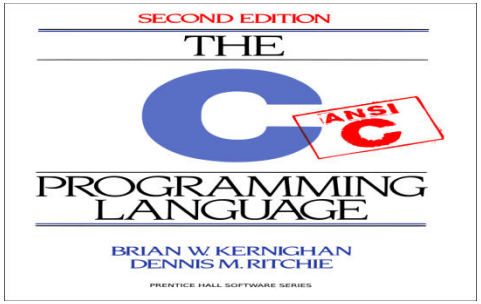
Day – 4

Disclaimer

- This material is developed for in-house training at TATA ELXSI.
- The training material therefore was solely developed for educational purposes. Currently, commercialization of the prototype does not exist, nor is the prototype available for general usage. TATA ELXSI, its staff, its students or any other participants can use it as a part of training. This material should not be found on any website.
- For the preparation of this material we have referred from the below mentioned links or books. Excerpts of these material has been taken where there is no copy right infringements. Some examples and concepts have been sourced from the below links and are open source material

  http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

ANSI C K&R



Day – 4 Agenda

- Pointer to array
- Two dimensional arrays and pointers
- Dynamic Address Allocation
- Address space allocation
- malloc
- calloc
- realloc
- free
- Memory leak

Pointers and Strings

```
int main(void)
{
    char strA[]=“Hello”;
    char strB[]=“World”;
    char *pA; /* a pointer to type character */
    char *pB; /* another pointer to type character */
    puts(strA); /* show string A */
    pA = strA; /* point pA at string A */
    puts(pA); /* show what pA is pointing to */
    pB = strB; /* point pB at string B */
    putchar('\n'); /* move down one line on the screen */
    while(*pA != '\0') /* line A (see text) */
    {
          *pB++ = *pA++; /* line B (see text) */
     }
     *pB = '\0'; /* line C (see text) */
     puts(strB); /* show strB on screen */
     return 0;
}
```

Pointers and function arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function.

In the swap(a, b); function call by value , swap can't affect the argument a and b in the routine that called it.

The way to obtain the desired effect is for the calling program to pass pointer to the values to be changed :

Swap (&a, &b);

```
void swap(int *px, int *py)
{
    int temp = *px;
    *px = *py;
    *py = temp
}
```

Pointers and functions

```
main()
{
      char str[25] = “hello world”;
      printf(“%d\n”,xstrlen(str));
}

/* xstrlen: return length of string s */
int xstrlen(char *s)
{
       int n;
       for (n = 0; *s != '\0', s++)
              n++;
      return n;
 }
```

The first function is strcpy(s,t), which copies the string t to the string s.

It would be nice just to say s=t but this copies the pointer, not the characters.

```
/* strcpy: copy t to s; array subscript version */

void xstrcpy(char *s, char *t)
{
      int i;
      i = 0;
      while ((s[i] = t[i]) != '\0')
            i++;
 }
```

For contrast, here is a version of strcpy with pointers:

```
 /* strcpy: copy t to s; pointer version */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0')   {
          s++;       t++;
    }
}
/* strcpy: copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
     while ((*s++ = *t++) != '\0') ;
}
```

As the final abbreviation, observe that a comparison against '\0' is redundant, since the question is merely whether the expression is zero.

So the function would likely be written as

```
/* strcpy: copy t to s; pointer version 3 */
void strcpy(char *s, char *t) {
      while (*s++ = *t++) ;
 }
```

```
    /* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    int i;
    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
        return s[i] - t[i];
}
```

The pointer version of strcmp:

```
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
         return *s - *t;
 }
```

Initialization of Pointer Arrays

Consider the problem of writing a function month_name(n), which returns a pointer to a character string containing the name of the n-th month.

This is an ideal application for an internal static array. month_name contains a private array of character strings, and returns a pointer to the proper one when called.

This section shows how that array of names is initialized.

The syntax is similar to previous initializations:

/* month_name: return name of n-th month */

// Function returning pointer

```
char *month_name(int n)
{
    static char *name[] = { "Illegal month", "January", "February",
                            "March", "April", "May", "June", "July",
                            "August", "September", "October",
                            "November", "December" };
    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

Array of pointers

Although we have phrased this discussion in terms of integers, by far the most frequent use of arrays of pointers is to store character strings of diverse lengths, as in the function month_name.

Compare the declaration and picture for an array of pointers:

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
```

with those for a two-dimensional array:

```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```

Dynamic mem allocation

Malloc library function

malloc calls upon the operating system to obtain memory as necessary.

Syntax:

void *malloc(size_t size)

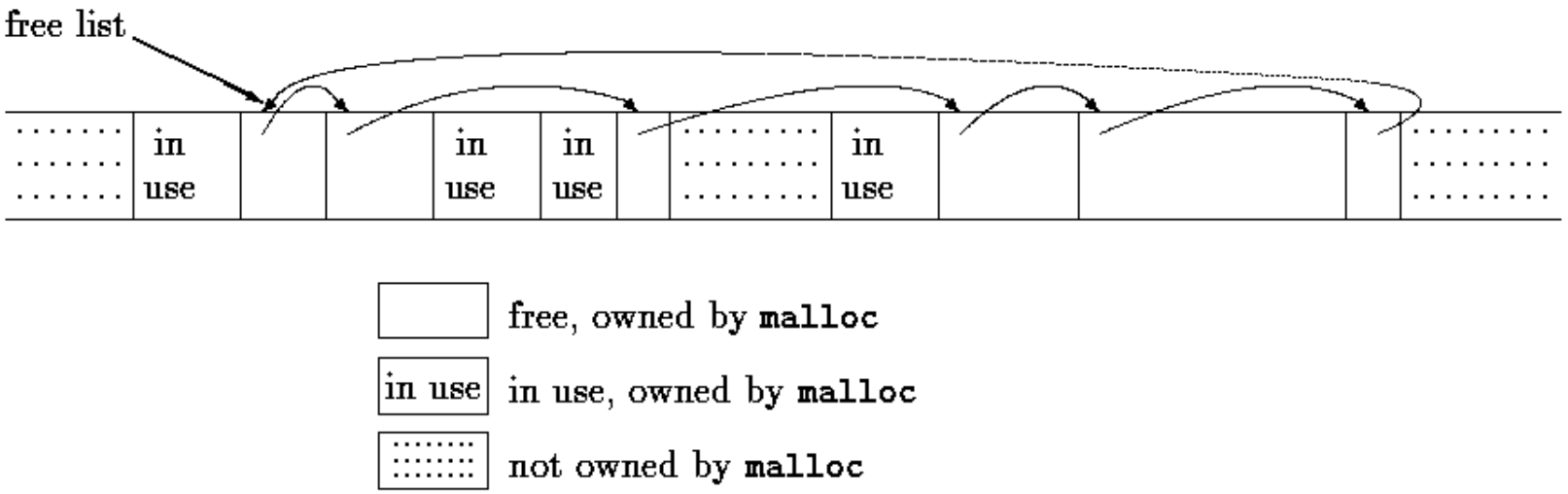
malloc returns a pointer to space(memory) for an object of size size, or NULL if the request cannot be satisfied.

The space is uninitialized.

Rather than allocating from a compiled-in fixed-size array, malloc will request space from the operating system as needed.

Thus its free storage is kept as a list of free blocks. Each block contains a size, a pointer to the next block, and the space itself.

A Storage Allocator



A Storage Allocator

When a request is made, the free list is scanned until a big-enough block is found.

This algorithm is called ``first fit,'' by contrast with ``best fit,'' which looks for the smallest block that will satisfy the request.

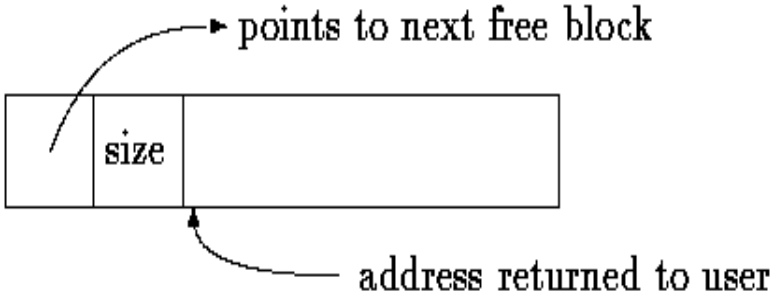
If the block is exactly the size requested it is unlinked from the list and returned to the user.

If the block is too big, it is split, and the proper amount is returned to the user while the residue remains on the free list.

If no big-enough block is found, another large chunk is obtained by the operating system and linked into the free list.

The Align field is never used; it just forces each header to be aligned on a worst-case boundary.

In malloc, the requested size in characters is rounded up to the proper number of header-sized units; the block that will be allocated contains one more unit, for the header itself, and this is the value recorded in the size field of the header.



A block returned by **malloc**

The pointer returned by malloc points at the free space, not at the header itself.

The user can do anything with the space requested, but if anything is written outside of the allocated space the list is likely to be scrambled.

```
/* Program to illustrate malloc */

#include<stdio.h>
#define MAX 1000
unsigned int nmax;
unsigned int *nums;
main()
{
unsigned int ctr, n;
void print_nums(unsigned int);
printf("Enter Max no. you want to input");
if(scanf("%d",&nmax)!=1)
        exit(1);
if(nmax <1 || nmax > MAX)    {
        fprintf(stderr,"Enter nmax properly");
        exit(2);
}
```

```
nums= (unsigned int *)malloc(nmax * sizeof (unsigned int));
if(nums == NULL) {
        perror(“malloc:”);
        exit(1);
}
ctr=0;
while(ctr<nmax)
{
        printf("Enter %dth number", ctr);
        if(scanf("%d",nums+ctr)!=1)break;
        ctr++;
}
n=ctr;
print_nums(n);
free(nums);             //Freeing allocated memory
nums = NULL;
}  //closing main
```

```
void print_nums(unsigned int N)
{
unsigned int i ; i=0;
while(i<N)
{
        printf("%dth Number is %d\n",i,*(nums+i));
        i++;
   }
}//closing function print_nums
```

realloc

realloc

realloc changes the size of the object pointed to by p to size.

Systax:

void *realloc(void *p, size_t size)

The contents will be unchanged up to the minimum of the old and new sizes.

If the new size is larger, the new space is uninitialized.

realloc returns a pointer to the new space, or NULL if the request cannot be satisfied, in which case *p is unchanged.

realloc(void * p, NULL);: Is as good as calling free() function.

```
/* Program to illustrate realloc function */
 #include<stdio.h>
int *nums;
main()
{
int next, n=0;
void print_nums(unsigned int );
nums=(int *)malloc(sizeof(int));
if(nums == NULL) {
        perror(“malloc:”);
        exit(1);
}
while(1){
        printf("Enter %dth number\n",n+1);
        if(scanf("%d",nums+n)!=1)      break;
        nums=(int *)realloc(nums,(++n+1)*sizeof *nums);// Check for exception conditions
}
print_nums(n);
free(nums);
Nums= NULL ;
```

```
void print_nums(unsigned int N)
{
unsigned int i ; i=0;
while(i<N)   {
        printf("Number is %d\n", *(nums+i));
        i++;
}
}//closing function print_nums
```

Passing Arrays to Functions

This relationship comes into play when an array has to be passed as an argument to a function.

The only way to pass an array to a function is by means of a pointer.

If a function is written which takes an array as an argument, a function that can handle arrays of different sizes.

How does the function know the size of the array whose address it was passed? Remember, the array passed to a function is a pointer to the first array element.

It could be the first of 10 elements or the first of 10,000.

There are two methods of letting a function know an array's size.

- void function(int array[10]);
- void function(int array[], int size);

Calloc function

Syntax:

void *calloc(size_t nobj, size_t size)

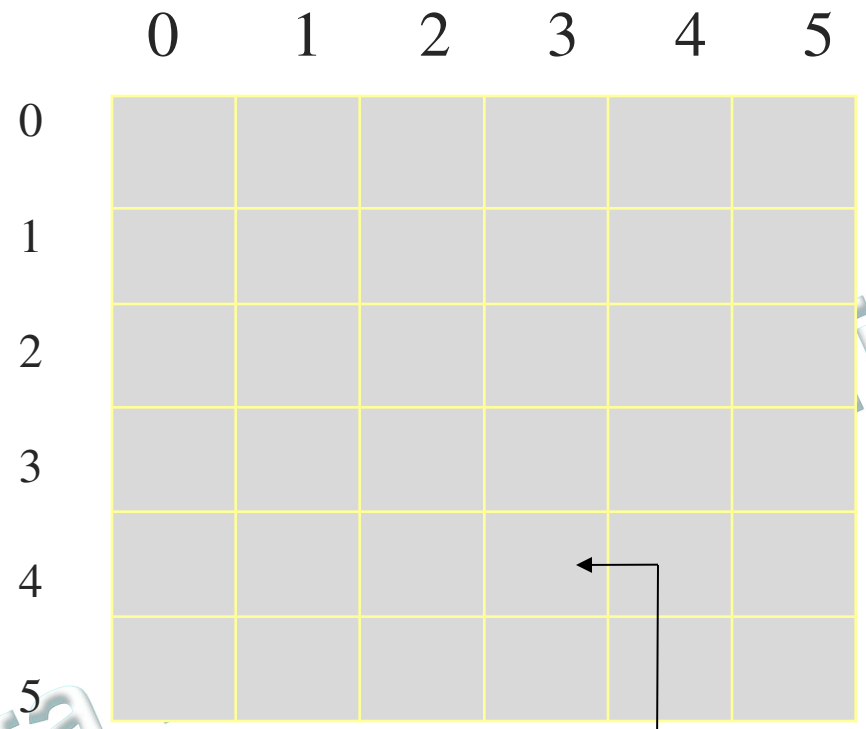
calloc returns a pointer to space for an array of nobj objects, each of size size, or NULL if the request cannot be satisfied. The space is initialized to zero bytes.

```
 /* using calloc allocating memory. sorting using selection sort algorithm
 in descending order.  Passing pointer to a function */
#include<stdio.h>
void selection(int *,int );
main()
{
int n,i;
int    *iptr ;

printf("enter limit of an array");
scanf("%d",&n);
 iptr=(int  *) calloc(n,sizeof(n)); //check for exception condition
for(i=0;i<n;i++) {
        scanf("%d",(p+i));
}
selection(iptr , n);
free(ptr);
}
```

```
void selection (int *p,int n)
{
int i,j,temp;
for(i=0;i<n;i++)
        for(j=i+1;j<n;j++)
                if(*(p+i)<*(p+j))       {
                        temp=*(p+i);
                        *(p+i)=*(p+j);
                        *(p+j)=temp;
                }
 for(i = 0;i < n;i ++)
        printf("\n%d",*(p+i));
}
```

Pointers and two dimensional arrays



p **pointer to first row**
p+i **pointer to ith row**
***(p+i)** **pointer to first element in the ith row**
***(p+i)+j** **pointer to jth eleement in the ith row**
***(*(p+i)+j)** **value stored in the cell**

Command-line Arguments

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing.

When main is called, it is called with two arguments.

The first (conventionally called argc, for argument count) is the number of command-line arguments the program was invoked with;

the second (argv, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string.

The simplest illustration is the program echo, which echoes its command-line arguments on a single line, separated by blanks.

That is, the command

$ echo hello, world

prints the output : hello, world

By convention, argv[0] is the name by which the program was invoked, so argc is at least 1.

If argc is 1, there are no command-line arguments after the program name.

In the example above, argc is 3,

argv[0] : echo

argv[1] : hello,

argv[2] : world

The first optional argument is argv[1] and the last is argv[argc-1]; additionally,

the standard requires that argv[argc] be a null pointer.

The first version of echo treats argv as an array of character pointers:

```
#include <stdio.h>
/* echo command-line arguments; 1st version */
 main(int argc, char *argv[])
{
        int i;
       for (i = 1; i < argc; i++)
          printf("%s%s", argv[i], (i < argc-1) ? " " : "");
       printf("\n");
       return 0;
}
```

```
main()
{
int  *a[5],  *b[5]  ,  *c[5],   i , j;
for(i = 0; i < 5;i++)
{
        a[i] = (int *) malloc(sizeof(int)*5);
        b[i] = (int *) malloc(sizeof(int)*5);
        c[i] = (int *) malloc(sizeof(int)*5);
}
puts("Enter elements to mat1\n");
for(i = 0; i < 2;i++)
        for(j = 0; j < 2;j++)
                scanf("%d",*(a+i)+j);
        for(i = 0; i < 2;i++)
                for(j = 0; j < 2;j++)
                        scanf("%d",*(b+i)+j);
```

```
for(i = 0; i < 2;i++)
        for(j = 0; j < 2;j++)
                *(*(c+i)+j) = *(*(a+i)+j) + *(*(b+i)+j);
for(i = 0; i < 2;i++)
{
        for(j = 0; j < 2;j++)
                printf("%d\t",*(*(c+i)+j));
        puts("\n");
}
        for(i = 0; i < rows; i++){
            free(a[i]);
            free(b[i]);
            free(c[i]);
        }
}
```

Program to add matrices using double pointers

```
#include<stdio.h>
#include<stdlib.h>
main()
{
int **a, **b, **c;
int i, j, rows, cols;
        printf("Enter the Number of Rows for the Matrices : \n");
        scanf("%d",&rows);
        a = (int **)malloc(rows * sizeof(int *));
        b = (int **)malloc(rows * sizeof(int *));
        c = (int **)malloc(rows * sizeof(int *));

        printf("Enter the Number of Columns for the Matrices : \n");
        scanf("%d",&cols);
```

```
for(i = 0; i < rows ; i++){
                *(a+i) = (int *)malloc(cols * sizeof(int));
                *(b+i) = (int *)malloc(cols * sizeof(int));
                *(c+i) = (int *)malloc(cols * sizeof(int));
                }
        printf("Enter the Elements for the first matrix : \n");
for(i = 0; i < rows; i++){
                for (j = 0; j < cols; j++){
                scanf("%d", (*(a+i)+j));
                }
        }
```

```
printf("Enter the Elements for the Second matrix : \n");
        for(i = 0; i < rows ; i++){
                for(j = 0; j < cols ; j++){
                scanf("%d",(*(b+i)+j));
                }
        }
        for(i = 0; i < rows; i++){
                for(j = 0 ; j < cols; j++){
                        c[i][j]  = a[i][j] + b[i][j];
                        printf("%d\t", c[i][j]);
                }       printf("\n");
        }
```

```
    for(i = 0; i < rows; i++){
    free(a[i]);
            free(b[i]);
            free(c[i]);
    }
   free(a);
   free(b);
   free(c);
}
```

```
void  _matrix_get_alloc(int ***p,int rows,int cols);
void  _matrix_enter_elements(int **p,int rows,int cols);
void  _matrix_addition_result(int **p,int **q,int **r, int rows,int cols);
void  _matrix_display(int **r, int rows,int cols);

main()
{
int **a, **b, **c;
int i, j, rows, cols;

printf("Enter the Number of rows : ");     scanf("%d",&rows);
printf("Enter the Number of Columns : ");      scanf("%d", &cols);
_matrix_get_alloc(&a, rows, cols); //address of pointer
_matrix_get_alloc(&b, rows, cols);
    _matrix_get_alloc(&c, rows, cols);
```

```
_matrix_enter_elements(a, rows, cols);
_matrix_enter_elements(b, rows, cols);
_matrix_addition_result(a, b, c, rows, cols);
        _matrix_display(c,  rows, cols);
}

void  _matrix_get_alloc(int ***p, int rows, int cols){
        int i;
        *p = (int **)malloc(rows * sizeof(int *));
        for(i = 0; i < rows ; i++){
                *(*p+i) = (int *)malloc(cols * sizeof(int));
        }
}
```

```
void        _matrix_enter_elements(int **p,int rows,int cols){
int i ,j;
printf("Enter the Elements for the matrix %c : \n", p);
for(i = 0; i < rows; i++){
    for (j = 0; j < cols; j++){
        scanf("%d",(*(p+i)+j));
    }
}
}

void  _matrix_addition_result(int **p,int **q,int **r, int rows,int cols){
    int i, j;
    printf("\n");
    for(i = 0; i < rows; i++)
        for(j = 0 ; j < cols; j++)
            r[i][j]  = p[i][j] + q[i][j];
}
```

```
void  _matrix_display(int **r, int rows,int cols)
{
int i, j;
printf("\n");
for(i = 0; i < rows; i++){
        for(j = 0 ; j < cols; j++)
                printf("%d\t", r[i][j]);
        printf("\n");
}
}
```

Pointer constant

const int *p;

int const *p;

defines p as a pointer to a constant integer

Constant Pointer

int * const p;

defines a constant pointer to an integer

Pointer constant

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

main()
{
    const char *p;
    p = (char *) malloc(100);
    puts("Enetr a string");
    gets(p);

    strcat(p,"This is appended");  //This is wrong modifying pointer constant
    puts(p);
    free(p);
}
```

Constant Pointer

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

main()
{
     char * const       p = (char *) malloc(100);
     puts("Enetr a string");
     gets(p);
     free(p);

     p = (char *) malloc(100);   // This is error since p is constant
     strcpy(p,"This is copied");
     puts(p);
     free(p);
 }
```

Pointers to functions

A Pointer to a function can be defined to hold the address of a function,

The same can be used to invoke a function.

It is also possible to pass address of different functions at different times thus making the function more flexible and abstract

Declaring pointer to a function

Syntax:

returntype (* ptrtofn)(arguments if any);

Address of a function

The address of a function can be obtained by just specifying the name of the function without the trailing parenthesis.

Invoking a function using pointers

Syntax

(*ptrtofn)(arguments if any);

(ptrtofn)(arguments if any);

Pointers to functions

Once a pointer a function is defined, it matches with the return type and the argument list stated in the definition of the pointer to a function.

- For example

  int (*anyfn)(int,int)

- Now the variable anyfn can point to any function that takes two integer arguments and returns a single integer value.
- For example,

  int min(int a,int b)

  int max(int i,int j)

  int add(int x,int y)

Example:

```
long fact(long int num)
{
if(num==0)
        return 1;
else
        return num * fact(num-1);
}
main()
{
long int n, f1;
long int(*ptrfact)(long int);
ptrfact=fact;
printf("Enter the number whose factorial has to be found\n");
scanf("%ld",&n);
f1=(*ptrfact)(n);
printf("The factorial of %ld is %ld\n",n,f1);

printf("The factorial of %ld is %ld\n",n+1,ptrfact(n+1));
}
```

Pointers to functions

```
//Sending function as a parameter to function
main()
{
int m,n,high,low;
int (*ptrf)(int,int);
printf("Enter 2 integers\n");
scanf("%d %d",&m,&n);
high=select(large,m,n);
ptrf=small;
low=select(ptrf,m,n);
printf("large=%d\n",high);
printf("small=%d\n",low);
}
```

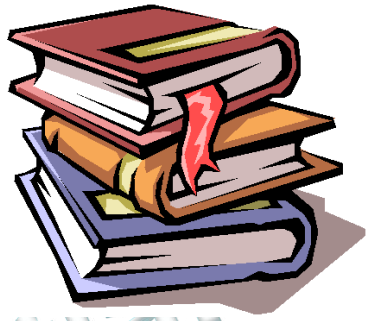
Pointers to functions

```
int small (int a, int b)
{
return a<b?a:b;
}

int large (int a, int b)
{
return a>b?a:b;
}

int select(int (*fn)(int,int),int x,int y)
{
int value=fn(x,y);
return value;
}
```

references



- The GNU C Reference Manual : https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

- The GNU C Reference Manual : http://www.cprogramming.com/

- C Programming Language

by Brian W. Kernighan , Dennis Ritchie

- C: The Complete Reference

by Herbert Schildt

Thank you

TATA ELXSI

ITPB Road Whitefield
Bangalore 560 048 India
Tel +91 80 2297 9123
Fax +91 80 2841 1474
e-mail info@tataelxsi.com

www.tataelxsi.com