WAP to print a string in reverse order using recursive function, in which string is passed as an argument.

```
# include <stdio.h>

/* Function to print reverse of the passed string */
void reverse(char *str)
{
  if(*str)
  {
     reverse(str+1);
     printf("%c", *str);
  }
}

/* Driver program to test above function */
int main()
{
  char a[] = "Geeks for Geeks";
  reverse(a);
  getchar();
  return 0;
}
```

WAP to reverse string Without using temporary variable in main itself

```
int main(){
   char str[50];
   char rev[50];
   int i=-1,j=0;

   printf("Enter any string : ");
   scanf("%s",str);

   while(str[++i]!='\0');

   while(i>=0)
    rev[j++] = str[--i];

   rev[j]='\0';

   printf("Reverse of string is : %s",rev);

   return 0;
```

```
}
```

WAP to find the length of the string using recursive function

```
int main(){
   char str[50];
   char rev[50];
   int i=-1,j=0;

   printf("Enter any string : ");
   scanf("%s",str);

   while(str[++i]!='\0');

   while(i>=0)
    rev[j++] = str[--i];

   rev[j]='\0';

   printf("Reverse of string is : %s",rev);

   return 0;
}
```

WAP to arrange entered string in alphabetical order.

Ex: i/p string = “WELCOME”, o/p = “CEELMOW”

WAP to remove repeated characters from a string.

Ex: i/p string = “hello world” , o/p = “helowrd”

WAP to find size of variable using bitwise operator and pointer

```
#include  <stdio.h>

int main()
{
    int i ;
    double *p=0;
```

```
    //    p = (char *)(&p +1) - (char *)&p;
          p++;
          printf("%d = ",p);

    }
```

WAP to convert integer into binary and print

```
int main(){
   long int decimalNumber,remainder,quotient;
   int binaryNumber[100],i=1,j;

   printf("Enter any decimal number: ");
   scanf("%ld",&decimalNumber);

   quotient = decimalNumber;

   while(quotient!=0){
       binaryNumber[i++]= quotient % 2;
       quotient = quotient / 2;
   }

   printf("Equivalent binary value of decimal number %d: ",decimalNumber);
   for(j = i -1 ;j> 0;j--)
       printf("%d",binaryNumber[j]);

   return 0;
}
```

WAP to set bits in between m,n positions in an integer.

WAP to check whether a number is divisible by2 and WAP to check whether a number is power of 2 using bit wise operators

If(num%2 ==0)

It is divisible by 2;

```
#include<stdio.h>
#define bool int

/* Function to check if x is power of 2*/
bool isPowerOfTwo(int n)
{
  if (n == 0)
    return 0;
  while (n != 1)
  {
    if (n%2 != 0)
      return 0;
    n = n/2;
  }
  return 1;
}

/*Driver program to test above function*/
int main()
{
  isPowerOfTwo(31)? printf("Yes\n"): printf("No\n");
```

```
 isPowerOfTwo(17)? printf("Yes\n"): printf("No\n");
 isPowerOfTwo(16)? printf("Yes\n"): printf("No\n");
 isPowerOfTwo(2)? printf("Yes\n"): printf("No\n");
 isPowerOfTwo(18)? printf("Yes\n"): printf("No\n");
 isPowerOfTwo(1)? printf("Yes\n"): printf("No\n");
 return 0;
}
```

WAP to add sum of elements of integer array by ignoring repetitive values.

Ex: int a[] = “1,4,6,7,23,45,2,67,8,3,1,4,9,20,45, ……”

0/p = 1+4+6+7+23+45+2+8+3+20

```
int main()
{
        int tot_arr=0,j=0;
        int a[] = {1,2,3,4,5,6,7,8,9,10,1,2,3};
        int size_of_arr = sizeof(a)/sizeof(*(a));
        printf("\n sizeofArr = %d",size_of_arr);
        for(int i = 0;i<size_of_arr;i++)
        {
                for(j = 0;j<i;j++)
                {
                        if(a[i]==a[j])
                        {
                                break;
                        }
                }
                if(i == j)
                {
                tot_arr += a[i];
                }
        }
        printf("\n tot_arr_add = %d",tot_arr);
}
```

What is a dangling pointer?

```
#include<stdlib.h>
{
   char *ptr = malloc(Constant_Value);
   .......
   .......
   .......
   free (ptr);      /* ptr now becomes a dangling pointer */
}
```

We have declared the character pointer in the first step. After execution of some statements we have de-allocated memory which is allocated previously for the pointer. pointer is still pointing such memory location

As soon as memory is de-allocated for pointer, pointer becomes dangling pointer

```
#include<stdlib.h>
{
   char *ptr = malloc(Constant_Value);
   .......
   .......
   .......
   free (ptr);  /* ptr now becomes a dangling pointer */
   ptr = NULL   /* ptr is no more dangling pointer */
}
```

After de-allocating memory, initialize pointer to NULL so that pointer will be no longer dangling. Assigning NULL value means pointer is not pointing to any memory location

What are execution steps of a C program?

- cpp hello.c -o hello.i" - Preprocessor preprocessing hello.c and saving output to hello.i.
- "cc1 hello.i -o hello.s" - Compiler compiling hello.i and saving output to hello.s.

- "as hello.s -o hello.o" - Assembler assembling hello.s and saving output to hello.o.
- "ld hello.o -o hello" - Linker linking hello.o and saving output to hello. (rellocatable functional code)
- "load hello" - Loader loading hello and running hello(executable a.out functional code + runtime code).

What are storage classes and explain each in detail?

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. These specifiers precede the type that they modify. There are the following storage classes, which can be used in a C Program

- auto
- register
- static
- extern

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main()
{
  while(count--)
  {
    func();
  }
  return 0;
}
```

```
/* function definition */
void func( void )
{
  static int i = 5; /* local static variable */
  i++;

  printf("i is %d and count is %d\n", i, count);
}
```

You may not understand this example at this time because I have used *function* and *global variables*, which I have not explained so far. So for now let us proceed even if you do not understand it completely. When the above code is compiled and executed, it produces the following result:

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

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

```
int count ;
extern void write_extern();

main()
{
   count = 5;
   write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void)
{
   printf("count is %d\n", count);
}
```

Here, *extern* keyword is being used to declare *count* in the second file where as it has its definition in the first file, main.c. Now, compile these two files as follows:

```
$gcc main.c support.c
```

This will produce **a.out** executable program, when this program is executed, it produces the following result:

```
5
```

Explain briefly about Dynamic memory allocation and function associated with it?

> The exact size of array is unknown untill the compile time,i.e., time when a compier compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()
The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.
Syntax of malloc()

```
ptr=(cast-type*)malloc(byte-size)
```

Here, *ptr* is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.
calloc()
The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.
Syntax of calloc()

```
ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of *n* elements. For example:

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()
Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.
syntax of free()

```
free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated.
Examples of calloc() and malloc()
Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
   if(ptr==NULL)
   {
       printf("Error! memory not allocated.");
       exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
       scanf("%d",ptr+i);
```

```
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
```

```
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

realloc()
If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().
Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, *ptr* is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *ptr,i,n1,n2;
    printf("Enter size of array: ");
    scanf("%d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i=0;i<n1;++i)
        printf("%u\t",ptr+i);
```

```
    printf("\nEnter new size of array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);
    for(i=0;i<n2;++i)
        printf("%u\t",ptr+i);
    return 0;
}
```

WAP to set, clear, toggle MSB of an integer

1480down voteaccepted

Setting a bit

Use the bitwise OR operator (|) to set a bit.

```
number |= 1 << x;
```

That will set bit `x`.

Clearing a bit

Use the bitwise AND operator (&) to clear a bit.

```
number &= ~(1 << x);
```

That will clear bit `x`. You must invert the bit string with the bitwise NOT operator (~), then AND it.

Toggling a bit

The XOR operator (^) can be used to toggle a bit.

```
number ^= 1 << x;
```

That will toggle bit `x`.

Checking a bit

You didn't ask for this but I might as well add it.

To check a bit, shift the number x to the right, then bitwise AND it:

```
bit = (number >> x) & 1;
```

That will put the value of bit `x` into the variable `bit`.

Changing the *n*th bit to *x*

Setting the `n`th bit to either `1` or `0` can be achieved with the following:

```
number ^= (-x ^ number) & (1 << n);
```

Bit `n` will be set if `x` is `1`, and cleared if `x` is `0`.

```
#include <stdio.h>
int main(int argc, char **argv) {
        int count = 0;
        int a = 0xFFFFFFFF;
        int temp = a;
        while (temp) {
                temp >>= 1;
                count++;
//              printf("\n c = %d", count);
        }
        a ^= 1 << (count - 1);
        printf("\na = %x ", a);
        return 0;
}
/* Function to get no of set bits in binary
   representation of passed binary no. */
int countSetBits(unsigned int n)
{
  unsigned int count = 0;
  while(n)
  {
    count += n & 1;
    n >>= 1;
  }
  return count;
}

/* Program to test function countSetBits */
int main()
{
```

```
    int i = 9;
    printf("%d", countSetBits(i));
    getchar();
    return 0;
}
```

WAP to set, clear, toggle nth bit in integer by using MACRO

```
#include <stdio.h>
#define SET_BIT(x,y)            (x |= (1<<y))
#define CLEAR_BIT(x,y)          (x &= ~(1<<y))
#define TOGGLE_BIT(x,y)         (x ^= (1<<y))
int main(int argc, char **argv) {
        int count = 0;
        int a = 0x7FFFFFF;
        int temp = a;
        while (temp) {
                temp >>= 1;
                count++;
//              printf("\n c = %d", count);
        }
//      a ^= 1 << (count - 1);
        a = SET_BIT(a, 7);
        printf("\na = %x ", a);
        a = CLEAR_BIT(a, 7);
        printf("\na = %x ", a);
        a = TOGGLE_BIT(a, 7);
        printf("\na = %x ", a);
        return 0;
    }
```

Explain following with example :

Volatile

Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time. The system always reads the current value of a volatile object from the memory location rather than keeping its value in temporary register at the point it is requested, even if a previous instruction asked for a value from the same object. So the simple question is, how can value of a variable change in such a way that compiler cannot predict. Consider the following cases for answer to this question.

```
/* Compile code without optimization option */
#include <stdio.h>
int main(void)
{
   const int local = 10;
   int *ptr = (int*) &local;

   printf("Initial value of local : %d \n", local);

   *ptr = 100;

   printf("Modified value of local: %d \n", local);

   return 0;
}
```

Output:

```
Initial value of local : 10
Modified value of local: 100
```

```
int main(void)
{
   const int local = 10;
   int *ptr = (int*) &local;

   printf("Initial value of local : %d \n", local);

   *ptr = 100;

   printf("Modified value of local: %d \n", local);

   return 0;
}
```

Output:

```
Initial value of local : 10
Modified value of local: 10
```

Constant volatile

To read the hardware registers and code cannot modify the variables.

```
int main(void)
{
   const volatile int local = 10;
   int *ptr = (int*) &local;

   printf("Initial value of local : %d \n", local);

   *ptr = 100;

   printf("Modified value of local: %d \n", local);

   return 0;
}
```

Output:

```
Initial value of local : 10
 Modified value of local: 100
```

Macro

A **macro** is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the **macro**.

```
#define max(A,B) ( (A) > (B) ? (A):(B))
```

Inline functions

In the C and C++ programming languages, an **inline function** is one qualified with the keyword inline; this serves two purposes. Firstly, it serves as a compiler directive, which suggests (but does not require) that the compiler substitute the body of the function inline by performing inline expansion, i.e.: by inserting the function code at the address of each function call, thereby saving the overhead of a function call. In this respect it is analogous to the register storage class specifier, which similarly provides an optimization hint

```
inline int max(int a, int b) {
  return a > b ? a : b;
}
```

Explain function pointer with an example

would have not been valid.

A constant pointer is declared as :

<type-of-pointer> *const <name-of-pointer>

For example :

```
#include<stdio.h>

int main(void)
{
   char ch = 'c';
   char c = 'a';

   char *const ptr = &ch; // A constant pointer
   ptr = &c; // Trying to assign new address to a constant pointer. WRONG!!!!

   return 0;
}
```

A pointer to a constant is declared as :

const <type-of-pointer> *<name-of-pointer>;

For example :

```
#include<stdio.h>

int main(void)
{
   char ch = 'c';
   const char *ptr = &ch; // A constant pointer 'ptr' pointing to 'ch'
   *ptr = 'a';// WRONG!!! Cannot change the value at address pointed by 'ptr'.

   return 0;
}
```

A function pointer can be declared as :

<return type of function> (*<name of pointer>) (type of function arguments)

For example :

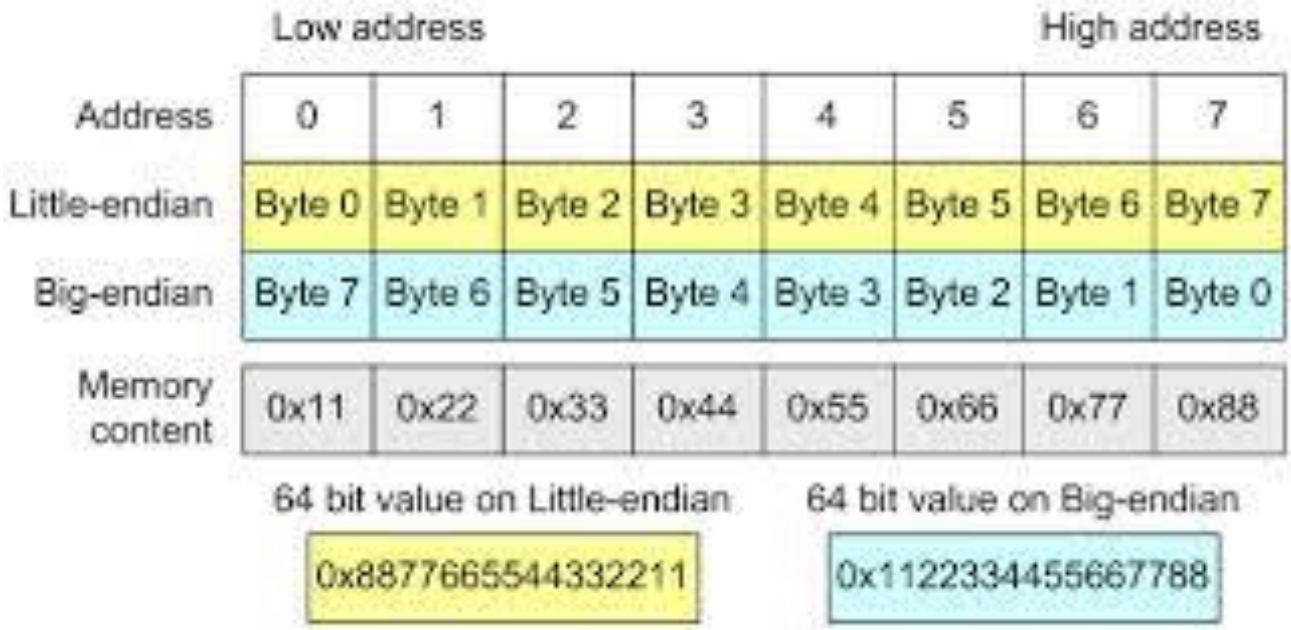
int (*fptr)(int, int)

The above line declares a function pointer ‘fptr’ that can point to a function whose return type is ‘int’ and takes two integers as arguments.

Lets take a working example :

```
#include<stdio.h>

int func (int a, int b)
{
   printf("\n a = %d\n",a);
   printf("\n b = %d\n",b);

   return 0;
}

int main(void)
{
   int(*fptr)(int,int); // Function pointer

   fptr = func; // Assign address to function pointer

   func(2,3);
   fptr(2,3);

   return 0;
}
```

WAP to check whether the system is Little endian or Big endian




```
int main()
{
        int a = 0x12345;
        char *ptr;
        ptr =(char *)&a;
        if(*ptr == 0x45)
        {
                printf("little indian");
        }
        else
        {
                printf("big indian");
        }


}


void show_mem_rep(char *start, int n)
{
   int i;
   for (i = 0; i < n; i++)
       printf(" %.2x", start[i]);
   printf("\n");
}

/*Main function to call above function for 0x01234567*/
int main()
{
```

```
  int i = 0x01234567;
  show_mem_rep((char *)&i, sizeof(i));
  getchar();
  return 0;
}
```

```
#include <stdio.h>
int main()
{
  unsigned int i = 1;
  char *c = (char*)&i;
  if (*c)
     printf("Little endian");
  else
     printf("Big endian");
  getchar();
  return 0;
}
```

WAP to convert little endian to Big endian

WAP to find sizeof a data type without using sizeof() function

```
#include <stdio.h>
#define SIZE_OF(x) ((char*)(&x+1)-(char*)&x)
int main()
{

        double i;
        int aa = SIZE_OF(i);
        printf("\n %d",aa);

    }
```

What is memory leak and explain tools to debug it?

What is structure padding? How we can overcome structure padding? Explain with example

```
struct mystruct_A

{

  char a;
```

```
  int b;
  char c;
} x;


struct mystruct_B
{
  int b;
  char a;
} y;
```

the size of structure is 12 and 8 respectively.

Is this structure padding or packing?

Padding aligns structure members to "natural" address boundaries - say, `int` members would have offsets, which are `mod(4) == 0` on 32-bit platform. Padding is on by default. It inserts the following "gaps" into your first structure:

```
struct mystruct_A {
   char a;
   char gap_0[3]; /* inserted by compiler: for alignment of b */
   int b;
   char c;
   char gap_1[3]; /* -"-: for alignment of the whole struct in an array */
} x;
```

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;
```

```
};
```

would produce structure of size `6` on a 32-bit architecture.

A note though - unaligned memory access is slower on architectures that allow it (like x86 and amd64), and is explicitly prohibited on *strict alignment architectures* like SPARC

What is a call back function and explain with example?

```
#include <stdio.h>
int A(int a,int b)
{
        printf("\n hello");
        return a+b;
}
int B(int(*ptr)(int,int))
{

        return ptr(2,3);
}
int main()
{
//      B(A);
        printf("\b sum = %d",B(A));
    }
```

Explain structures, Unions and Enums? Difference between structurs and unions?

Structure	Union
1.The keyword **struct** is used to define a structure	1. The keyword union is used to define a union.
2. 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. The smaller members may end with unused slack bytes.	2. 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.
3. Each member within a structure is assigned	3. Memory allocated is shared by individual

unique storage area of location.	members of union.
4. The address of each member will be in ascending order This indicates that memory for each member will start at different offset values.	4. The address is same for all the members of a union. This indicates that every member begins at the same offset value.
5 Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. Individual member can be accessed at a time	6. Only one member can be accessed at a time.
7. Several members of a structure can initialize at once.	7. Only the first member of a union can be initialized.

WAP to find the middle node of a single linked list with single traversal.

WAP to delete nth node in a single and double linked list

WAP to print single linked list in reverse order using recursion

WAP to reverse a single linked list

List out difference between inline functions and macros?