C Pointers

What does *p++ do? Does it increment p or the value pointed by p?

The postfix "++" operator has higher precedence than prefix "*" operator. Thus, *p++ is same as *(p++); it increments the pointer p, and returns the value which p pointed to before p was incremented. If you want to increment the value pointed to by p, try (*p)++.

What is a NULL pointer? How is it different from an unitialized pointer? How is a NULL pointer defined?

A null pointer simply means "I am not allocated yet!" and "I am not pointing to anything yet!".

The C language definition states that for every available pointer type, there is a special value which is called the null pointer. It is guaranteed to compare unequal to a pointer to any object or function.

A null pointer is very different from an uninitialized pointer. A null pointer does not point to any object or function; but an uninitialized pointer can point anywhere.

There is usually a null pointer for each type of a pointer, and the internal values of these null pointers for different pointer types may be different, its up to the compiler. The & operator will never yield a null pointer, nor will a successful call to malloc() (malloc() does return a null pointer when it fails).

```
execl("/bin/ls", "ls", "-l", (char *)0);
```

In this call to execl(), the last argument has been explicitly casted to force the 0 to be treated as a pointer.

Also, if ptr is a pointer then

```
if(ptr){}
```

and

```
if(!ptr){}
```

are perfectly valid.

How is NULL defined?, you may ask.

ANSI C allows the following definition

```
#define NULL ((void *)0)
```

NULL and 0 are interchangeable in pointer contexts.

Make sure you are able to distinguish between the following : the null pointer, the internal representation of a null pointer, the null pointer constant (i.e, 0), the NULL macro, the ASCII null character (NUL), the null string ("").

What is a null pointer assignment error?

This error means that the program has written, through a null (probably because its an uninitialized) pointer, to a location thats invalid.

More to come....

Does an array always get converted to a pointer? What is the difference between arr and &arr? How does one declare a pointer to an entire array?

Well, not always.

In C, the array and pointer arithmetic is such that a pointer can be used to access an array or to simulate an array. What this means is whenever an array appears in an expression, the compiler automatically generates a pointer to the array's first element (i.e, &a[0]).

There are three exceptions to this rule

```
1. When the array is the operand of the sizeof() operator.
2. When using the & operator.
3. When the array is a string literal initializer for a character array
.
```

Also, on a side note, the rule by which arrays decay into pointers is not applied recursively!. An array of arrays (i.e. a two-dimensional array in C) decays into a pointer to an array, not a pointer to a pointer.

If you are passing a two-dimensional array to a function:

```
int myarray[NO_OF_ROWS][NO_OF_COLUMNS];
myfunc(myarray);
```

then, the function's declaration must match:

```
void myfunc(int myarray[][NO_OF_COLUMNS])

or

void myfunc(int (*myarray)[NO_OF_COLUMNS])
```

Since the called function does not allocate space for the array, it does not need to know the overall size, so the number of rows, NO_OF_ROWS, can be omitted. The width of the array is still important, so the column dimension
NO_OF_COLUMNS must be present.

An array is never passed to a function, but a pointer to the first element of the array is passed to the function. Arrays are automatically allocated memory. They can't be relocated or resized later. Pointers must be assigned to allocated memory (by using (say) malloc), but pointers can be reassigned and made to point to other memory chunks.

So, whats the difference between func(arr) and func(&arr)?

In C, &arr yields a pointer to the entire array. On the other hand, a simple reference to arr returns a pointer to the first element of the array arr. Pointers to arrays (as in &arr) when subscripted or incremented, step over entire arrays, and are useful only when operating on arrays of arrays. Declaring a pointer to an entire array can be done like int (*arr)[N];, where N is the size of the array.

Also, note that sizeof() will not report the size of an array when the array is a parameter to a function, simply because the compiler pretends that the array parameter was declared as a
pointer and sizeof reports the size of the pointer.

Is the cast to malloc() required at all?

Before ANSI C introduced the void * generic pointer, these casts were required because older compilers used to return a char pointer.

```
int *myarray;
myarray = (int *)malloc(no_of_elements * sizeof(int));
```

But, under ANSI Standard C, these casts are no longer necessary as a void pointer can be assigned to any pointer. These casts are still required with C++, however.

What does malloc() , calloc(), realloc(), free() do? What are the common problems with malloc()? Is there a way to find out how much memory a pointer was allocated?

malloc() is used to allocate memory. Its a memory manager.

calloc(m, n) is also used to allocate memory, just like malloc(). But in addition, it also zero fills the allocated memory area. The zero fill is all-bits-zero. calloc(m.n) is essentially equivalent to

```
p = malloc(m * n);
memset(p, 0, m * n);
```

The malloc() function allocates raw memory given a size in bytes. On the other hand, calloc() clears the requested memory to zeros before return a pointer to it. (It can also compute the request size given the size of the base data structure and the number of them desired.)

The most common source of problems with malloc() are

```
1. Writing more data to a malloc'ed region than it was allocated to hol
d.
2. malloc(strlen(string)) instead of (strlen(string) + 1).
3. Using pointers to memory that has been freed.
4. Freeing pointers twice.
5. Freeing pointers not obtained from malloc.
6. Trying to realloc a null pointer.
```

How does free() work?

Any memory allocated using malloc() realloc() must be freed using free(). In general, for every call to malloc(), there should be a corresponding call to free(). When you call free(), the memory pointed to by the passed pointer is freed. However, the value of the pointer in the caller remains unchanged. Its a good practice to set the pointer to NULL after freeing it to prevent accidental usage. The malloc()/free() implementation keeps track of the size of each block as it is allocated, so it is not required to remind it of the size when freeing it using free(). You can't use dynamically-allocated memory after you free it.

Is there a way to know how big an allocated block is?

Unfortunately there is no standard or portable way to know how big an allocated block is using the pointer to the block!. God knows why this was left out in C.

Is this a valid expression?

```
pointer = realloc(0, sizeof(int));
```

Yes, it is!

What's the difference between const char *p, char * const p and const

char * const p?

```
const char *p    -
   This is a pointer to a constant char. One cannot change the value
                    pointed at by p, but can change the pointer p itse
lf.

                    *p = 'A' is illegal.
                    p  = "Hello" is legal.

                    Note that even char const *p is the same!

const * char p   -   This is a constant pointer to (non-
const) char. One cannot change
                    the pointer p, but can change the value pointed at
 by p.

                    *p = 'A' is legal.
                    p  = "Hello" is illegal.



const char * const p  -
   This is a constant pointer to constant char! One cannot
                         change the value pointed to by p nor the poin
ter.

                         *p = 'A' is illegal.
                         p  = "Hello" is also illegal.
```

To interpret these declarations, let us first consider the general form of declaration:

```
  [qualifier] [storage-class] type [*[*]..] [qualifier] ident ;

                                or

  [storage-class] [qualifier] type [*[*]..] [qualifier] ident ;
```

where,

```
qualifier:
            volatile
```

```
            const

storage-class:

            auto            extern
            static          register

type:
            void            char            short
            int             long            float
            double          signed          unsigned
            enum-specifier
            typedef-name
            struct-or-union-specifier
```

Both the forms are equivalent. Keywords in the brackets are optional. The simplest tip here is to notice the relative position of the `const' keyword with respect to the asterisk (*).

Note the following points:

- If the `const' keyword is to the left of the asterisk, and is the only such keyword in the declaration, then object pointed by the pointer is constant, however, the pointer itself is variable. For example:

```
const char * pcc;
char const * pcc;
```

- If the `const' keyword is to the right of the asterisk, and is the only such keyword in the declaration, then the object pointed by the pointer is variable, but the pointer is constant; i.e., the pointer, once initialized, will always point to the same object through out it's scope. For example:

```
char * const cpc;
```

- If the `const' keyword is on both sides of the asterisk, the both the pointer and the pointed object are constant. For example:

```
const char * const cpcc;
char const * const cpcc2;
```

One can also follow the "nearness" principle; i.e.,

- If the `const' keyword is nearest to the `type', then the object is constant. For example:

```
char const * pcc;
```

- If the `const' keyword is nearest to the identifier, then the pointer is constant. For example:

```
char * const cpc;
```

- If the `const' keyword is nearest, both to the identifier and the type, then both the pointer and the object are constant. For example:

```
const char * const cpcc;
char const * const cpcc2;
```

However, the first method seems more reliable...

What is a void pointer? Why can't we perform arithmetic on a void * pointer?

The void data type is used when no other data type is appropriate. A void pointer is a pointer that may point to any kind of object at all. It is used when a pointer must be specified but its type is unknown.

The compiler doesn't know the size of the pointed-to objects incase of a void * pointer. Before performing arithmetic, convert the pointer either to char * or to the pointer type you're trying to manipulate

What do Segmentation fault, access violation, core dump and Bus error mean?

The segmentation fault, core dump, bus error kind of errors usually mean that the program tried to access memory it shouldn't have.

Probable causes are overflow of local arrays; improper use of null pointers; corruption of the malloc() data structures; mismatched function arguments (specially variable argument functions like sprintf(), fprintf(), scanf(), printf()).

For example, the following code is a sure shot way of inviting a segmentation fault in your program:

```
sprintf(buffer,
        "%s %d",
        "Hello");
```

So whats the difference between a bus error and a segmentation fault?

A bus error is a fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus.

Such conditions include:

```
- Invalid address alignment (accessing a multi-
byte number at an odd address).
- Accessing a memory location outside its address space.
- Accessing a physical address that does not correspond to any device.
- Out-of-bounds array references.
- References through uninitialized or mangled pointers.
```

A bus error triggers a processor-level exception, which Unix translates into a "SIGBUS" signal,which if not caught, will terminate the current process. It looks like a SIGSEGV, but the difference between the two is that SIGSEGV indicates an invalid access to valid memory, while SIGBUS indicates an access to an invalid address.

Bus errors mean different thing on different machines. On systems such as Sparcs a bus error occurs when you access memory that is not positioned correctly.

Maybe an example will help to understand how a bus error occurs

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

int main(void)
{
  char *c;
  long int *i;
  c = (char *) malloc(sizeof(char));
  c++;
  i = (long int *)c;
  printf("%ld", *i);
  return 0;
}
```

On Sparc machines long ints have to be at addresses that are multiples of four (because they are four bytes long), while chars do not (they are only one byte long so they can be put anywhere). The example code uses the char to create an invalid address, and assigns the long int to the invalid address. This causes a bus error when the long int is dereferenced.

A segfault occurs when a process tries to access memory that it is not allowed to, such as the memory at address 0 (where NULL usually points). It is easy to get a segfault, such as the following example, which dereferences NULL.

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

```
int main(void)
{
  char *p;
  p = NULL;
  putchar(*p);
  return 0;
}
```

What is the difference between an array of pointers and a pointer to an array?

This is an array of pointers

```
int *p[10];
```

This is a pointer to a 10 element array

```
int (*p)[10];
```

What is a memory leak?

Its an scenario where the program has lost a reference to an area in the memory. Its a programming term describing the loss of memory. This happens when the program allocates some memory but fails to return it to the system

What are brk() and sbrk() used for? How are they different from malloc()?

brk() and sbrk() are the only calls of memory management in UNIX. For one value of the address, beyond the last logical data page of the process, the MMU generates a segmentation violation interrupt and UNIX kills the process. This address is known as the break address of a process. Addition of a logical page to the data space implies raising of the break address (by a multiple of a page size). Removal of an entry from the page translation table automatically lowers the break address.

brk()and sbrk() are systems calls for this process

```
char *brk(char *new_break);
char *sbrk(displacement)
```

Both calls return the old break address to the process. In brk(), the new break address desired needs to be specified as the parameter. In sbrk(), the displacement (+ve or -ve) is the difference between the new and the old break address. sbrk() is very similar to malloc() when it allocates memory (+ve displacement).

malloc() is really a memory manager and not a memory allocator since, brk/sbrk only can do memory allocations under UNIX. malloc() keeps track of occupied and free peices of memory. Each malloc request is expected to give consecutive bytes and hence malloc selects the smallest free pieces that satisfy a request. When free is called, any consecutive free pieces are coalesced into a large free piece. These is done to avoid fragmentation.

realloc() can be used only with a preallocated/malloced/realloced memory. realloc() will automatically allocate new memory and transfer maximum possible contents if the new space is not available. Hence the returned value of realloc must always be stored back into the old pointer itself.

What is a dangling pointer? What are reference counters with respect to pointers?

A pointer which points to an object that no longer exists. Its a pointer referring to an area of memory that has been deallocated. Dereferencing such a pointer usually produces garbage.

Using reference counters which keep track of how many pointers are pointing to this memory location can prevent such issues. The reference counts are incremented when a new pointer starts to point to the memory location and decremented when they no longer need to point to that memory. When the reference count reaches zero, the memory can be safely freed. Also, once freed, the corresponding pointer must be set to NULL.

What do pointers contain?

Since addresses are always whole numbers, pointers always contain whole numbers

Is *(*(p+i)+j) is equivalent to p[i][j]? Is num[i] == i[num] == *(num + i) == *(i + num)?

Yes

```
*(*(p+i)+j) == p[i][j].
```

So is

```
num[i] == i[num] == *(num + i) == *(i + num)
```

What operations are valid on pointers? When does one get the Illegal use of pointer in function error?

This is what is Valid

```
px<py
px>=py
px==py
px!=py
px==NULL
px=px+n
px=px-n
px-py
```

Everything else is invalid (multiplication, division, addition of two pointers)!

Something like

```
j = j * 2;
k = k / 2;
```

where j and k are pointers will give this error

```
Illegal use of pointer in function main
```

What are near, far and huge pointers?

While working under DOS only 1 mb of memory is accessible. Any of these memory locations are accessed using CPU registers. Under DOS, the CPU registers are only 16 bits long. Therefore, the minimum value present in a CPU register could be 0, and maximum 65,535. Then how do we access memory locations beyond 65535th byte? By using two registers (segment and offset) in conjunction. For this the total

memory (1 mb) is divided into a number of units each comprising 65,536 (64 kb) locations. Each such unit is called a segment. Each segment always begins at a location number which is exactly divisible by 16. The segment register contains the address where a segment begins, whereas the offset register contains the offset of the data/code from where the segment begins. For example, let us consider the first byte in B block of video memory. The segment address of video memory is B0000h (20-bit address), whereas the offset value of the first byte in the upper 32K block of this segment is 8000h. Since 8000h is a 16-bit address it can be easily placed in the offset register, but how do we store the 20-bit address B0000h in a 16-bit segment register? For this out of B0000h only first four hex digits (16 bits) are stored in segment register. We can afford to do this because a segment address is always a multiple of 16 and hence always contains a 0 as the last digit. Therefore, the first byte in the upper 32K chunk of B block of video memory is referred using segment:offset format as B000h:8000h. Thus, the offset register works relative to segment register. Using both these, we can point to a specific location anywhere in the 1 mb address space.

Suppose we want to write a character `A' at location B000:8000. We must convert this address into a form which C understands. This is done by simply writing the segment and offset addresses side by side to obtain a 32 bit address. In our example this address would be 0xB0008000. Now whether C would support this 32 bit address or not depends upon the memory model in use. For example, if we are using a large data model (compact, large, huge) the above address is acceptable. This is because in these models all pointers to data are 32 bits long. As against this, if we are using a small data model (tiny, small, medium) the above address won't work since in these models each pointer is 16 bits long.

What if we are working in small data model and still want to access the first byte of the upper 32K chunk of B block of video memory? In such cases both Microsoft C and Turbo C provide a keyword called far, which is used as shown below,

char far *s = 0XB0008000;

A far pointer is always treated as 32 bit pointer and contains both a segment address and an offset.

A huge pointer is also 32 bits long, again containing a segment address and an offset. However, there are a few differences between a far pointer and a huge pointer.

A near pointer is only 16 bits long, it uses the contents of CS register (if the pointer is pointing to code) or contents of DS register (if the pointer is pointing to data) for the segment part, whereas the offset part is stored in the 16-bit near pointer. Using near pointer limits your data/code to current 64 kb segment.

A far pointer (32 bit) contains the segment as well as the offset. By using far pointers we can have multiple code segments, which in turn allow you to have programs longer than 64 kb. Likewise, with far data pointers we can address more than 64 kb worth of data. However, while using far pointers some problems may crop up as is illustrated by the following program.

```
main( )
{
  char far *a = OX00000120;
  char far *b = OX00100020;
  char far *c = OX00120000;

  if ( a == b )
    printf ( "Hello" ) ;

  if ( a == c )
    printf ( "Hi" ) ;

  if ( b == c )
    printf ( "Hello Hi" ) ;

  if ( a > b && a > c && b > c )
    printf ( "Bye" ) ;
```

```
}
```

Note that all the 32 bit addresses stored in variables a, b, and c refer to the same memory location. This deduces from the method of obtaining the 20-bit physical address from the segment:offset pair. This is shown below.

```
00000  segment address left shifted by 4 bits
0120   offset address
--------
00120 resultant 20 bit address
```

```
00100  segment address left shifted by 4 bits
0020   offset address
--------
00120 resultant 20 bit address
```

```
00120  segment address left shifted by 4 bits
0000   offset address
--------
00120 resultant 20 bit address
```

Now if a, b and c refer to same location in memory we expect the first three ifs to be satisfied. However this doesn't happen. This is because while comparing the far pointers using == (and !=) the full 32-bit value is used and since the 32-bit values are different the ifs fail. The last if however gets satisfied, because while comparing using > (and >=, <, <=) only the offset value is used for comparison. And the offset values of a, b and c are such that the last condition is satisfied.

These limitations are overcome if we use huge pointer instead of far pointers. Unlike far pointers huge pointers are `normalized' to avoid these problems. What is a normalized pointer? It is a 32- bit pointer which has as much of its value in the segment address as possible. Since a segment can start every 16 bytes, this means that the offset will only have a value from 0 to F.

How do we normalize a pointer? Simple. Convert it to its 20-bit address then use the the left 16 bits for the segment address and the right 4 bits for the offset address. For example, given the pointer 500D:9407, we convert it to the absolute address 594D7, which we then normalize to 594D:0007.

Huge pointers are always kept normalized. As a result, for any given memory address there is only one possible huge address - segment:offset pair for it. Run the above program using huge instead of far and now you would find that the first three ifs are satisfied, whereas the fourth fails. This is more logical than the result obtained while using far. But then there is a price to be paid for using huge pointers. Huge pointer arithmetic is done with calls to special subroutines. Because of this, huge pointer arithmetic is significantly slower than that of far or near pointers. The information presented is specific to DOS operating system only.

What is the difference between malloc() and calloc()?

First lets look at the prototypes of these two popular functions..

```
#include <stdlib.h>
void *calloc(size_t n, size_t size);
void *malloc(size_t size);
```

The two functions malloc() and calloc() are functionally same in that they both allocate memory from a storage pool (generally called heap). Actually, the right thing to say is that these two functions are memory managers and not memory allocators. Memory allocation is done by OS specific routines (like brk() and sbrk()). But lets not get into that for now...

Here are some differences between these two functions..

- malloc() takes one argument, whereas calloc() takes two.
- calloc() initializes all the bits in the allocated space to zero (this is all-bits-zero!, where as malloc() does not do this.
- A call to calloc() is equivalent to a call to malloc() followed by one to memset().

  ```
  calloc(m, n)

  is essentially equivalent to

  p = malloc(m * n);
  memset(p, 0, m * n);
  ```

  Using calloc(), we can carry out the functionality in a faster way than a combination of malloc() and memset() probably would. You will agree that one libray call is faster than two calls. Additionally, if provided by the native CPU, calloc() could be implementated by the CPU's "allocate-and-initialize-to-zero" instruction.
- The reason for providing the "n" argument is that sometimes it is required to allocate a number ("n") of uniform objects of a particular size ("size"). Database application, for instance, will have such requirements. Proper planning for the values of "n" and "size" can lead to good memory utilization.

Why is sizeof() an operator and not a function?

sizeof() is a compile time operator. To calculate the size of an object, we need the type information. This type information is available only at compile time. At the end of the compilation phase, the resulting object code doesn't have (or not required to have) the type information. Of course, type information can be stored to access it at run-time, but this results in bigger object code and less performance. And most of the time, we don't need it. All the runtime environments that support run time type identification (RTTI) will retain type information even after compilation phase. But, if something can be done in compilation time itself, why do it

at run time?

On a side note, something like this is illegal...

```
printf("%u\n", sizeof(main));
```

This asks for the size of the main function, which is actually illegal:

```
6.5.3.4 The sizeof operator
The sizeof operator shall not be applied to an expression that has func
tion type....
```

What is an opaque pointer?

A pointer is said to be opaque if the definition of the type to which it points to is not included in the current translation unit. A translation unit is the result of merging an implementation file with all its headers and header files

What are the common causes of pointer bugs?

- Uninitialized pointers : One of the easiest ways to create a pointer bug is to try to reference the value of a pointer even though the pointer is uninitialized and does not yet point to a valid address. For example:

  ```
  int *p;
  *p = 12;
  ```

  The pointer p is uninitialized and points to a random location in memory when you declare it. It could be pointing into the system stack, or the global variables, or into the program's code space, or into the operating system. When you say *p=12;, the program will simply try to write a 12 to whatever random location p points to. The program may explode immediately, or may wait half an hour and then explode, or it may subtly corrupt data in another part of your program and you may never realize it. This can make this error very hard to track down. Make sure you initialize all pointers to a valid address before dereferencing them.

- Invalid Pointer References : An invalid pointer reference occurs when a pointer's value is referenced even though the pointer doesn't point to a valid block. One way to create this error is to

say p=q;, when q is uninitialized. The pointer p will then become uninitialized as well, and any reference to *p is an invalid pointer reference. The only way to avoid this bug is to draw pictures of each step of the program and make sure that all pointers point somewhere. Invalid pointer references cause a program to crash inexplicably for the same reasons given in cause 1.

- Zero Pointer Reference : A zero pointer reference occurs whenever a pointer pointing to zero is used in a statement that attempts to reference a block. For example, if p is a pointer to an integer, the following code is invalid:

```
p = 0;
*p = 12;
```

There is no block pointed to by p. Therefore, trying to read or write anything from or to that block is an invalid zero pointer reference. There are good, valid reasons to point a pointer to zero. Dereferencing such a pointer, however, is invalid.

C Functions

How to declare a pointer to a function?

Use something like this

```
int myfunc(); // The function.
int (*fp)();  // Pointer to that function.

fp = myfunc;  // Assigning the address of the function to the pointer.

(*fp)();      // Calling the function.
fp();         // Another way to call the function.
```

So then, what are function pointers, huh?

In simple words, a function pointer is a pointer variable which holds the address of a function and can be used to invoke that function indirectly. Function pointers are useful when you want to invoke seperate functions based on different scenarios. In that case, you just pass in the pointer to the function you want to be invoked. Function pointers are used extensively when writing code with call back scenarios.

For example, when writing code involving a XML parser, you pass in the pointers to your event handling functions and whenever there is a specific event, the XML parser calls your functions through their pointers. This function whose pointer is passed is generally called as the call back function.

Here is an implementation of a Generic linked list which uses function pointers really well...

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

typedef struct list {
    void *data;
    struct list *next;
} List;

struct check {
    int i;
    char c;
    double d;
} chk[] = { { 1, 'a', 1.1 },
          { 2, 'b', 2.2 },
          { 3, 'c', 3.3 } };




// See how the print() function takes in a pointer to a function!
void print(List *, void (*)(void *));




void insert(List **, void *, unsigned int);
void printstr(void *);
void printint(void *);
void printchar(void *);
void printcomp(void *);

List *list1, *list2, *list3, *list4;

int main(void)
{
    char c[]    = { 'a', 'b', 'c', 'd' };
    int i[]     = { 1, 2, 3, 4 };
    char *str[] = { "hello1", "hello2", "hello3", "hello4" };

    list1 = list2 = list3 = list4 = NULL;

    insert(&list1, &c[0], sizeof(char));
    insert(&list1, &c[1], sizeof(char));
    insert(&list1, &c[2], sizeof(char));
    insert(&list1, &c[3], sizeof(char));

    insert(&list2, &i[0], sizeof(int));
    insert(&list2, &i[1], sizeof(int));
    insert(&list2, &i[2], sizeof(int));
    insert(&list2, &i[3], sizeof(int));

    insert(&list3, str[0], strlen(str[0])+1);
```

```
    insert(&list3, str[1], strlen(str[0])+1);
    insert(&list3, str[2], strlen(str[0])+1);
    insert(&list3, str[3], strlen(str[0])+1);

    insert(&list4, &chk[0], sizeof chk[0]);
    insert(&list4, &chk[1], sizeof chk[1]);
    insert(&list4, &chk[2], sizeof chk[2]);

    printf("Printing characters:");
    print(list1, printchar);
    printf(" : done\n\n");

    printf("Printing integers:");
    print(list2, printint);
    printf(" : done\n\n");

    printf("Printing strings:");
    print(list3, printstr);
    printf(" : done\n\n");

    printf("Printing composite:");
    print(list4, printcomp);
    printf(" : done\n");

    return 0;
}

void insert(List **p, void *data, unsigned int n)
{
    List *temp;
    int i;

    /* Error check is ignored */
    temp = malloc(sizeof(List));
    temp->data = malloc(n);
    for (i = 0; i < n; i++)
        *(char *)(temp->data + i) = *(char *)(data + i);
    temp->next = *p;
    *p = temp;
}

void print(List *p, void (*f)(void *))
{
    while (p)
    {
        (*f)(p->data);
        p = p->next;
    }
}

void printstr(void *str)
{
    printf(" \"%s\"", (char *)str);
}

void printint(void *n)
```

```
{
    printf(" %d", *(int *)n);
}

void printchar(void *c)
{
    printf(" %c", *(char *)c);
}

void printcomp(void *comp)
{
    struct check temp = *(struct check *)comp;
    printf(" '%d:%c:%f", temp.i, temp.c, temp.d);
}
```

Does extern in a function declaration mean anything?

The extern in a function's declaration is sometimes used to indicate that the function's definition is in some other source file, but there is no difference between

extern int function_name();

and

int function_name();

How can I return multiple values from a function?

You can pass pointers to locations which the function being called can populate, or have the function return a structure containing the desired values, or use global variables.

Does C support function overloading?

Function overload is not present in C. In C, either use different names or pass a union of supported types (with additional identifier that give hints of the type to be used).

Most people think its supported because they might unknowingly be using a C++ compiler to compile their C code and C+= does have function overloading

What is the purpose of a function prototype?

A function prototype tells the compiler to expect a given function to be used in a given way. That is, it tells the compiler the nature of the parameters passed to the function (the quantity, type and order) and the nature of the value returned by the function.

What are inline functions?

Before reading the answer, please be aware that inline functions are non-standard C. They are provided as compiler extensions. But, nevertheless, one should know what they are used for.

The inline comment is a request to the compiler to copy the code into the object at every place the function is called. That is, the function is expanded at each point of call. Most of the advantage of inline functions comes from avoiding the overhead of calling an actual function. Such overhead includes saving registers, setting up stack frames, and so on. But with large functions the overhead becomes less important. Inlining tends to blow up the size of code, because the function is expanded at each point of call.

```
int myfunc(int a)
{
  ...
}

inline int myfunc(int a)
{
  ...
}
```

Inlined functions are not the fastest, but they are the kind of better than macros (which people use normally to write small functions).

```
#define myfunc(a)   \
{ \
   ...                              \
}
```

The problem with macros is that the code is literally copied into the location it was called from. So if the user passes a "double" instead of an "int" then problems could occur. However, if this senerio happens with an inline function the compiler will complain about incompatible types. This will save you debugging time stage.

Good time to use inline functions is

```
1. There is a time criticle function.
2. That is called often.
3. Its small. Remember that inline functions take more space than norma
l functions.
```

Some typical reasons why inlining is sometimes not done are:

```
1. The function calls itself, that is, is recursive.
2. The function contains loops such as for(;;) or while().
3. The function size is too large.
```

How to declare an array of N pointers to functions returning pointers to functions returning pointers to characters?

Declare it this way

```
char *(*(*a[N])())();
```

Even this seems to be correct

```
 (char*(*fun_ptr)())(*func[n])();
```

Can we declare a function that can return a pointer to a function of the same type?

We cannot do it directly. Either have the function return a generic function pointer, with casts to adjust the types as the pointers are passed around; or have it return a structure containing only a pointer to a function returning that structure.

How can I write a function that takes a variable number of arguments? What are the limitations with this? What is vprintf()?

The header stdarg.h provides this functionality. All functions like printf(), scanf() etc use this functionality.

The program below uses a var_arg type of function to count the overall length of strings passed to the function.

```
#include <stdarg.h>

int myfunction(char *first_argument,...)
```

```
{
   int length;
   va_list argp;
   va_start(argp, first);
   char *p;

   length = strlen(first_argument);
   while((p = va_arg(argp, char *)) != NULL)
   {
     length = length + strlen(p);
   }

   va_end(argp);
   return(length);
}

int main()
{
  int length;
  length = myfunction("Hello","Hi","Hey!",(char *)NULL);
  return(0);
}
```

How can I find how many arguments a function was passed?

Any function which takes a variable number of arguments must be able to determine from the arguments themselves, how many of them there have been passed. printf() and some similar functions achieve this by looking for the format string. This is also why these functions fail badly if the format string does not match the argument list. Another common technique, applicable when the arguments are all of the same type, is to use a sentinel value (often 0, -1, or an appropriately-cast null pointer) at the end of the list. Also, one can pass an explicit count of the number of variable arguments. Some older compilers did provided a nargs() function, but it was never portable.

Is this allowed?

```
int f(...)
{
  ...
}
```

No! Standard C requires at least one fixed argument, in part so that you can hand it to va_start().

So how do I get floating point numbers passed as arguments?

Arguments of type float are always promoted to type double, and types char and short int are promoted to int. Therefore, it is never correct to invoke

```
va_arg(argp, float);
```

instead you should always use

```
va_arg(argp, double)
```

Similarly, use

```
va_arg(argp, int)
```

to retrieve arguments which were originally char, short, or int.

How can I create a function which takes a variable number of arguments and passes them to some other function (which takes a variable number of arguments)?

You should provide a version of that other function which accepts a va_list type of pointer.

So how can I call a function with an argument list built up at run time?

There is no portable way to do this. Instead of an actual argument list, you might want to pass an array of generic (void *) pointers. The called function can then step through the array, much like main() steps through char *argv[].

What is the use of vprintf(), vfprintf() and vsprintf()?

Below, the myerror() function prints an error message, preceded by the string "error: " and terminated with a newline:

```
#include <stdio.h>
#include <stdarg.h>
void myerror(char *fmt, ...)
{
  va_list argp;
  fprintf(stderr, "error: ");
  va_start(argp, fmt);
  vfprintf(stderr, fmt, argp);
  va_end(argp);
  fprintf(stderr, "\n");
}
```

With respect to function parameter passing, what is the difference between call-by-value and call-by-reference? Which method does C

use?

In the case of call-by-reference, a pointer reference to a variable is passed into a function instead of the actual value. The function's operations will effect the variable in a global as well as local sense. Call-by-value (C's method of parameter passing), by contrast, passes a copy of the variable's value into the function. Any changes to the variable made by function have only a local effect and do not alter the state of the variable passed into the function.

Does C really have pass by reference?

No.

C uses pass by value, but it can pretend doing pass by reference, by having functions that have pointer arguments and by using the & operator when calling the function. This way, the compiler will simulate this feature (like when you pass an array to a function, it actually passes a pointer instead). C does not have something like the formal pass by reference or C++ reference parameters.

How will you decide whether to use pass by reference, by pointer and by value?

The selection of the argument passing depends on the situation.

- A function uses passed data without modifying it:
    - If the data object is small, such as a built-in data type or a small structure then pass it by value.
    - If the data object is an array, use a pointer because that is the only choice. Make the pointer a pointer to const.
    - If the data object is a good-sized structure, use a const pointer or a const reference to increase program efficiency. You save the time and space needed to copy a structure or a class design, make the pointer or reference const.
    - If the data object is a class object, use a const reference. The semantics of class design often require using a reference. The standard way to pass class object arguments is by reference.

- A function modifies data in the calling function:
    - If the data object is a built-in data type, use a pointer. If you spot a code like fixit(&x), where x is an int, its clear that this function intends to modify x.
    - If the data object is an array, use the only choice, a pointer.
    - If the data object is a structure, use a reference or a pointer.
    - If the data object is a class object, use a reference.

If I have the name of a function in the form of a string, how can I invoke that function?

Keep a table of names and their function pointers:

```
int myfunc1(), myfunc2();

struct
{
  char *name;
  int (*func_ptr)();
} func_table[] = {"myfunc1", myfunc1,
                  "myfunc2", myfunc2,};
```

Search the table for the name, and call via the associated function pointer.

What does the error, invalid redeclaration of a function mean?

If there is no declaration in scope then it is assumed to be declared as returning an int and without any argument type information. This can lead to discrepancies if the function is later declared or defined. Such functions must be declared before they are called. Also check if there is another function in some header file with the same name.

How can I pass the variable argument list passed to one function to another function.

Good question

Something like this wont work

```
#include <stdarg.h>
main()
{
  display("Hello", 4, 12, 13, 14, 44);
}
```

```
display(char *s,...)
{
  show(s,...);
}

show(char *t,...)
{
  va_list ptr;
  int a;
  va_start(ptr,t);
  a = va_arg(ptr, int);
  printf("%f", a);
}
```

This is the right way of doing it

```
#include <stdarg.h>
main()
{
  display("Hello", 4, 12, 13, 14, 44);
}

display(char *s,...)
{
  va_list ptr;
  va_start(ptr, s);
  show(s,ptr);
}

show(char *t, va_list ptr1)
{
  int a, n, i;
  a=va_arg(ptr1, int);

  for(i=0; i<a; i++)
  {
     n=va_arg(ptr1, int);
     printf("\n%d", n);
  }
}
```

How do I pass a variable number of function pointers to a variable argument (va_arg) function?

Glad that you thought about doing something like this!

Here is some code

```
#include <stdarg.h>

main()
{
  int (*p1)();
  int (*p2)();
  int fun1(), fun2();

  p1 = fun1;
  p2 = fun2;
  display("Bye", p1, p2);
}

display(char *s,...)
{
  int (*pp1)(), (*pp2)();
  va_list ptr;
  typedef int (*f)(); //This typedef is  very important.

  va_start(ptr,s);

  pp1 = va_arg(ptr, f); // va_arg(ptr, int (*)()); would NOT have worke
d!
  pp2 = va_arg(ptr, f);

  (*pp1)();
  (*pp2)();

}

fun1()
{
  printf("\nHello!\n");
}

fun2()
{
  printf("\nHi!\n");
}
```

Will C allow passing more or less arguments than required to a function.

It wont if the prototpe is around. It will ideally scream out with an error like

```
Too many arguments
```

or

```
Too few arguments
```

But if the prototype is not around, the behavior is undefined.

Try this out

```
#include <stdio.h>

/*
int foo(int a);
int foo2(int a, int b);
*/

int main(int a)
{
  int (*fp)(int a);

  a = foo();
  a = foo2(1);

  exit(0);
}

int foo(int a)
{
  return(a);
}

int foo2(int a, int b)
{
  return(a+b);
}
```

C Statements

Whats short-circuiting in C expressions?

What this means is that the right hand side of the expression is not evaluated if the left hand side determines the outcome. That is if the left hand side is true for || or false for &&, the right hand side is not evaluated.

Whats wrong with the expression a[i]=i++; ? Whats a sequence point?

Although its surprising that an expression like i=i+1; is completely valid, something like a[i]=i++; is not. This is because all accesses to an element must be to change the value of that variable. In the statement a[i]=i++; , the access to i is not for itself, but for a[i] and so its invalid. On similar lines, i=i++; or i=++i; are invalid. If you want to increment the value of i, use i=i+1; or i+=1; or i++; or ++i; and not some combination.

A sequence point is a state in time (just after the evaluation of a full expression, or at the ||, &&, ?:, or comma operators, or just before a call to a function) at which there are no side effects.

The ANSI/ISO C Standard states that

```
Between the previous and next sequence point an object shall have its s
tored
value modified at most once by the evaluation of an expression. Further
more,
the prior value shall be accessed only to determine the value to be sto
red.
```

At each sequence point, the side effects of all previous expressions will be completed. This is why you cannot rely on expressions such as a[i] = i++;, because there is no sequence point specified for the assignment, increment or index operators, you don't know when the effect of the increment on i occurs.

The sequence points laid down in the Standard are the following:

- The point of calling a function, after evaluating its arguments.
- The end of the first operand of the && operator.
- The end of the first operand of the || operator.
- The end of the first operand of the ?: conditional operator.
- The end of the each operand of the comma operator.
- Completing the evaluation of a full expression. They are the following:

    - Evaluating the initializer of an auto object.
    - The expression in an ?ordinary? statement?an expression followed by semicolon.
    - The controlling expressions in do, while, if, switch or for statements.
    - The other two expressions in a for statement.
    - The expression in a return statement.

Does the ?: (ternary operator) return a lvalue? How can I assign a value to the output of the ternary operator?

No, it does not return an "lvalue"

Try doing something like this if you want to assign a value to the output of this operator

```
*((mycondition) ? &var1 : &var2) = myexpression;
```

Is 5[array] the same as array[5]?

Yes!

Since array subscripting is commutative in C. That is

```
array[n] == *((array)+(n)) == *((n)+(array)) == n[array]
```

But frankly, this feature is of no use to anyone. Anyone asking this question is more or less a fool trying to be cool.

What are #pragmas?

The directive provides a single, well-defined "escape hatch" which can be used for all sorts of (nonportable) implementation-specific controls and extensions: source listing control, structure packing, warning suppression (like lint's old /* NOTREACHED */ comments), etc.

For example

```
#pragma once
```

inside a header file is an extension implemented by some preprocessors to help make header files idempotent (to prevent a header file from included twice).

What is the difference between if(0 == x) and if(x == 0)?

Nothing!. But, it's a good trick to prevent the common error of writing

```
if(x = 0)
```

The error above is the source of a lot of serious bugs and is very difficult to catch. If you

cultivate the habit of writing the constant before the ==, the compiler will complain if you accidentally type

```
if(0 = x)
```

Of course, the trick only helps when comparing to a constant.

Should we use goto or not?

You should use gotos wherever it suits your needs really well. There is nothing wrong in using them. Really.

There are cases where each function must have a single exit point. In these cases, it makes much sense to use gotos.

```
myfunction()
{
  if(error_condition1)
  {
    // Do some processing.
    goto failure;
  }

  if(error_condition2)
  {
    // Do some processing.
    goto failure;
  }

success:
  return(TRUE);

failure:

  // Do some cleanup.
  return(FALSE);
}
```

Also, a lot of coding problems lend very well to the use of gotos. The only argument against gotos is that it can make the code a little un-readable. But if its commented properly, it works quite fine.

Is ++i really faster than i = i + 1?

Anyone asking this question does not really know what he is talking about.

Any good compiler will and should generate identical code for ++i, i += 1, and i = i + 1. Compilers are meant to optimize code. The programmer should not be bother about such things. Also, it depends on the processor and compiler you are using. One needs to check the compiler's assembly language output, to see which one of the different approcahes are better, if at all.

Note that speed comes Good, well written algorithms and not from such silly tricks.

What do lvalue and rvalue mean?

An lvalue is an expression that could appear on the left-hand sign of an assignment (An object that has a location). An rvalue is any expression that has a value (and that can appear on the right-hand sign of an assignment).

The lvalue refers to the left-hand side of an assignment expression. It must always evaluate to a memory location. The rvalue represents the right-hand side of an assignment expression; it may have any meaningful combination of variables and constants.

Is an array an expression to which we can assign a value?

An lvalue was defined as an expression to which a value can be assigned. The answer to this question is no, because an array is composed of several separate array elements that cannot be treated as a whole for assignment purposes.

The following statement is therefore illegal:

```
int x[5], y[5];
x = y;
```

Additionally, you might want to copy the whole array all at once. You can do so using a library function such as the memcpy() function, which is shown here:

```
memcpy(x, y, sizeof(y));
```

It should be noted here that unlike arrays, structures can be treated as lvalues. Thus, you can assign one structure variable to another structure variable of the same type, such as this:

```
typedef struct t_name
{
  char last_name[25];
  char first_name[15];
  char middle_init[2];
} NAME;
...
  NAME my_name, your_name;
```

```
...
  your_name = my_name;
...
```

What does the term cast refer to? Why is it used?

Casting is a mechanism built into C that allows the programmer to force the conversion of data types. This may be needed because most C functions are very particular about the data types they process. A programmer may wish to override the default way the C compiler promotes data types.

What is the difference between a statement and a block?

A statement is a single C expression terminated with a semicolon. A block is a series of statements, the group of which is enclosed in curly-braces.

Can comments be nested in C?

No

What is type checking?

The process by which the C compiler ensures that functions and operators use data of the appropriate type(s). This form of check helps ensure the semantic correctness of the program

Why can't you nest structure definitions?

This is a trick question!

You can nest structure definitions.

```
struct salary
{
    char empname[20];
    struct
    {
        int dearness;
    }
    allowance;
}employee;
```

What is a forward reference?

It is a reference to a variable or function before it is defined to the compiler. The cardinal rule of structured languages is that everything must be defined before it can be used. There are rare occasions where this is not possible. It is possible (and sometimes necessary) to define two functions in terms of each other. One will obey the cardinal rule while the other will need a forward declaration of the former in order to know of the former's existence.

What is the difference between the & and && operators and the | and || operators?

& and | are bitwise AND and OR operators respectively. They are usually used to manipulate the contents of a variable on the bit level. && and || are logical AND and OR operators respectively. They are usually used in conditionals

Is C case sensitive (ie: does C differentiate between upper and lower case letters)?

Yes, ofcourse!

Can goto be used to jump across functions?

No!

This wont work

```
main()
{
  int i=1;
  while (i<=5)
  {
    printf("%d",i);
    if (i>2)
        goto here;
    i++;
  }
}

fun()
{
here:
  printf("PP");
}
```

Whats wrong with #define myptr int *?

```
#define myptr int *
myptr p, q;
```

Only p will be a pointer to an int, and q will just be an int!.

Use typdef for such a requirement.

What purpose do #if, #else, #elif, #endif, #ifdef, #ifndef serve?

The following preprocessor directives are used for conditional compilation. Conditional compilation allows statements to be included or omitted based on conditions at compile time.

```
#if
#else
#elif
#endif
#ifdef
#ifndef
```

In the following example, the printf statements are compiled when the symbol DEBUG is defined, but not compiled otherwise

```
/* remove to suppress debug printf's*/
#define DEBUG
...
x = ....

  #ifdef DEBUG
    printf( "x=%d\n" );
  #endif...

y = ....;

  #ifdef DEBUG
    printf( "y=%d\n" );
  #endif...
```

#if, #else, #elif statements

#if directive

- #if is followed by a intger constant expression.
- If the expression is not zero, the statement(s) following the #if are compiled, otherwise they are ignored.
- #if statements are bounded by a matching #endif, #else or #elif
- Macros, if any, are expanded, and any undefined tokens are replaced with 0 before the constant expression is evaluated
- Relational operators and integer operators may be used

Expression examples

```
#if 1
#if 0
#if ABE == 3
#if ZOO < 12
#if ZIP == 'g'
#if (ABE + 2 - 3 * ZIP) > (ZIP - 2)
```

In most uses, expression is simple relational, often equality test

```
#if SPARKY == '7'
```

#else directive

- #else marks the beginning of statement(s) to be compiled if the preceding #if or #elif expression is zero (false)
- Statements following #else are bounded by matching #endif

Examples

```
#if OS = 'A'
  system( "clear" );
#else
  system( "cls" );
#endif
```

#elif directive

- #elif adds an else-if branch to a previous #if
- A series of #elif's provides a case-select type of structure
- Statement(s) following the #elif are compiled if the expression is not zero, ignored otherwise
- Expression is evaluated just like for #if

Examples

```
#if TST == 1
  z = fn1( y );
#elif TST == 2
  z = fn2( y, x );
#elif TST == 3
  z = fn3( y, z, w );
#endif

...
#if ZIP == 'g'
  rc = gzip( fn );
#elif ZIP == 'q'
  rc = qzip( fn );
#else
```

```
  rc = zip( fn );
#endif
```

#ifdef and #ifndef directives

Testing for defined macros with #ifdef, #ifndef, and defined()

- #ifdef is used to include or omit statements from compilation depending of whether a macro name is defined or not.
- Often used to allow the same source module to be compiled in different environments (UNIX/ DOS/MVS), or with different options (development/production).
- #ifndef similar, but includes code when macro name is not defined.

Examples

```
#ifdef TESTENV
  printf( "%d ", i );
#endif
#ifndef DOS
  #define LOGFL "/tmp/loga.b";
#else
  #define LOGFL "c:\\tmp\\log.b";
#endif
```

defined() operator

- defined(mac), operator is used with #if and #elif and gives 1 (true) if macro name mac is defined, 0 (false) otherwise.
- Equivalent to using #ifdef and #ifndef, but many shops prefer #if with defined(mac) or !defined(mac)

Examples

```
#if defined(TESTENV)
  printf( "%d ", i );
#endif
#if !defined(DOS)
  #define LOGFL "/tmp/loga.b";
#else
  #define LOGFL "c:\\tmp\\log.b";
#endif
```

Nesting conditional statements

Conditional compilation structures may be nested:

```
#if defined(UNIX)
  #if LOGGING == 'y'
     #define LOGFL "/tmp/err.log"
  #else
     #define LOGFL "/dev/null"
  #endif
#elif defined( MVS )
  #if LOGGING == 'y'
     #define LOGFL "TAP.AVS.LOG"
  #else
     #define LOGFL "NULLFILE"
  #endif
#elif defined( DOS )
  #if LOGGING == 'y'
     #define LOGFL "C:\\tmp\\err.log"
  #else
     #define LOGFL "nul"
  #endif
#endif
```

Can we use variables inside a switch statement? Can we use floating point numbers? Can we use expressions?

No

The only things that case be used inside a switch statement are constants or enums. Anything else will give you a

```
constant expression required
```

error. That is something like this is not valid

```
switch(i)
{
  case 1: // Something;
          break;
  case j: // Something;
          break;
}
```

So is this. You cannot switch() on strings

```
switch(i)
{
  case "string1" : // Something;
                   break;
  case "string2" : // Something;
                   break;
}
```

This is valid, however

```
switch(i)
{
  case 1:     // Something;
              break;
  case 1*2+4: // Something;
              break;
}
```

This is also valid, where t is an enum

```
switch(i)
{
  case 1: // Something;
          break;
  case t: // Something;
          break;
}
```

Also note that the default case does not require a break; if and only if its at the end of the switch() statement. Otherwise, even the default case requires a break;

What is more efficient? A switch() or an if() else()?

Both are equally efficient. Usually the compiler implements them using jump instructions. But each of them has their own unique advantages.

What is the difference between a deep copy and a shallow copy?

Deep copy involves using the contents of one object to create another instance of the same class. In a deep copy, the two objects may contain ht same information but the target object will have its own buffers and resources. the destruction of either object will not affect the remaining object. The overloaded assignment operator would create a deep copy of objects.

Shallow copy involves copying the contents of one object into another instance of the same class thus creating a mirror image. Owing to straight copying of references and pointers, the two objects will share the same externally contained contents of the other object to be unpredictable.
Using a copy constructor we simply copy the data values member by member. This method of copying is called shallow copy. If the object is a simple class, comprised of built in types and no pointers this would be acceptable. This function would use the values and the objects and its behavior would not be altered with a shallow copy, only the addresses of pointers that are members are copied and not the value the address is pointing to. The data values of the object would then be inadvertently altered by the function. When the function goes out of scope, the copy of the object with all its data is popped off the stack. If the object has any pointers a deep copy needs to be executed. With the deep copy of an object, memory is allocated for the object in free store and the elements pointed to are copied. A deep copy is used for objects that are returned from a function.

What is operator precedence?

This questions will be answered soon :)

C Arrays

How to write functions which accept two-dimensional arrays when the width is not known before hand?

Try something like

```
myfunc(&myarray[0][0], NO_OF_ROWS, NO_OF_COLUMNS);

void myfunc(int *array_pointer, int no_of_rows, int no_of_columns)
{
  // myarray[i][j] is accessed as array_pointer[i * no_of_columns + j]
}
```

Is char a[3] = "abc"; legal? What does it mean?

It declares an array of size three, initialized with the three characters 'a', 'b', and 'c', without the usual terminating '\0' character. The array is therefore not a true C string and cannot be used with strcpy, printf %s, etc. But its legal.

If a is an array, is a++ valid?

No

You will get an error like

```
Error message : Lvalue required in function main.
```

Doing a++ is asking the compiler to change the base address of the array. This the only thing that the compiler remembers about an array once its declared and it wont allow you to change the base address. If it allows, it would be unable to remember the beginning of the array.

How can we find out the length of an array dynamically in C?

Here is a C program to do the same...

```
#include <stdio.h>
#include <conio.h>
int main()
{
  int arr[] = {3,4,65,78,1,2,4};
  int arr_length = sizeof(arr)/sizeof(arr[0]);

  printf("\nLength of the array is :[%d]\n\n", arr_length);

  getch();
  return(0);
}
```

C Variables

What is the difference between the declaration and the definition of a variable?

The definition is the one that actually allocates space, and provides an initialization value, if any.

There can be many declarations, but there must be exactly one definition. A definition tells the compiler to set aside storage for the variable. A declaration makes the variable known to parts of the program that may wish to use it. A variable might be defined and declared in the same statement.

Do Global variables start out as zero?

Glad you asked!

Uninitialized variables declared with the "static" keyword are initialized to zero. Such variables are implicitly initialized to the null pointer if they are pointers, and to 0.0F if they are floating point numbers.

Local variables start out containing garbage, unless they are explicitly initialized.

Memory obtained with malloc() and realloc() is likely to contain junk, and must be initialized. Memory obtained with calloc() is all-bits-0, but this is not necessarily useful for pointer or floating-point values (This is in contrast to Global pointers and Global floating point numbers, which start as zeroes of the right type).

Does C have boolean variable type?

No, C does not have a boolean variable type. One can use ints, chars, #defines or enums to achieve the same in C.

```
#define TRUE 1
#define FALSE 0

enum bool {false, true};
```

An enum may be good if the debugger shows the names of enum constants when examining variables.

Where may variables be defined in C?

Outside a function definition (global scope, from the point of definition downward in the source code). Inside a block before any statements other than variable declarations (local scope with respect to the block).

To what does the term storage class refer? What are auto, static, extern, volatile, const classes?

This is a part of a variable declaration that tells the compiler how to interpret the variable's symbol. It does not in itself allocate storage, but it usually tells the compiler how the variable should be stored. Storage class specifiers help you to specify the type of storage used for data objects. Only one storage class specifier is permitted in a declaration this makes sense, as there is only one way of storing things and if you omit the storage class specifier in a declaration, a default is chosen. The default depends on whether the declaration is made outside a function (external declarations) or inside a function (internal declarations). For external declarations the default storage class specifier will be extern and for internal declarations it will be auto. The only exception to this rule is the declaration of functions, whose default storage class specifier is always extern.

Here are C's storage classes and what they signify:

- auto - local variables.
- static - variables are defined in a nonvolatile region of memory such that they retain their contents though out the program's execution.
- register - asks the compiler to devote a processor register to this variable in order to speed the program's execution. The compiler may not comply and the variable looses it contents and identity when the function it which it is defined terminates.
- extern - tells the compiler that the variable is defined in another module.

In C, const and volatile are type qualifiers. The const and volatile type qualifiers are completely independent. A common misconception is to imagine that somehow const is the opposite of volatile and vice versa. This is wrong. The keywords const and volatile can be applied to any declaration, including those of structures, unions, enumerated types or typedef names. Applying them to a declaration is called qualifying the declaration?that's why const and volatile are called type qualifiers, rather than type specifiers.

- const means that something is not modifiable, so a data object that is declared with const as a part of its type specification must not be assigned to in any way during the run of a program. The main

intention of introducing const objects was to allow them to be put into read-only store, and to permit compilers to do extra consistency checking in a program. Unless you defeat the intent by doing naughty things with pointers, a compiler is able to check that const objects are not modified explicitly by the user. It is very likely that the definition of the object will contain an initializer (otherwise, since you can't assign to it, how would it ever get a value?), but this is not always the case. For example, if you were accessing a hardware port at a fixed memory address and promised only to read from it, then it would be declared to be const but not initialized.

- volatile tells the compiler that other programs will be modifying this variable in addition to the program being compiled. For example, an I/O device might need write directly into a program or data space. Meanwhile, the program itself may never directly access the memory area in question. In such a case, we would not want the compiler to optimize-out this data area that never seems to be used by the program, yet must exist for the program to function correctly in a larger context. It tells the compiler that the object is subject to sudden change for reasons which cannot be predicted from a study of the program itself, and forces every reference to such an object to be a genuine reference.
- const volatile - Both constant and volatile.

The "volatile" modifier

The volatile modifier is a directive to the compiler?s optimizer that operations involving this variable should not be optimized in certain ways. There are two special cases in which use of the volatile modifier is desirable. The first case involves memory-mapped hardware (a device such as a graphics adaptor that appears to the computer?s hardware as if it were part of the computer?s memory), and the second involves shared memory (memory used by two or more programs running simultaneously). Most computers have a set of registers that can be accessed faster than the computers main memory. A good compiler will perform a kind of optimization called redundant load and store removal. The compiler looks for places in the code where it can either remove an instruction to load data from memory because the value is already in a register, or remove an instruction to store data to memory because the value can stay in a register until it is changed again anyway.
If a variable is a pointer to something other than normal memory, such as memory-mapped ports on a peripheral, redundant load and store optimizations might be detrimental. For instance, here?s a piece of code that might be used to time some operation:

```
time_t time_addition(volatile const struct timer *t, int a)
{
  int n;
  int x;
  time_t then;
  x = 0;
  then = t->value;
  for (n = 0; n < 1000; n++)
  {
    x = x + a;
  }
  return t->value - then;
}
```

In this code, the variable t->value is actually a hardware counter that is being incremented as time passes. The function adds the value of a to x 1000 times, and it returns the amount the timer was incremented by while the 1000 additions were being performed. Without the volatile modifier, a clever optimizer might assume that the value of t does not change during the execution of the function, because there is no statement that explicitly changes it. In that case, there?s no need to read it from memory a second time and subtract it, because the answer will always be 0. The compiler might therefore ?optimize? the function by making it always return 0. If a variable points to data in shared memory, you also don?t want the compiler to perform redundant load and store optimizations. Shared memory is normally used to enable two programs to

communicate with each other by having one program store data in the shared portion of memory and the other program read the same portion of memory. If the compiler optimizes away a load or store of shared memory, communication between the two programs will be affected.

What does the typedef keyword do?

This keyword provides a short-hand way to write variable declarations. It is not a true data typing mechanism, rather, it is syntactic "sugar coating".

For example

```
typedef struct node
{
  int value;
  struct node *next;
}mynode;
```

This can later be used to declare variables like this

```
mynode *ptr1;
```

and not by the lengthy expression

```
struct node *ptr1;
```

There are three main reasons for using typedefs:

- It makes the writing of complicated declarations a lot easier. This helps in eliminating a lot of clutter in the code.
- It helps in achieving portability in programs. That is, if we use typedefs for data types that are machine dependent, only the typedefs need to change when the program is ported to a new platform.
- It helps in providing better documentation for a program. For example, a node of a doubly linked list is better understood as ptrToList than just a pointer to a complicated structure.

What is the difference between constants defined through #define and the constant keyword?

A constant is similar to a variable in the sense that it represents a memory location (or simply, a value). It is different from a normal variable, in that it cannot change it's value in the proram - it must stay for ever stay

constant. In general, constants are a useful because they can prevent program bugs and logical errors(errors are explained later). Unintended modifications are prevented from occurring. The compiler will catch attempts to reassign new values to constants.

Constants may be defined using the preprocessor directive #define. They may also be defined using the const keyword.

So whats the difference between these two?

```
#define ABC 5

and

const int abc = 5;
```

There are two main advantages of the second one over the first technique. First, the type of the constant is defined. "pi" is float. This allows for some type checking by the compiler. Second, these constants are variables with a definite scope. The scope of a variable relates to parts of your program in which it is defined.

There is also one good use of the important use of the const keyword. Suppose you want to make use of some structure data in some function. You will pass a pointer to that structure as argument to that function. But to make sure that your structure is readonly inside the function you can declare the structure argument as const in function prototype. This will prevent any accidental modification of the structure values inside the function

What are Trigraph characters?

These are used when you keyboard does not support some special characters

```
??=     #
??(     [
??)     ]
??<     {
??>     }
??!     |
??/     \
??'     ^
??-     ~
```

How are floating point numbers stored? Whats the IEEE format?

IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms.

IEEE floating point numbers have three basic components: the sign, the exponent, and the

mantissa. The mantissa is composed of the fraction and an implicit leading digit (explained below). The exponent base(2) is implicit and need not be stored.

The following figure shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

```
                  Sign   Exponent   Fraction   Bias
-------------------------------------------------
Single Precision 1 [31] 8 [30-23]  23 [22-00] 127
Double Precision 1 [63] 11 [62-52] 52 [51-00] 1023
```

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers. For double precision, the exponent field is 11 bits, and has a bias of 1023.

The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits. To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

```
5.00 × 100
0.05 × 10 ^ 2
5000 × 10 ^ -3
```

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0 × 100. A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

So, to sum up:

```
1. The sign bit is 0 for positive, 1 for negative.
2. The exponent's base is two.
3. The exponent field contains 127 plus the true exponent for single-
precision,
   or 1023 plus the true exponent for double precision.
4. The first bit of the mantissa is typically assumed to be 1.f, where
f is the
   field of fraction bits.
```

When should the register modifier be used?

The register modifier hints to the compiler that the variable will be heavily used and should be kept in the CPU?s registers, if possible, so that it can be accessed faster. There are several restrictions on the use of

the register modifier.

First, the variable must be of a type that can be held in the CPU?s register. This usually means a single value of a size less than or equal to the size of an integer. Some machines have registers that can hold floating-point numbers as well. Second, because the variable might not be stored in memory, its address cannot be taken with the unary & operator. An attempt to do so is flagged as an error by the compiler. Some additional rules affect how useful the register modifier is. Because the number of registers is limited, and because some registers can hold only certain types of data (such as pointers or floating-point numbers), the number and types of register modifiers that will actually have any effect are dependent on what machine the program will run on. Any additional register modifiers are silently ignored by the compiler. Also, in some cases, it might actually be slower to keep a variable in a register because that register then becomes unavailable for other purposes or because the variable isn?t used enough to justify the overhead of loading and storing it. So when should the register modifier be used? The answer is never, with most modern compilers. Early C compilers did not keep any variables in registers unless directed to do so, and the register modifier was a valuable addition to the language. C compiler design has advanced to the point, however, where the compiler will usually make better decisions than the programmer about which variables should be stored in registers. In fact, many compilers actually ignore the register modifier, which is perfectly legal, because it is only a hint and not a directive

When should a type cast be used?

There are two situations in which to use a type cast.

The first use is to change the type of an operand to an arithmetic operation so that the operation will be performed properly.

The second case is to cast pointer types to and from void * in order to interface with functions that expect or return void pointers. For example, the following line type casts the return value of the call to malloc() to be a pointer to a foo structure.

```
struct foo *p = (struct foo *) malloc(sizeof(struct foo));
```

A type cast should not be used to override a const or volatile declaration. Overriding these type modifiers can cause the program to fail to run correctly. A type cast should not be used to turn a pointer to one type of structure or data type into another. In the
rare events in which this action is beneficial, using a union to hold the values makes the programmers intentions clearer.

C Structures

Can structures be assigned to variables and passed to and from functions?

Yes, they can!

But note that when structures are passed, returned or assigned, the copying is done only at one level (The data pointed to by any pointer fields is not copied!.

Can we directly compare two structures using the == operator?

No, you cannot!

The only way to compare two structures is to write your own function that compares the structures field by field. Also, the comparison should be only on fields that contain data (You would not want to compare the next fields of each structure!).

A byte by byte comparison (say using memcmp()) will also fail. This is because the comparison might fonder on random bits present in unused "holes" in the structure (padding used to keep the alignment of the later fields correct). So a memcmp() of the two structure will almost never work. Also, any strings inside the strucutres must be compared using strcmp() for similar reasons.

There is also a very good reason why structures can be compared directly - unions!. It is because of unions that structures cannot be compared for equality. The possibility that a structure might contain a union makes it hard to compare such structures; the compiler can't tell what the union currently contains and so wouldn't know how to compare the structures. This sounds a bit hard to swallow and isn't 100% true, most structures don't contain unions, but there is also a philosophical issue at stake about just what is meant by "equality" when applied to structures. Anyhow, the union business gives the Standard a good excuse to avoid the issue by not supporting structure comparison.

If your structures dont have stuff like floating point numbers, pointers, unions etc..., then you could possibly do a memset() before using the structure variables..

```
memset (&myStruct, 0, sizeof(myStruct));
```

This will set the whole structure (including the padding) to all-bits-zero. We can then do a memcmp() on two such structures.

```
memcmp (&s1,&s2,sizeof(s1));
```

But this is very risky and can end up being a major bug in your code!. So try not to do this kind of memcmp() operations on structures variables as far as possible!

Can we pass constant values to functions which accept structure arguments?

If you are trying to do something like this

```
myfunction((struct mystruct){10,20});
```

then, it wont work!. Use a temporary structure variable.

How does one use fread() and fwrite()? Can we read/write structures to/from files?

Its easy to write a structure into a file using fwrite()

```
fwrite(&somestruct, sizeof somestruct, 1, fp);
```

A similat fread() invocation can read it back in. But, data files so written will not be portable (specially if they contain floating point numbers). Also that if the structure has any pointers, only the pointer values will be written, and they are most unlikely to be valid when read back in. One must use the "rb/wb" flag when opening the files.

A more portable solution is to write code to write and read a structure, field-by-field, in a portable (perhaps ASCII) way!. This is simpler to port and maintain.

Why do structures get padded? Why does sizeof() return a larger size?

Padding enables the CPU to access the members faster. If they are not aligned (say to word boundaries), then accessing them might take up more time. So the padding results in faster access. This is also required to ensure that alignment properties are preserved when an array of contiguous structures is allocated. Even if the structure is not part of an array, the end padding remains, so that sizeof() can always return a consistent size.

Can we determine the offset of a field within a structure and directly access that element?

The offsetof() macro(<stddef.h>) does exactly this.

A probable implementation of the macro is

```
#define offsetof(type, mem) ((size_t)((char *)&((type *)0)->mem -
 (char *)(type *)0))
```

This can be used as follows. The offset of field a in struct mystruct is

```
offset_of_a = offsetof(struct mystruct, a)
```

If structpointer is a pointer to an instance of this structure, and field a is an int, its value can be set indirectly with

```
*(int *)((char *)structpointer + offset_of_a) = some_value;
```

What are bit fields in structures?

To avoid wastage of memory in structures, a group of bits can be packed together into an integer and its called a bit field.

```
struct tag-name
{
    data-type name1:bit-length;
    data-type name2:bit-length;
    ...
    ...
    data-type nameN:bit-length;
}
```

A real example

```
struct student;
{
   char name[30];
   unsigned sex:1;
   unsigned age:5;
   unsigned rollno:7;
   unsigned branch:2;
};

struct student a[100];

scanf("%d", &sex);
a[i].sex=sex;
```

There are some limitations with respect to these bit fields, however:

```
1. Cannot scanf() directly into bit fields.
2. Pointers cannot be used on bit fields to access them.
3. Cannot have an array of bit fields.
```

The main use of bitfields is either to allow tight packing of data or to be able to specify the fields within some externally produced data files. C gives no guarantee of the ordering of fields within machine words, so if you do use them for the latter reason, you program will not only be non-portable, it will be compiler-dependent too. The Standard says that fields are packed into storage units, which are typically machine words. The packing order, and whether or not a bitfield may cross a storage unit boundary, is implementation defined. To force alignment to a storage unit boundary, a zero width field is used before the one that you want to have aligned. Be careful using them. It can require a surprising amount of run-time code to manipulate these things and you can end up using more space than they save. Bit fields do not have addresses?you can't have pointers to them or arrays of them.

What is a union? Where does one use unions? What are the limitations of unions?

A union is a variable type that can contain many different variables (like a structure), but only actually holds one of them at a time (not like a structure). This can save memory if you have a group of data where only one of the types is used at a time. The size of a union is equal to the size of it's largest data member. The C compiler allocates just enough space for the largest member. This is because only one member can be used at a time, so the size of the largest, is the most you will need. Here is an example:

```
union person
{
   int age;
   char name[100];
}person1;
```

The union above could be used to either store the age or it could be used to hold the name of the person. There are cases when you would want one or the other, but not both (This is a bad example, but you get the point). To access the fields of a union, use the dot operator(.) just as you would for a structure. When a value is assigned to one member, the other member(s) get whipped out since they share the same

memory. Using the example above, the precise time can be accessed like this:

```
person1.age;
```

In larger programs it may be difficult to keep track of which field is the currently used field. This is usually handled by using another variable to keep track of that. For example, you might use an integer called field. When field equals one, the age is used. If field is two, then name is used. The C compiler does no more than work out what the biggest member in a union can be and allocates enough storage (appropriately aligned if neccessary). In particular, no checking is done to make sure that the right sort of use is made of the members. That is your task, and you'll soon find out if you get it wrong. The members of a union all start at the same address?there is guaranteed to be no padding in front of any of them.

ANSI Standard C allows an initializer for the first member of a union. There is no standard way of initializing any other member (nor, under a pre-ANSI compiler, is there generally any way of initializing a union at all).

It is because of unions that structures cannot be compared for equality. The possibility that a structure might contain a union makes it hard to compare such structures; the compiler can't tell what the union currently contains and so wouldn't know how to compare the structures. This sounds a bit hard to swallow and isn't 100% true, most structures don't contain unions, but there is also a philosophical issue at stake about just what is meant by "equality" when applied to structures. Anyhow, the union business gives the Standard a good excuse to avoid the issue by not supporting structure comparison.

C Macros

How should we write a multi-statement macro?

This is the correct way to write a multi statement macro.

```
#define MYMACRO(argument1, argument2) do { \
                                    stament1; \
                                    stament2; \
                                   } while(1) /* no trailing semicolon */
```

How can I write a macro which takes a variable number of arguments?

One can define the macro with a single, parenthesized "argument" which in the macro expansion becomes the entire argument list, parentheses and all, for a function such as printf():

```
#define DEBUG(args) (printf("DEBUG: "), printf args)

if(n != 0) DEBUG(("n is %d\n", n));
```

The caller must always remember to use the extra parentheses. Other possible solutions are to use different macros depending on the number of arguments. C9X will introduce formal support for function-like macros with variable-length argument lists. The notation ... will appear at the end of the macro "prototype" (just as it does for varargs functions).

What is the token pasting operator and stringizing operator in C?

Token pasting operator

ANSI has introduced a well-defined token-pasting operator, ##, which can be used like this:

```
#define f(g,g2) g##g2

main()
{
  int var12=100;
  printf("%d",f(var,12));
}
```

O/P

100

Stringizing operator

```
#define sum(xy) printf(#xy " = %f\n",xy);

main()
{
   sum(a+b); // As good as printf("a+b = %f\n", a+b);
}
```

So what does the message "warning: macro replacement within a string literal" mean?

```
#define TRACE(var, fmt) printf("TRACE: var = fmt\n", var)
TRACE(i, %d);
```

gets expanded as

```
printf("TRACE: i = %d\n", i);
```

In other words, macro parameters were expanded even inside string literals and character constants. Macro expansion is *not* defined in this way by K&R or by Standard C. When you do want to turn macro

arguments into
strings, you can use the new # preprocessing operator, along with string literal concatenation:

```
#define TRACE(var, fmt) printf("TRACE: " #var " = " #fmt "\n", var)
```

See and try to understand this special application of the strigizing operator!

```
#define Str(x) #x
#define Xstr(x) Str(x)
#define OP plus

char *opname = Xstr(OP); //This code sets opname to "plus" rather than "OP".
```

Here are some more examples

Example1

Define a macro DEBUG such that the following program

```
int main()
{
  int x=4;
  float a = 3.14;
  char ch = 'A';

  DEBUG(x, %d);
  DEBUG(a, %f);
  DEBUG(ch, %c);
}
```

outputs

```
DEBUG: x=4
DEBUG: y=3.140000
DEBUG: ch=A
```

The macro would be

```
#define DEBUG(var, fmt) printf("DEBUG:" #var "=" #fmt "\n", var);
```

Example2

Write a macro PRINT for the following program

```
int main()
{
 int x=4, y=4, z=5;
 int a=1, b=2, c=3;
 PRINT(x,y,z);
 PRINT(a,b,c);
}
```

such that it outputs

x=4 y=4 z=5
a=1 b=2 c=3

Here is a macro that will do this
#define PRINT(v1,v2,v3) printf("\n" #v1 "=%d" #v2 "=%d" #v3 "=%d", v1, v2, v3);

Define a macro called SQR which squares a number.

This is the wrong way of doing it.

```
#define SQR(x) x*x
```

Thats because, something like

```
#include stdio.h>
#define SQR(x) x*x

int main()
{
  int i;
  i = 64/SQR(4);
  printf("[%d]",i);
  return(0);
}
```

will produce an output of 64, because of how macros work (The macro call square(4) will substituted by 4*4 so the expression becomes i = 64/4*4 . Since / and * has equal priority the expression will be evaluated as (64/4)*4 i.e. 16*4 = 64).

This is the right way of doing it.

```
#define SQR(x) ((x)*(x))
```

But, if x is an expression with side effects, the macro evaluates them twice. The following is one way to do it while evaluating x only once.

```
#include <math.h>
#define SQR(x) pow((x),2.0)
```

C Headers

What should go in header files? How to prevent a header file being included twice? Whats wrong with including more headers?

Generally, a header (.h) file should have (A header file need not have a .h extension, its just a convention):

```
1. Macro definitions (preprocessor #defines).
2. Structure, union, and enumeration declarations.
3. Any typedef declarations.
4. External function declarations.
5. Global variable declarations.
```

Put declarations / definitions in header files if they will be shared between several other files. If some of the definitions / declarations should be kept private to some .c file, don;t add it to the header files.

How does one prevent a header file from included twice?. Including header files twice can lead to multiple-definition errors.

There are two methods to prevent a header file from included twice

Method1

```
#ifndef HEADER_FILE
  #define HEADER_FILE
  ...header file contents...
#endif
```

Method2

A line like

```
#pragma once
```

inside a header file is an extension implemented by some preprocessors to help make header files idempotent (to prevent a header file from included twice).

So, what's the difference between #include <> and #include ""?

The <> syntax is typically used with Standard or system-supplied headers, while "" is typically used for a program's own header files. Headers named with <> syntax are searched for in one or more standard

places. Header files named with font class=announcelink>"" syntax are first searched for in the "current directory," then (if not found) in the same standard places.

What happens if you include unwanted headers? You will end up increasing the size of your executables!

Is there a limit on the number of characters in the name of a header file?

The limitation is only that identifiers be significant in the first six characters, not that they be restricted to six characters in length.

Is it acceptable to declare/define a variable in a C header?

A global variable that must be accessed from more than one file can and should be declared in a header file. In addition, such a variable must be defined in one source file. Variables should not be defined in header files, because the header file can be included in multiple source files, which would cause multiple definitions of the variable. The ANSI C standard will allow multiple external definitions, provided that there is only one initialization. But because there?s really no advantage to using this feature, it?s probably best to avoid it and maintain a higher level of portability. "Global" variables that do not have to be accessed from more than one file should be declared static and should not appear in a header file.

C File operations

How do stat(), fstat(), vstat() work? How to check whether a file exists?

The functions stat(), fstat(), lstat() are used to get the file status.

Here are their prototypes

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *file_name, struct stat *buf);
int fstat(int file_desc, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

These functions return information about the specified file. You do not need any access rights to the file to get this information but you need search rights to all directories named in the path leading to the file.

```
stat      -
     stats the file pointed to by file_name and fills in buf.
lstat     -
     identical to stat, except in the case of a symbolic link,
                where the link itself is stat-
ed, not the file that it refers to.
fstat     -
     identical to stat, only the open file pointed to by file_desc
                is stated in place of file_name.
```

They all return a stat structure, which contains the following fields:

```
              struct stat {
                  dev_t         st_dev;      /* device */
                  ino_t         st_ino;      /* inode */
                  mode_t        st_mode;     /* protection */
                  nlink_t       st_nlink;    /* number of hard links */
                  uid_t         st_uid;      /* user ID of owner */
                  gid_t         st_gid;      /* group ID of owner */
                  dev_t         st_rdev;     /* device type (if inode d
evice) */
                  off_t         st_size;     /* total size, in bytes */
                  unsigned long st_blksize;  /* blocksize for filesyste
m I/O */
                  unsigned long st_blocks;   /* number of blocks alloca
ted */
                  time_t        st_atime;    /* time of last access */
                  time_t        st_mtime;    /* time of last modificati
on */
                  time_t        st_ctime;    /* time of last change */
              };
```

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

Here is a small piece of code which returns the size of a file by accessing the st_size member of the stat structure.

```
boolean get_file_size(char *file_path,
                      int  *file_size)
{
  struct stat stat_buffer;

  if (stat((char *)file_path, &stat_buffer)!=0)
    return(FALSE);
```

```
  *file_size =  stat_buffer.st_size;
  return(TRUE);
}
```

So how do we check if a file exists or not?

Use functions like stat() as used above to find out if a file exits or not. Also, one can use fopen(). When using fopen(), just open for reading and close immediately. If the file does not exists, fopen() will given an error.

How can I insert or delete a line (or record) in the middle of a file?

The only way is to rewrite the file.

How can I recover the file name using its file descriptor?

Have a wrapper around fopen() to remember the names of files as you open them.

How can I delete a file? How do I copy files? How can I read a directory in a C program?

Deleting a file

The Standard C Library function is remove(). If thats not there, use unlink().

Copying a file

Directly use system() to invoke the operating system's copy() utility, or open the source and destination files, read characters or blocks of characters from the source file, and write them to the destination file.

How to read directories?

Use the opendir() and readdir() functions, which are part of the POSIX standard and are available on most Unix variants.

Whats the use of fopen(), fclose(), fprintf(), getc(), putc(), getw(), putw(), fscanf(), feof(), ftell(), fseek(), rewind(), fread(), fwrite(), fgets(), fputs(), freopen(), fflush(), ungetc()?

Whew!, thats a huge list.

fopen()

This function is used to open a stream.

```
FILE *fp;
fp = fopen("filename","mode");
fp = fopen("data","r");
fp = fopen("results","w");

Modes

    "r"     -> Open for reading.
    "w"     -> Open for writing.
    "a"     -> Open for appending.
    "r+"    -> Both reading and writing.
    "w+"    -> Both reading and writing, create new file if it exists,
    "a+"    -> Open for both reading and appending.
```

fopen()

fclose() is used to close a stream .

```
fclose(fp);
```

putc(), getc(), putw(), getw(), fgetc(), getchar(), putchar(), fputs()

These functions are used to read/write different types of data to the stream.

```
putc(ch,fp);
c=getc(fp);
putw(integer, fp);
integer=getw(fp);
```

fprintf(), fscanf()

Read/Write formatted data from/to the stream.

```
fprintf(fp,"control string",list);
fscanf(fp,"control string", list);
```

feof()

Check the status of a stream

```
if(feof(fp)!=0)
```

ftell(), fseek(), rewind(), fgetpos(), fsetpos()

Reposition the file pointer of a stream

```
n=ftell(fp); //Relative offset (in bytes) of the current position.
rewind(fp);

fseek(fp, offset, position);

Position can be

0->start of file
1->current position
2->end of file

fseek(fp,0L,0);  // Same as rewind.
fseek(fp,0L,1);  // Stay at current position.
fseek(fp,0L,2);  // Past end of file.
fseek(fp,m,0);   // Move to (m+1) byte.
fseek(fp,m,1)    // Go forward m bytes.
fseek(fp,-m,1);  // Go backward m bytes from current position.
fseek(fp,-m,2);  // Go backward from end of file.
```

fread(), fwrite()

Binary stream input/output.

```
fwrite(&customer, sizeof(record),1,fp);
fread(&customer,sizeof(record),1,fp);
```

Here is a simple piece of code which reads from a file

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

int main()
{

 FILE *f;
 char buffer[1000];
 f=fopen("E:\\Misc\\__Temp\\FileDrag\\Main.cpp","r");
 if(f)
 {
  printf("\nOpenened the file!\n");
  while(fgets(buffer,1000,f))
  {
   printf("(%d)-> %s\n",strlen(buffer),buffer);
  }
 }

 fclose(f);
 getch();
 return(0);
}
```

How to check if a file is a binary file or an ascii file?

Here is some sample C code. The idea is to check the bytes in the file to see if they are ASCII or not...

```
#include <stdio.h>

int main(int argc, char *argv[])
{
 unsigned char ch;
 FILE *file;
 int binaryFile = FALSE;

 file = fopen(<FILE_PATH>, "rb");            // Open in Binary mode for
 the first time.


 while((fread(&ch, 1, 1, file) == 1) && (binaryFile == FALSE))
 {
    if(ch < 9 || ch == 11 || (ch > 13 && ch < 32) || ch == 255)
    {
```

```
      binaryFile = 1;
   }
}

fclose(file);

if(binaryFile)
   file = fopen(<FILE_PATH>, "rb");
else
   file = fopen(<FILE_PATH>, "r");


if(binaryFile)
{
   while(fread(&ch, 1, 1, file) == 1)
   {
     // Do whatever you want here with the binary file byte...
   }
}
else
{
   while(fread(&ch, 1, 1, file) == 1)
   {
     // This is ASCII data, can easily print it!
     putchar(ch);
   }
}

fclose(file);

return(0);
}
```

C Declarations and Definitions

What is the difference between char *a and char a[]?

There is a lot of difference!

```
char a[] = "string";
char *a = "string";
```

The declaration char a[] asks for space for 7 characters and see that its known by the name "a". In contrast, the declaration char *a, asks for a place that holds a pointer, to be known by the name "a". This pointer "a" can point anywhere. In this case its pointing to an anonymous array of 7 characters, which does have any name in particular. Its just present in memory with a pointer keeping track of its location.

```
char a[] = "string";

   +----+----+----+----+----+----+------+
a: | s  |  t |  r |  i |  n |  g | '\0' |
   +----+----+----+----+----+----+------+
   a[0] a[1] a[2] a[3] a[4] a[5] a[6]


char *a = "string";

+-----+          +---+---+---+---+---+---+------+
|  a: | *======> | s | t | r | i | n | g | '\0' |
+-----+          +---+---+---+---+---+---+------+
Pointer          Anonymous array
```

It is curcial to know that a[3] generates different code depending on whether a is an array or a pointer. When the compiler sees the expression a[3] and if a is an array, it starts at the location "a", goes three elements past it, and returns the character there. When it sees the expression a[3] and if a is a pointer, it starts at the location "a", gets the pointer value there, adds 3 to the pointer value, and gets the character pointed to by that value.

If a is an array, a[3] is three places past a. If a is a pointer, then a[3] is three places past the memory location pointed to by a. In the example above, both a[3] and a[3] return the same character, but the way they do it is different!

Doing something like this would be illegal.

```
char *p = "hello, world!";
p[0] = 'H';
```

How can I declare an array with only one element and still access elements beyond the first element (in a valid fashion)?

This is a trick question :)

There is a way to do this. Using structures.

```
struct mystruct {
  int  value;
  int length;
  char string[1];
};
```

Now, when allocating memory to the structure using malloc(), allocate more memory than what the structure would normally require!. This way, you can access beyond string[0] (till the extra amount of memory you have allocated, ofcourse).
But remember, compilers which check for array bounds carefully might throw warnings. Also, you need to have a length field in the structure to keep a count of how big your one element array really is :).

A cleaner way of doing this is to have a pointer instead of the one element array and allocate memory for it seperately after allocating memory for the structure.

```
struct mystruct {
```

```
  int  value;
  char *string;  // Need to allocate memory using malloc() after alloca
ting memory for the strucure.};
```

What is the difference between enumeration variables and the preprocessor #defines?

Functionality	Enumerations	#defines
Numeric values assigned automatically?	YES	NO
Can the debugger display the symbolic values?	YES	NO
Obey block scope?	YES	NO
Control over the size of the variables?	NO	NO

C Functions - built-in

Whats the difference between gets() and fgets()? Whats the correct way to use fgets() when reading a file?

Unlike fgets(), gets() cannot be told the size of the buffer it's to read into, so it cannot be prevented from overflowing that buffer. As a general rule, always use fgets(). fgets() also takes in the size of the buffer, so that the end of the array cannot be overwritten.

```
fgets(buffer, sizeof(buffer), stdin);
if((p = strchr(buffer, '\n')) != NULL)
{
  *p = '\0';
}
```

Also, fgets() does not delete the trailing "\n", as gets().

So whats the right way to use fgets() when reading a file?

```
while(!feof(inp_file_ptr))
```

```
{
  fgets(buffer, MAX_LINE_SIZE, inp_file_ptr);
  fputs(buffer, out_file_ptr);
}
```

The code above will copy the last line twice! This is because, in C, end-of-file is only indicated after an input routine has tried to read, and failed. We should just check the return value of the input routine (in this case, fgets() will return NULL on end-of-file); often, you don't need to use feof() at all.

This is the right way to do it

```
while(fgets(buffer, MAX_LINE_SIZE, inp_file_ptr))
{
  fputs(buffer, out_file_ptr);
}
```

How can I have a variable field width with printf?

Use something like

```
printf("%*d", width, x);
```

Here is a C program...

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

#define WIDTH 5

int main ( void )
{
    char str1[] = "Good Boy";
    char str2[] = "The earth is round";

    int width = strlen ( str1 )  + WIDTH;
    int prec  = strlen ( str2 )  + WIDTH;

    printf ( "%*.*s\n", width, prec, str1 );
    return 0;
}
```

How can I specify a variable width in a scanf() format string?

You cant!.

An asterisk in a scanf() format string means to suppress assignment. You may be able to use ANSI stringizing and string concatenation to accomplish about the same thing, or you can construct the scanf format string at run time.

How can I convert numbers to strings (the opposite of atoi)?

Use sprintf()

Note!, since sprintf() is also a variable argument function, it fails badly if passed with wrong arguments or if some of the arguments are missed causing segmentation faults. So be very careful when using sprintf() and pass the right number and type of arguments to it!

Why should one use strncpy() and not strcpy()? What are the problems with strncpy()?

strcpy() is the cause of many bugs. Thats because programmers almost always end up copying more data into a buffer than it can hold. To prevent this problem, strncpy() comes in handy as you can specify the exact number of bytes to be copied.

But there are problems with strncpy() also. strncpy() does not always place a '\0' terminator in the destination string. (Funnily, it pads short strings with multiple \0's, out to the specified length). We must often append a '\0' to the destination string by hand. You can get around the problem by using strncat() instead of strncpy(): if the destination string starts out empty, strncat() does what you probably wanted strncpy() to do. Another possibility is sprintf(dest, "%.*s", n, source). When arbitrary bytes (as opposed to strings) are being copied, memcpy() is usually a more appropriate function to use than strncpy().

How does the function strtok() work?

strtok() is the only standard function available for "tokenizing" strings.

The strtok() function can be used to parse a string into tokens. The first call to strtok() should have the string as its first argument. Subsequent calls should have the first argument set to NULL. Each call returns a pointer to the next token, or NULL when no more tokens are found. If a token ends with a delimiter, this delimiting character is overwritten with a "\0" and a pointer to the next character is saved for the next call to strtok(). The delimiter string delim may be different for each call.

The strtok_r() function works the same as the strtok() function, but instead of using a static buffer it uses a pointer to a user allocated char* pointer. This pointer must be the same while parsing the same string.

An example

```
main()
{
```

```
    char str[]="This is a test";
    char *ptr[10];
    char *p;
    int i=1;
    int j;

    p=strtok(str," ");

    if(p!=NULL)
    {
        ptr[0]=p;
        while(1)
        {
            p=strtok(NULL, " ");

            if(p==NULL)break;
            else
            {
               ptr[i]=p;i++;
            }
        }
    }

    for(j=0;j<i;j++)
    {
            printf("\n%s\n", ptr[j]);
    }
}
```

Why do we get the floating point formats not linked error?

Some compilers leave out certain floating point support if it looks like it will not be needed. In particular, the non-floating-point versions of printf() and scanf() save space by not including code to handle %e, %f, and %g. It happens that Borland's heuristics for determining whether the program uses floating point are insufficient, and the programmer must sometimes insert a dummy call to a floating-point library function (such as sqrt(); any will do) to force loading of floating-point support

Why do some people put void cast before each call to printf()?

printf() returns a value. Some compilers (and specially lint) will warn about return values not used by the program. An explicit cast to (void) is a way of telling lint that you have decided to ignore the return value from this specific function call. It's also common to use void casts on calls to strcpy() and strcat(), since the return value is never used for anything useful.

What is assert() and when would I use it?

An assertion is a macro, defined in <assert.h>, for testing assumptions. An assertion is an assumption made by the programmer. If its violated, it would be caught by this macro.

For example

```
int i,j;

for(i=0;i<=10;i++)
{
  j+=5;
  assert(i<5);
}

Runtime error: Abnormal program termination.
assert failed (i<5), <file name>,<line number>
```

If anytime during the execution, i gets a value of 0, then the program would break into the assertion since the assumption that the programmer made was wrong.

What do memcpy(), memchr(), memcmp(), memset(), strdup(), strncat(), strcmp(), strncmp(), strcpy(), strncpy(), strlen(), strchr(), strchr(), strpbrk(), strspn(), strcspn(), strtok() do?

Here are the prototypes...

```
   void *memchr(const void *s, int c, size_t n);
   int memcmp(const void *s1, const void *s2, size_t n);
   void *memcpy(void * restrict s1, const void * restrict s2, size_t n)
;
   void *memmove(void *s1, const void *s2, size_t n);
   void *memset(void *s, int c, size_t n);
   char *strcat(char *restrict s1, const char *restrict s2);
   char *strchr(const char *s, int c);
   int strcmp(const char *s1, const char *s2);
   char *strcpy(char *restrict s1, const char *restrict s2);
   size_t strcspn(const char *s1, const char *s2);
   char *strdup(const char *s1);
   size_t strlen(const char *s);
   char *strncat(char *restrict s1, const char *restrict s2, size_t n);
   int strncmp(const char *s1, const char *s2, size_t n);
   char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
   char *strpbrk(const char *s1, const char *s2);
   char *strrchr(const char *s, int c);
   size_t strspn(const char *s1, const char *s2);
   char *strstr(const char *s1, const char *s2);
   char *strtok(char *restrict s1, const char *restrict s2);
```

The memchr() function shall locate the first occurrence of c (converted to an unsigned char) in the initial n bytes (each interpreted as unsigned char) of the object pointed to by s. The memchr() function shall return a pointer to the located byte, or a null pointer if the byte does not occur in the object.

The memcmp() function shall compare the first n bytes (each interpreted as unsigned char) of the object pointed to by s1 to the first n bytes of the object pointed to by s2. The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ in the objects being compared. The memcmp() function shall return an integer greater than, equal to, or less than 0, if the object pointed to by s1 is greater than, equal to, or less than the object pointed to by s2, respectively.

The memcpy() function shall copy n bytes from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined. The memcpy() function returns s1; no value is reserved to indicate an error. The memmove() function shall copy n bytes from the object pointed to by s2 into the object pointed to by s1. Copying takes place as if the n bytes from the object pointed to by s2 are first copied into a temporary of n bytes that does not overlap the objects pointed to by s1 and s2, and then the n bytes from the temporary array are copied into the object pointed to by s1. The memmove() function returns s1; no value is reserved to indicate an error.

The memset() function shall copy c (converted to an unsigned char) into each of the first n bytes of the object pointed to by s. The memset() function returns s; no value is reserved to indicate an error.

The strcat() function shall append a copy of the string pointed to by s2 (including the terminating null byte) to the end of the string pointed to by s1. The initial byte of s2 overwrites the null byte at the end of s1. If copying takes place between objects that overlap, the behavior is undefined. The strcat() function shall return s1; no return value is reserved to indicate an error.

The strchr() function shall locate the first occurrence of c (converted to a char) in the string pointed to by s. The terminating null byte is considered to be part of the string. Upon completion, strchr() shall return a pointer to the byte, or a null pointer if the byte was not found.

The strcmp() function shall compare the string pointed to by s1 to the string pointed to by s2. The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ in the strings being compared. Upon completion, strcmp() shall return an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2, respectively.

The strcpy() function shall copy the string pointed to by s2 (including the terminating null byte) into the array pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined. The strcpy() function returns s1; no value is reserved to indicate an error.

The strcspn() function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by s1 which consists entirely of bytes not from the string pointed to by s2. The strcspn() function shall return the length of the computed segment of the string pointed to by s1; no return value is reserved to indicate an error.

The strdup() function shall return a pointer to a new string, which is a duplicate of the string pointed to by s1, if memory can be successfully allocated for the new string. The returned pointer can be passed to free (). A null pointer is returned if the new string cannot be created. The function may set errno to ENOMEM if the allocation failed.

The strlen() function shall compute the number of bytes in the string to which s points, not including the terminating null byte. The strlen() function shall return the length of s; no return value shall be reserved to indicate an error.

The strncat() function shall append not more than n bytes (a null byte and bytes that follow it are not appended) from the array pointed to by s2 to the end of the string pointed to by s1. The initial byte of s2 overwrites the null byte at the end of s1. A terminating null byte is always appended to the result. If copying

takes place between objects that overlap, the behavior is undefined. The strncat() function shall return s1; no return value shall be reserved to indicate an error.

The strncmp() function shall compare not more than n bytes (bytes that follow a null byte are not compared) from the array pointed to by s1 to the array pointed to by s2. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ in the strings being compared. Upon successful completion, strncmp() shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by s1 is greater than, equal to, or less than the possibly null-terminated array pointed to by s2 respectively.

The strncpy() function shall copy not more than n bytes (bytes that follow a null byte are not copied) from the array pointed to by s2 to the array pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined. If the array pointed to by s2 is a string that is shorter than n bytes, null bytes shall be appended to the copy in the array pointed to by s1, until n bytes in all are written. The strncpy() function shall return s1; no return value is reserved to indicate an error.

The strpbrk() function shall locate the first occurrence in the string pointed to by s1 of any byte from the string pointed to by s2. Upon successful completion, strpbrk() shall return a pointer to the byte or a null pointer if no byte from s2 occurs in s1.

The strrchr() function shall locate the last occurrence of c (converted to a char) in the string pointed to by s. The terminating null byte is considered to be part of the string. Upon successful completion, strrchr() shall return a pointer to the byte or a null pointer if c does not occur in the string.

The strspn() function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by s1 which consists entirely of bytes from the string pointed to by s2. The strspn() function shall return the computed length; no return value is reserved to indicate an error.

The strstr() function shall locate the first occurrence in the string pointed to by s1 of the sequence of bytes (excluding the terminating null byte) in the string pointed to by s2. Upon successful completion, strstr() shall return a pointer to the located string or a null pointer if the string is not found. If s2 points to a string with zero length, the function shall return s1.

A sequence of calls to strtok() breaks the string pointed to by s1 into a sequence of tokens, each of which is delimited by a byte from the string pointed to by s2. The first call in the sequence has s1 as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by s2 may be different from call to call. The first call in the sequence searches the string pointed to by s1 for the first byte that is not contained in the current separator string pointed to by s2. If no such byte is found, then there are no tokens in the string pointed to by s1 and strtok() shall return a null pointer. If such a byte is found, it is the start of the first token. The strtok() function then searches from there for a byte that is contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by s1, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The strtok() function saves a pointer to the following byte, from which the next search for a token shall start. Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above. Upon successful completion, strtok() shall return a pointer to the first byte of a token. Otherwise, if there is no token, strtok() shall return a null pointer.

What does alloca() do?

alloca() allocates memory which is automatically freed when the function which called alloca() returns. However, note that alloca() is not portable and its usage is not recommended.

Can you compare two strings like string1==string2? Why do we need strcmp()?

Do you think this will work?

```
if(string1 == string2)
{

}
```

No!, strings in C cannot be compared like that!.

The == operator will end up comparing two pointers (that is, if they have the same address). It wont compare the contents of those locations. In C, strings are represented as arrays of characters, and the language never manipulates (assigns, compares, etc.) arrays as a whole.

The correct way to compare strings is to use strcmp()

```
if(strcmp(string1, string2) == 0)
{

}
```

What does printf() return?

Upon a successful return, the printf() function returns the number of characters printed (not including the trailing '\0' used to end output to strings). If the output was truncated due to this limit then the return value is the number of characters (not including the trailing '\0') which would have been written to the final string if enough space had been available. Thus, a return value of size or more means that the output was truncated. If an output error is encountered, a negative value is returned.

There is a very funny yet interesting interview question around this....

Look at the code below.

```
void main()
{
   if(X)
  {
```

```
      printf("Hello");
   }
   else
   {
      printf(" World");
   }
}
```

What should X be replaced with inorder to get the output as "Hello World"?

And here comes the answer....

```
#include <stdio.h>
int main()
{
    if(!printf("Hello"))
    {
        printf("Hello");
    }
    else
    {
        printf(" World");
    }
}
```

Kind of stupid isn't it? But they do ask these type of questions. Believe me!

What do setjmp() and longjump() functions do?

A goto statement implements a local jump of program execution, and the longjmp() and setjmp() functions implement a nonlocal, or far, jump of program execution. Generally, a jump in execution of any kind should be avoided because it is not considered good programming practice to use such statements as goto and longjmp in your program. A goto statement simply bypasses code in your program and jumps to a predefined position. To use the goto statement, you give it a labeled position to jump to. This predefined position must be within the same function. You cannot implement gotos between functions.

When your program calls setjmp(), the current state of your program is saved in a structure of type jmp_buf. Later, your program can call the longjmp() function to restore the program?s state as it was when you called setjmp().Unlike the goto statement, the longjmp() and setjmp() functions do not need to be implemented in the same function. However, there is a major drawback to using these functions: your program, when restored to its previously saved state, will lose its references to any dynamically allocated memory between the longjmp() and the setjmp(). This means you will waste memory for every malloc() or calloc() you have implemented between your longjmp() and setjmp(), and your program will be horribly inefficient. It is highly recommended that you avoid using functions such as longjmp() and setjmp() because they, like the goto statement, are quite often an indication of poor programming practice.

It is not possible to jump from one function to another by means of a goto and a label, since labels have only

function scope. However, the macro setjmp and function longjmp provide an alternative, known as a non-local goto, or a non-local jump. The header file <setjmp.h> declares something called a jmp_buf, which is used by the cooperating macro and function to store the information necessary to make the jump. The declarations are as follows:

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

The setjmp macro is used to initialise the jmp_buf and returns zero on its initial call. Then, it returns again, later, with a non-zero value, when the corresponding longjmp call is made! The non-zero value is whatever value was supplied to the call of longjmp.

This is best explained by way of an example:

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

void func(void);
jmp_buf place;

main()
{
   int retval;

    /*
     * First call returns 0,
     * a later longjmp will return non-zero.
     */

     if(setjmp(place)) != 0)
     {
        printf("Returned using longjmp\n");
        exit(EXIT_SUCCESS);
     }

     /*
      * This call will never return - it
      * 'jumps' back above.
      */

     func();
     printf("What! func returned!\n");
}

void func(void)
{
     /*
      * Return to main.
      * Looks like a second return from setjmp,
      * returning 4!
      */
```

```
    longjmp(place, 4);
    printf("What! longjmp returned!\n");
}
```

The val argument to longjmp is the value seen in the second and subsequent ?returns? from setjmp. It should normally be something other than 0; if you attempt to return 0 via longjmp, it will be changed to 1. It is therefore possible to tell whether the setjmp was called directly, or whether it was reached by calling longjmp. If there has been no call to setjmp before calling longjmp, the effect of longjmp is undefined, almost certainly causing the program to crash. The longjmp function is never expected to return, in the normal sense, to the instructions immediately following the call. All accessible objects on ?return? from setjmp have the values that they had when longjmp was called, except for objects of automatic storage class that do not have volatile type; if they have been changed between the setjmp and longjmp calls, their values are indeterminate.

The longjmp function executes correctly in the contexts of interrupts, signals and any of their associated functions. If longjmp is invoked from a function called as a result of a signal arriving while handling another signal, the behaviour is undefined.

It's a serious error to longjmp to a function which is no longer active (i.e. it has already returned or another longjump call has transferred to a setjmp occurring earlier in a set of nested calls).

The Standard insists that, apart from appearing as the only expression in an expression statement, setjmp may only be used as the entire controlling expression in an if, switch, do, while, or for statement. A slight extension to that rule is that as long as it is the whole controlling expression (as above) the setjmp call may be the subject of the ! operator, or may be directly compared with an integral constant expression using one of the relational or equality operators. No more complex expressions may be employed.

Examples are:

```
setjmp(place);                    /* expression statement */
if(setjmp(place)) ...             /* whole controlling expression */
if(!setjmp(place)) ...            /* whole controlling expression */
if(setjmp(place) < 4) ...         /* whole controlling expression */
if(setjmp(place)<;4 && 1!=2) ...  /* forbidden */
```

C Functions - The main function

Whats the prototype of main()? Can main() return a structure?

The right declaration of main() is

```
int main(void)
```

or

```
int main(int argc, char *argv[])
```

or

```
int main(int argc, char *argv[], char *env[]) //Compiler dependent, non
-standard C.
```

In C, main() cannot return anything other than an int.Something like

```
void main()
```

is illegal. There are only three valid return values from main() - 0, EXIT_SUCCESS, and EXIT_FAILURE, the latter two are defined in <stdlib.h>.

Something like this can cause unpredicted behavior

```
struct mystruct {
  int value;
  struct mystruct *next;
}

main(argc, argv)
{ ... }
```

Here the missing semicolon after the structure declaration causes main to be misdeclared.

Is exit(status) equivalent to returning the same status from main()?

No.

The two forms are not equivalent in a recursive call to main().

Can main() be called recursively?

Yes

```
main()
{
    main();
}
```

But this will go on till a point where you get a

```
Runtime error : Stack overflow.
```

How to print the arguments recieved by main()?

Here is some code

```
main(int argc, char*argv[])
{
  int i;
  for(i=0; i<argc; i++)
  {
    printf("\n[%s]\n", argv[i]);
  }
}
```

OS Concepts

What do the system calls fork(), vfork(), exec(), wait(), waitpid() do? Whats a Zombie process? Whats the difference between fork() and vfork()?

The system call fork() is used to create new processes. It does not take any arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller (which is called the parent). After a new child process is created, both processes will execute the next instruction following the fork() system call. We can distinguish the parent from the child by testing the returned value of fork():

If fork() returns a negative value, the creation of a child process was unsuccessful. A call to fork() returns a zero to the newly created child process and the same call to fork() returns a positive value (the process ID of the child process) to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process. Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes will have separate address spaces. Both processes start their execution right after the system call fork(). Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names.

Trick question!

How many processes are forked by the below program

```
main()
{
  fork();
  fork();
}
```

Answer is 2 raise to n, when n is the number of calls to fork (2 in this case).

Each process has certain information associated with it including:

```
The UID (numeric user identity)
The GID (numeric group identity)
A process ID number used to identify the process
A parent process ID
The execution status, e.g. active, runnable, waiting for input etc.
```

```
Environment variables and values.
The current directory.
Where the process currently resides in memory
The relative process priority see nice(1)
Where it gets standard input from
Where it sends standard output to
Any other files currently open
```

Certain process resources are unique to the individual process. A few among these are:

- Stack Space: which is where local variables, function calls, etc. are stored.
- Environment Space: which is used for storage of specific environment variables.
- Program Pointer (counter) : PC.
- File Descriptors
- Variables

Undex unix, each sub directory under /proc corresponds to a running process (PID #). A ps provide detailed information about processes running

A typical output of ps looks as follows:

```
         PID               TTY            STAT           TIME         C
OMMAND
         ------------------------------------------------------------
------
         8811              3              SW             0:00         (
login)
         3466              3              SW             0:00         (
bash)
         8777              3              SW             0:00         (
startx)
            .              .               .               .          .

         1262              p7              R              0:00         p
s
```

The columns refer to the following:

```
PID      - The process id's (PID).
TTY      - The terminal the process was started from.
STAT     -
 The current status of all the processes. Info about the process status
          can be broken into more than 1 field. The first of these fie
lds can
          contain the following entries:

            R        - Runnable.
            S        - Sleeping.
```

```
            D         - Un-interuptable sleep.
            T         - Stopped or Traced.
            Z         - Zombie Process.

The second field can contain the following entry:

            W       - If the process has no residual pages.

And the third field:

            N       - If the process has a positive nice value.
            TIME    - The CPU time used by the process so far.
            COMMAND - The actual command.
```

The init process is the very first process run upon startup. It starts additional processes. When it runs it reads a file called /etc/inittab which specifies init how to set up the system, what processes it should start with respect to specific runlevels. One crucial process which it starts is the getty program. A getty process is usually started for each terminal upon which a user can log into the system. The getty program produces the login: prompt on each terminal and then waits for activity. Once a getty process detects activity (at a user attempts to log in to the system), the getty program passes control over to the login program.

There are two command to set a process's priority nice and renice.

One can start a process using the system() function call

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

int main()
{

  printf("Running ls.....\n");
  system("ls -lrt");
  printf("Done.\n");
  exit(0);
}
```

The exec() system call

The exec() functions replace a current process with another created according to the arguments given.

The syntax of these functions is as follows:

```
#include <unistd.h>

char *env[];

int  execl(const char *path, const char *arg0, ..., (char *)0);
int  execv(const char *path, const char *argv[]);

int execlp(const char *path, const char *arg0, ..., (char *)0);
```

```
int execvp(const char *path, const char *argv[]);

int execle(const char *path, const char *arg0, ... , (char *)0, const c
har *env[]);
int execve(const char *path, const char *argv[], const char *env[]);
```

The program given by the path argument is used as the program to execute in place of what is currently running. In the case of the execl() the new program is passed arguments arg0, arg1, arg2,... up to a null pointer. By convention, the first argument supplied (i.e. arg0) should point to the file name of the file being executed. In the case of the execv() programs the arguments can be given in the form of a pointer to an array of strings, i.e. the argv array. The new program starts with the given arguments appearing in the argv array passed to main. Again, by convention, the first argument listed should point to the file name of the file being executed. The function name suffixed with a p (execlp() and execvp())differ in that they will search the PATH environment variable to find the new program executable file. If the executable is not on the path, and absolute file name, including directories, will need to be passed to the function as a parameter. The global variable environ is available to pass a value for the new program environment. In addition, an additional argument to the exec() functions execle() and execve() is available for passing an array of strings to be used as the new program environment.

Examples to run the ls command using exec are:

```
const char *argv[] = ("ls", "-lrt", 0);
const char *env[] = {"PATH=/bin:/usr/bin", "TERM=console", 0};

execl("/bin/ls", "ls", "-lrt", 0);
execv("/bin/ls", argv);

execlp("ls", "ls", "-lrt", 0);
execle("/bin/ls", "ls", "-lrt", 0, env);

execvp("ls", argv);
execve("/bin/ls", argv, env);
```

A simple call to fork() would be something like this

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  pid_t pid;
  pid=fork();
  switch(pid)
  {
     case -1: exit(1); // fork() error.
     case  0: // Child process, can call exec here.
              break;
     default: // Parent.
              break;
  }
  exit(0);
}
```

The call wait() can be used to determine when a child process has completed it's job and finished. We can arrange for the parent process to wait untill the child finishes before continuing by calling wait(). wait() causes a parent process to pause untill one of the child processes dies or is stopped. The call returns the PID of the child process for which status information is available. This will usually be a child process which has terminated. The status information allows the parent process to determine the exit status of the child process, the value returned from main or passed to exit. If it is not a null pointer the status information will be written to the location pointed to by stat_loc. We can interrogate the status information using macros defined in sys/wait.h.

```
Macro                   Definition
-------------------------------------------------------------------------
------------
WIFEXITED(stat_val);    Nonzero if the child is terminated normally
WEXITSTATUS(stat_val);  If WIFEXITED is nonzero, this returns child exi
t code.
WIFSIGNALLED(stat_val); Nonzero if the child is terminated on an uncaug
ht signal.
WTERMSIG(stat_val);     If WIFSIGNALLED is nonzero, this returns a sign
al number.
WIFSTOPPED(stat_val);   Nonzero if the child stopped on a signal.
WSTOPSIG(stat_val);     If WIFSTOPPED is nonzero, this returns a signal
 number.
```

An example code which used wait() is shown below

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void){
    pid_t child_pid;
    int *status=NULL;
    if(fork()){
        /* wait for child, getting  PID */
        child_pid=wait(status);
        printf("I'm the parent.\n");
        printf("My child's PID was: %d\n",child_pid);
    } else {
        printf("I'm the child.\n");
    }
    return 0;
}
```

Or a more detailed program

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
```

```
int main()
{

  pid_t pid;
  int exit_code;

  pid = fork();
  switch(pid)
  {
    case -1: exit(1);
    case 0:  exit_code = 11; //Set the child exit process
             break;
    default: exit_code = 0;
             break;
  }

  if (pid)
  {
     // This is the parent process
     int status;
     pid_t child_pid;

     child_pid = wait(&status);

     printf("Child process finished with PID [%d]\n", child_pid);
     if (WIFEXITED(status))
     {
       printf("Child exited with code [%d]\n", WEXITSTATUS(status));
     }
     else
     {
       printf("Child terminated abnormally.\n");
     }
   }

   exit(exit_code);
}
```

So now, whats a Zombie process?

When using fork() to create child processes it is important to keep track of these processes. For instance, when a child process terminates, an association with the parent survives untill the parent either terminates normally or calls wait(). The child process entry in the process table is not freed up immediately. Although it is no longer active, the child process is still in the system because it's exit code needs to be stored in the even the parent process calls wait(). The child process is at that point referred to as a zombie process. Note, if the parent terminates abnormally then the child process gets the process with PID 1, (init) as parent. (such a child process is often referred to as an orphan). The child process is now a zombie. It is no longer running, it's origional parent process is gone, and it has been inherited by init. It will remain in the process table as a zombie untill the next time the table is processed. If the process table is long this may take a while. till init cleans them up. As a general rule, program wisely and try to avoid zombie processes. When zobbies accumulate they eat up valuable resources.

The waitpid() system call is another call that can be used to wait for child processes. This system call however can be used to wait for a specific process to terminate.

```
#include <sys/types.h>
#include <sys/wait.h>
pit_t waitpid(pid_t pid, int *status, int options);
```

The pid argument specifies the PID of the particular child process to wait for. If it is a -1 then waitpid() will return information to the child process. Status information will be written to the location pointed to by status. The options argument enables us to change the behavior of waitpid(). A very usefull oprion is WNOHANG which prevents the call to waitpid() from suspending the execution of the caller. We can it to find out whether any child process has terminated and, if not, to continue.

Synchronous and asynchronous process execution

In some cases, for example if the child process is a server or "daemon" (a process expected to run all the time in the background to deliver services such as mail forwarding) the parent process would not wait for the child to finish. In other cases, e.g. running an interactive command where it is not good design for the parent's and child's output to be mixed up into the same output stream, the parent process, e.g. a shell program, would normally wait for the child to exit before continuing. If you run a shell command with an ampersand as it's last argument, e.g. sleep 60 & the parent shell doesn't wait for this child process to finish.

So whats the difference between fork() and vfork()?

The system call vfork(), is a low overhead version of fork(), as fork() involves copying the entire address space of the process and is therefore quite expensive. The basic difference between the two is that when a new process is created with vfork(), the parent process is temporarily suspended, and the child process might borrow the parent's address space. This strange state of affairs continues until the child process either exits, or calls execve(), at which point the parent process continues. This means that the child process of a vfork() must be careful to avoid unexpectedly modifying variables of the parent process. In particular, the child process must not return from the function containing the vfork() call, and it must not call exit() (if it needs to exit, it should use _exit(); actually, this is also true for the child of a normal fork()).

However, since vfork() was created, the implementation of fork() has improved , most notably with the introduction of `copy-on-write', where the copying of the process address space is transparently faked by allowing both processes to refer to the same physical memory until either of them modify it. This largely removes the justification for vfork(); indeed, a large proportion of systems now lack the original functionality of vfork() completely. For compatibility, though, there may still be a vfork() call present, that simply calls fork() without attempting to emulate all of the vfork() semantics.

How does freopen() work?
What are threads? What is a lightweight process? What is a heavyweight process? How different is a thread from a process?
How are signals handled?
What is a deadlock?
What are semaphores?
What is meant by context switching in an OS?

What is Belady's anomaly?

Usually, on increasing the number of frames allocated to a process' virtual memory, the process execution is faster, because fewer page faults occur. Sometimes, the reverse happens, i.e., the execution time increases

even when more frames are allocated to the process. This is known as the Belady's Anomaly. This is true for certain page reference patterns. This is also called the FIFO anomaly.

What is thrashing?

What are short-, long- and medium-term scheduling?

Long term scheduler determines which programs are admitted to the system for processing. It controls the degree of multiprogramming. Once admitted, a job becomes a process.

Medium term scheduling is part of the swapping function. This relates to processes that are in a blocked or suspended state. They are swapped out of main-memory until they are ready to execute. The swapping-in decision is based on memory-management criteria.

Short term scheduler, also know as a dispatcher executes most frequently, and makes the finest-grained decision of which process should execute next. This scheduler is invoked whenever an event occurs. It may lead to interruption of one process by preemption.

What are turnaround time and response time?

Turnaround time is the interval between the submission of a job and its completion. Response time is the interval between submission of a request, and the first response to that request.

What is the Translation Lookaside Buffer (TLB)?

In a cached system, the base addresses of the last few referenced pages is maintained in registers called the TLB that aids in faster lookup. TLB contains those page-table entries that have been most recently used. Normally, each virtual memory reference causes 2 physical memory accesses-- one to fetch appropriate page-table entry, and one to fetch the desired data. Using TLB in-between, this is reduced to just one physical memory access in cases of TLB-hit.

What is cycle stealing?

We encounter cycle stealing in the context of Direct Memory Access (DMA). Either the DMA controller can use the data bus when the CPU does not need it, or it may force the CPU to temporarily suspend operation. The latter technique is called cycle stealing. Note that cycle stealing can be done only at specific break points in an instruction cycle.

What is a reentrant program?

Re-entrancy is a useful, memory-saving technique for multiprogrammed timesharing systems. A Reentrant Procedure is one in which multiple users can share a single copy of a program during the same period.

Reentrancy has 2 key aspects:

- The program code cannot modify itself.
- The local data for each user process must be stored separately.

Thus, the permanent part is the code, and the temporary part is the pointer back to the calling program and local variables used by that program. Each execution instance is called activation. It executes the code in the permanent part, but has its own copy of local variables/parameters. The temporary part associated with each activation is the activation record. Generally, the activation record is kept on the stack.
Note: A reentrant procedure can be interrupted and called by an interrupting program, and still execute correctly on returning to the procedure.

When is a system in safe state?

The set of dispatchable processes is in a safe state if there exist at least one temporal order in which all processes can be run to completion without resulting in a deadlock.

What is busy waiting?

The repeated execution of a loop of code while waiting for an event to occur is called busy-waiting. The CPU is not engaged in any real productive activity during this period, and the process does not progress toward completion

What is pages replacement? What are local and global page replacements?

What is meant by latency, transfer and seek time with respect to disk I/O?

Seek time is the time required to move the disk arm to the required track. Rotational delay or latency is the time to move the required sector to the disk head. Sums of seek time (if any) and the latency is the access time, for accessing a particular track in a particular sector. Time taken to actually transfer a span of data is transfer time.

What are monitors? How are they different from semaphores?

In the context of memory management, what are placement and replacement algorithms?

Placement algorithms determine where in the available main-memory to load the incoming process. Common methods are first-fit, next-fit, and best-fit.

Replacement algorithms are used when memory is full, and one process (or part of a process) needs to be swapped out to accommodate the new incoming process. The replacement algorithm determines which are the partitions (memory portions occupied by the processes) to be swapped out.

What is paging? What are demand- and pre-paging?

What is mounting?

Mounting is the mechanism by which two different file systems can be combined together. This is one of the services provided by the operating system, which allows the user to work with two different file systems, and some of the secondary devices.

What do you mean by dispatch latency?

The time taken by the dispatcher to stop one process and start running another process is known as the dispatch latency.

What is multi-processing? What is multi-tasking? What is multi-threading? What is multi-programming?

What is compaction?

Compaction refers to the mechanism of shuffling the memory portions such that all the free portions of the memory can be aligned (or merged) together in a single large block. OS to overcome the problem of fragmentation, either internal or external, performs this mechanism, frequently. Compaction is possible only if relocation is dynamic and done at run-time, and if relocation is static and done at assembly or load-time compaction is not possible.

What is memory-mapped I/O? How is it different frim I/O mapped I/O?

Memory-mapped I/O, meaning that the communication between the I/O devices and the processor is done through physical memory locations in the address space. Each I/O device will occupy some locations in the I/O address space. I.e., it will respond when those addresses are placed on the bus. The processor can write those locations to send commands and information to the I/O device and read those locations to get information and status from the I/O device. Memory-mapped I/O makes it easy to write device drivers in a high-level language as long as the high-level language can load and store from arbitrary addresses.

List out some reasons for process termination.

- Normal completion
- Time limit exceeded
- Memory unavailable
- Bounds violation
- Protection error
- Arithmetic error
- Time overrun
- I/O failure
- Invalid instruction
- Privileged instruction
- Data misuse
- Operator or OS intervention
- Parent termination.

General Concepts

What is the difference between statically linked libraries and dynamically linked libraries (dll)?

Static linking means all the referenced code is included in the binary produced. When linking statically, the linker finds the bits that the program modules need, and physically copies them into the executable output file that it generates.

Incase of dynamic linking, the binary simply contains a pointer to the needed routine, which will be loaded by the OS as needed. An important benefit is that libraries can be updated to fix bugs or security problems and the binaries that use them are immediately using the fixed code. It also saves disk space, saves memory usage, saves launch time, improve multi-tasking efficiency. But, deployment of dynamically-linked executable generally requires coordination across hosts of shared-object libraries, installation locations, installation environment settings, settings for compile-, link-, and use-time environment variables Dynamic linking often precludes relocations backward across OS versions.

On Linux, static libraries have names like libname.a, while shared libraries are called libname.so.x.y.z where x.y.z is some form of version number. Shared libraries often also have links pointing to them, which are important, and (on a.out configurations) associated .sa files. The standard libraries come in both shared and static formats. You can find out what shared libraries a program requires by using ldd (List Dynamic Dependencies)

Dynamic linking allows an exe or dll to use required information at run time to call a DLL function. In static linking, the linker gets all the referenced functions from the static link library and places it with your code into your executable. Using DLLs instead of static link libraries makes the size of the executable file smaller. Dynamic linking is faster than static linking.

What are the most common causes of bugs in C?

Murphy's law states that "If something can go wrong, it will. So is the case with programming. Here are a few things than can go wrong when coding in C.

- Typing if(x=0) instead of if(x==0). Use if(0==x) instead.
- Using strcpy() instead of strncpy() and copying more memory than the buffer can hold.
- Dereferencing null pointers.
- Improper malloc(), free() usage. Assuming malloc'ed memory contains 0, assuming freed storage persists even after freeing it, freeing something twice, corrupting the malloc() data structures.
- Uninitialized local variables.
- Integer overflow.
- Mismatch between printf() format and arguments, especially trying to print long ints using %d. Problems with wrong arguments to sprintf() leading to core dumps.
- Array out of bounds problems, especially of small, temporary buffers.
- Leaving files opened.
- Undefined evaluation order, undefined statements.
- Omitted declaration of external functions, especially those which return something other than int, or have
  "narrow" or variable arguments.
- Floating point problems.
- Missing function prototypes.

What is hashing?

Hashing is the process of mapping strings to integers, in a relatively small range.

A hash function maps a string (or some other data structure) to a bounded number (called the hash bucket) which can more easily be used as an index in an array, or for performing repeated comparisons.

A mapping from a potentially huge set of strings to a small set of integers will not be unique. Any algorithm using hashing therefore has to deal with the possibility of collisions.

Here are some common methods used to create hashing functions

```
Direct method
Subtraction method
```

```
Modulo-Division method
Digit-Extraction method
Mid-Square method
Folding method
Pseudo-random method
```

Here is a simple implementation of a hashing scheme in C

```
#include <stdio.h>
#define HASHSIZE 197

typedef struct node
{
  int value;
  struct node *next;
}mynode;


typedef struct hashtable
{
  mynode *llist;
}HT;


HT myHashTable[HASHSIZE];


int main()
{
  initialize();
  add(2);
  add(2500);
  add(199);

  display_hash();
  getch();
  return(0);
}

int initialize()
{
  int i;
  for(i=0;i<HASHSIZE;i++)
     myHashTable[i].llist=NULL;
  return(1);
}

int add(int value)
{
  int hashkey;
  int i;
  mynode *tempnode1, *tempnode2, *tempnode3;

  hashkey=value%HASHSIZE;
```

```
  printf("\nHashkey : [%d]\n", hashkey);

  if(myHashTable[hashkey].llist==NULL)
   {
      //This hash bucket is empty, add the first element!
      tempnode1 = malloc(sizeof(mynode));
      tempnode1->value=value;
      tempnode1->next=NULL;
      myHashTable[hashkey].llist=tempnode1;
   }
   else
   {
      //This hash bucket already has some items. Add to it at the end.
      //Check if this element is already there?

      for(tempnode1=myHashTable[hashkey].llist;
          tempnode1!=NULL;
          tempnode3=tmpnode1,tempnode1=tempnode1->next)
      {
         if(tempnode1->value==value)
         {
            printf("\nThis value [%d] already exists in the Hash!\n", v
alue);
            return(1);
         }
    }

    tempnode2 = malloc(sizeof(mynode));
    tempnode2->value = value;
    tempnode2->next=NULL;
    tempnode3->next=tempnode2;
  }
  return(1);
}


int display_hash()
{
  int i;
  mynode *tempnode;

  for(i=0;i<HASHSIZE;i++)
  {
     if(myHashTable[i].llist==NULL)
     {
        printf("\nmyHashTable[%d].llist -> (empty)\n",i);
     }
     else
     {
        printf("\nmyHashTable[%d].llist -> ",i);
        for(tempnode=myHashTable[i].llist;tempnode!=NULL;tempnode=tempn
ode->next)
        {
           printf("[%d] -> ",tempnode->value);
        }
        printf("(end)\n");
```

```
      }
      if(i%20==0)getch();
  }
  return(0);
}
```

And here is the output

```
Hashkey : [2]

Hashkey : [136]

Hashkey : [2]

Hashkey : [2]

This value [2] already exists in the Hash!

myHashTable[0].llist -> (empty)

myHashTable[1].llist -> (empty)

myHashTable[2].llist -> [2] -> [199] -> (end)

myHashTable[3].llist -> (empty)

myHashTable[4].llist -> (empty)

myHashTable[5].llist -> (empty)

myHashTable[6].llist -> (empty)

myHashTable[7].llist -> (empty)

myHashTable[8].llist -> (empty)

myHashTable[9].llist -> (empty)

myHashTable[10].llist -> (empty)

myHashTable[11].llist -> (empty)

myHashTable[12].llist -> (empty)

myHashTable[13].llist -> (empty)

myHashTable[14].llist -> (empty)

myHashTable[15].llist -> (empty)

myHashTable[16].llist -> (empty)

myHashTable[17].llist -> (empty)
```

```
myHashTable[18].llist -> (empty)

myHashTable[19].llist -> (empty)

myHashTable[20].llist -> (empty)

...

myHashTable[133].llist -> (empty)

myHashTable[134].llist -> (empty)

myHashTable[135].llist -> (empty)

myHashTable[136].llist -> [2500] -> (end)

myHashTable[137].llist -> (empty)

...

myHashTable[196].llist -> (empty)
```

What do you mean by Predefined?

It means already written, compiled and linked together with our program at the time of linking.

What is data structure?

A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.

What are the time complexities of some famous algorithms?

Here you are

```
Binary search                                   :    O(logn)
```

```
Finding max & min for a given set of numbers :    O(3n/2-2)
Merge Sort                                   :    O(nlogn)
Insertion Sort                               :    O(n*n)
Quick Sort                                   :    O(nlogn)
Selection Sort                               :    O(n*n)
```

What is row major and column major form of storage in matrices?

This is row major representation

If this is your matrix

```
a b c d
e f g h
i j k l
```

Convert into 1D array by collecting elements by rows. Within a row elements are collected from left to right. Rows are collected from top to bottom. So, x[i][j] is mapped to position i*no_of_columns + j.

This is column major representation

On similar lines, in column major representation, we store elements column wise. Here, x[i][j] is mapped to position i + j * no_of_rows.

Explain the BigOh notation.

Time complexity

The Big Oh for the function f(n) states that

```
f(n)=O(g(n)); iff there exist positive constants c and d such that: f(n
) <=c*g(n) for all n,n>=d.
```

Here are some examples...

```
1       ->Constant
logn    ->Logarithmic
n       ->Linear
```

```
nlogn   ->nlogn
n*n     ->quadratic
n*n*n   ->cubic
2^n     ->exponential
n!      ->Factorial
```

We also have a few other notations...

Omega notation

```
f(n)=Omega(g(n)), iff positive constants c and d exist such as f(n)>=cg
(n) for all n, n>=d.
```

Theta notation

```
f(n)=Theta(g(n)), iff positive constants c1 and c2 and an d exists such
 that c1g(n)<=f(n)<=c2g(n) for all n, n>=d.
```

Little Oh(o)

```
f(n)=og(n), iff f(n)=Og(n) and f(n)!=Omega(g(n)).
```

Then there is also something called Space complexity...

Fixed part : This is the independent of the instance characteristics. This typically includes the instruction space (i.e. space for the code), space for simple variables and fixed-size component variables, space for constants, and so on.

Variable part: A variable part that consists of the space needed by the component variables whose size depends on the particular problem instance being solved, dynamically allocated space (to the extent that this space depends on the instance characteristics); and the recursion stack space (in so far as this space depends on the instance characteristics).

A complete description of these concepts is out of the scope of this website. There are plenty of books and millions of pages on the Internet. Help yourself!

Give the most important types of algorithms.

There are five important basic designs for algorithms.

They are:

- Divide and conquer : For a function to compute on n inputs the divide and conquer strategy suggests the inputs into a k distinct subsets, 1<k<=n, yielding k sub-problems. These sub-problems

must be solved and then a method must be found to combine the sub-solutions into a solution of the whole. An example for this approach is "binary search" algorithm. The time complexity of binary search algorithm is O(log n).

- The greedy method : The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is an optimal solution. An example for solution using greedy method is "knapsack problem". Greedy algorithms attempt not only to find a solution, but to find the ideal solution to any given problem.
- Dynamic programming : Dynamic Programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions. An example for algorithm using dynamic programming is "multistage graphs". This class remembers older results and attempts to use this to speed the process of finding new results.
- Back-tracking : Here if we reach a dead-end, we use the stack to pop-off the dead end and try something else we had not tried before. The famous 8-queens problem uses back tracking. Backtracking algorithms test for a solution, if one is found the algorithm has solved, if not it recurs once and tests again, continuing until a solution is found.
- Branch and bound : Branch and bound algorithms form a tree of subproblems to the primary problem, following each branch until it is either solved or lumped in with another branch.

What is marshalling and demarshalling?

When objects in memory are to be passed across a network to another host or persisted to storage, their in-memory representation must be converted to a suitable out-of-memory format. This process is called marshalling, and converting back to an in memory representation is called demarshalling. During marshalling, the objects must be respresented with enough information that the destination host can understand the type of object being created. The objects? state data must be converted to the appropriate format. Complex object trees that refer to each other via object references (or pointers) need to refer to each other via some form of ID that is independent of any memory model. During demarshalling, the destination host must reverse all that and must also validate that the objects it receives are consistent with the expected object type (i.e. it validate that it doesn?t get a string where it expects a number).

What is the difference between the stack and the heap? Where are the different types of variables of a program stored in memory?

When a program is loaded into memory, it is organized into three areas of memory, called segments: the text segment, stack segment, and the heap segment. The text segment (sometimes also called the code segment) is where the compiled code of the program itself resides. This is the machine language representation of the program steps to be carried out, including all functions making up the program, both user defined and system.

The remaining two areas of system memory is where storage may be allocated by the compiler for data storage. The stack is where memory is allocated for automatic variables within functions. A stack is a Last In First Out (LIFO) storage device where new storage is allocated and deallocated at only one "end", called the Top of the stack. When a program begins executing in the function main(), space is allocated on the stack for all variables declared within main(). If main() calls a function, func(), additional storage is allocated for the variables in func() at the top of the stack. Notice that the parameters passed by main() to func() are also stored on the stack. If func() were to call any additional functions, storage would be allocated at the new Top

of stack. When func() returns, storage for its local variables is deallocated, and the Top of the stack returns to its old position. If main() were to call another function, storage would be allocated for that function at the Top. The memory allocated in the stack area is used and reused during program execution. It should be clear that memory allocated in this area will contain garbage values left over from previous usage.

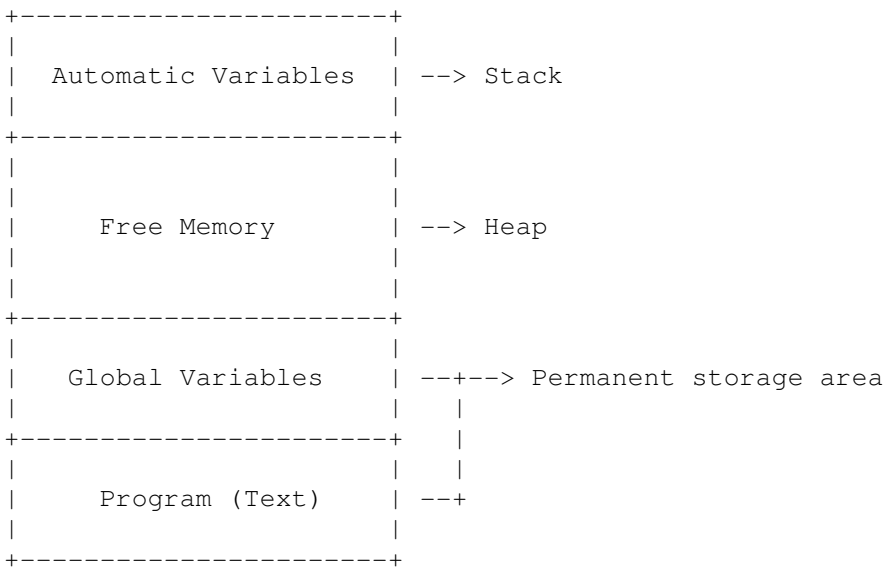
The heap segment provides more stable storage of data for a program; memory allocated in the heap remains in existence for the duration of a program. Therefore, global variables (storage class external), and static variables are allocated on the heap. The memory allocated in the heap area, if initialized to zero at program start, remains zero until the program makes use of it. Thus, the heap area need not contain garbage.

Describe the memory map of a C program.

This is a quite popular question. But I have never understood what exactly is the purpose of asking this question. The memory map of a C program depends heavily on the compiler used.

Nevertheless, here is a simple explanation..

During the execution of a C program, the program is loaded into main memory. This area is called permanent storage area. The memory for Global variables and static variables is also allocated in the permanent storage area. All the automatic variables are stored in another area called the stack. The free memory area available between these two is called the heap. This is the memory region available for dynamic memory allocation during the execution of a program.



What is infix, prefix, postfix? How can you convert from one representation to another? How do you evaluate these expressions?

There are three different ways in which an expression like a+b can be represented.

Prefix (Polish)

```
+ab
```

Postfix (Suffix or reverse polish)

```
ab+
```

Infix

```
a+b
```

Note than an infix expression can have parathesis, but postfix and prefix expressions are paranthesis free expressions.

Conversion from infix to postfix

Suppose this is the infix expresssion

```
((A + (B - C) * D) ^ E + F)
```

To convert it to postfix, we add an extra special value] at the end of the infix string and push [onto the stack.

```
((A + (B - C) * D) ^ E + F)]

--------->
```

We move from left to right in the infix expression. We keep on pushing elements onto the stack till we reach a operand. Once we reach an operand, we add it to the output. If we reach a) symbol, we pop elements off the stack till we reach a corresponding { symbol. If we reach an operator, we pop off any operators on the stack which are of higher precedence than this one and push this operator onto the stack.

As an example

```
Expresssion                          Stack                    Output
-------------------------------------------------------------------------
-----
((A + (B - C) * D) ^ E + F)]      [
^

((A + (B - C) * D) ^ E + F)]      [((                        A
  ^
```

```
((A + (B - C) * D) ^ E + F)]       [((+                          A
     ^

((A + (B - C) * D) ^ E + F)]       [((+(-
                          ABC
              ^

((A + (B - C) * D) ^ E + F)]       [(                            ABC-D*+
                    ^

((A + (B - C) * D) ^ E + F)]       [(                            ABC-
D*+E^F+
                             ^

((A + (B - C) * D) ^ E + F)]       [                             ABC-
D*+E^F+
                               ^
```

Is there a way to find out if the converted postfix expression is valid or not

Yes. We need to associate a rank for each symbol of the expression. The rank of an operator is -1 and the rank of an operand is +1. The total rank of an expression can be determined as follows:

```
-
 If an operand is placed in the post fix expression, increment the rank
 by 1.
-
 If an operator is placed in the post fix expression, decrement the ran
k by 1.
```

At any point of time, while converting an infix expression to a postfix expression, the rank of the expression can be greater than or equal to one. If the rank is anytime less than one, the expression is invalid. Once the entire expression is converted, the rank must be equal to 1. Else the expression is invalid.

Conversion from infix to prefix

This is very similar to the method mentioned above, except the fact that we add the special value [at the start of the expression and] to the stack and we move through the infix expression from right to left. Also at the end, reverse the output expression got to get the prefix expression.

Evaluation of a postfix expression

Here is the pseudocode. As we scan from left to right

- If we encounter an operand, push it onto the stack.
- If we encounter an operator, pop 2 operand from the stack. The first one popped is called operand2 and the second one is called operand1.
- Perform Result=operand1 operator operand2.

- Push the result onto the stack.
- Repeat the above steps till the end of the input.

Convert from postfix expression to infix

This is the same as postfix evaluation, the only difference being that we wont evaluate the expression, but just present the answer. The scanning is done from left to right.

```
ABC-D*+

A(B-C)D*+

A((B-C)*D)+

A+((B-C)*D)
```

Convert from postfix expression to infix

The scanning is done from right to left and is similar to what we did above.

```
+A*-BCD

Reverse it

DCB-*A+

D(B-C)*A+

((B-C)*D)A+

A+((B-C)*D)
```

How can we detect and prevent integer overflow and underflow?

This questions will be answered soon :)

Compiling and Linking

How to list all the predefined identifiers?

gcc provides a -dM option which works with -E.

How the compiler make difference between C and C++?

Most compilers recognized the file type by looking at the file extension.

You might also be able to force the compiler to ignore the file type by supplying compiler switch. In MS VC++ 6, for example, the MSCRT defines a macro, __cplusplus. If you undefine that macro, then the compiler will treat your code as C code. You don't define the __cplusplus macro. It is defined by the compiler when compiling a C++ source. In MSVC6 there's a switch for the compiler, /Tc, that forces a C compilation instead of C++.

What are the general steps in compilation?

```
1. Lexical analysis.
2. Syntactic analysis.
3. Sematic analysis.
4. Pre-optimization of internal representation.
5. Code generation.
6. Post optimization.
```

What are the different types of linkages?

Linkage is used to determine what makes the same name declared in different scopes refer to the same thing. An object only ever has one name, but in many cases we would like to be able to refer to the same object from different scopes. A typical example is the wish to be able to call printf() from several different places in a program, even if those places are not all in the same source file.

The Standard warns that declarations which refer to the same thing must all have compatible type, or the behaviour of the program will be undefined. Except for the use of the storage class specifier, the declarations must be identical.

The three different types of linkage are:

- external linkage
- internal linkage

- no linkage

In an entire program, built up perhaps from a number of source files and libraries, if a name has external linkage, then every instance of a that name refers to the same object throughout the program. For something which has internal linkage, it is only within a given source code file that instances of the same name will refer to the same thing. Finally, names with no linkage refer to separate things.

Linkage and definitions

Every data object or function that is actually used in a program (except as the operand of a sizeof operator) must have one and only one corresponding definition. This "exactly one" rule means that for objects with external linkage there must be exactly one definition in the whole program; for things with internal linkage (confined to one source code file) there must be exactly one definition in the file where it is declared; for things with no linkage, whose declaration is always a definition, there is exactly one definition as well.

The three types of accessibility that you will want of data objects or functions are:

- Throughout the entire program,
- Restricted to one source file,
- Restricted to one function (or perhaps a single compound statement).

For the three cases above, you will want external linkage, internal linkage, and no linkage respectively. The external linkage declarations would be prefixed with extern, the internal linkage declarations with static.

```
#include <stdio.h>

// External linkage.
extern int var1;

// Definitions with external linkage.
extern int var2 = 0;

// Internal linkage:
static int var3;

// Function with external linkage
void f1(int a){}

// Function can only be invoked by name from within this file.
static int f2(int a1, int a2)
{
    return(a1 * a2);
}
```

What do you mean by scope and duration?

The duration of an object describes whether its storage is allocated once only, at program start-up, or is more transient in its nature, being allocated and freed as necessary.

There are only two types of duration of objects: static duration and automatic duration. Static duration means that the object has its storage allocated permanently, automatic means that the storage is allocated and freed as necessary.

It's easy to tell which is which: you only get automatic duration if

- The declaration is inside a function.
- And the declaration does not contain the static or extern keywords.
- And the declaration is not the declaration of a function.

The scope of the names of objects defines when and where a given name has a particular meaning. The different types of scope are the following:

- function scope
- file scope
- block scope
- function prototype scope

The easiest is function scope. This only applies to labels, whose names are visible throughout the function where they are declared, irrespective of the block structure. No two labels in the same function may have the same name, but because the name only has function scope, the same name can be used for labels in every function. Labels are not objects?they have no storage associated with them and the concepts of linkage and duration have no meaning for them. Any name declared outside a function has file scope, which means that the name is usable at any point from the declaration on to the end of the source code file containing the declaration. Of course it is possible for these names to be temporarily hidden by declarations within compound statements. As we know, function definitions must be outside other functions, so the name introduced by any function definition will always have file scope. A name declared inside a compound statement, or as a formal parameter to a function, has block scope and is usable up to the end of the associated } which closes the compound statement. Any declaration of a name within a compound statement hides any outer declaration of the same name until the end of the compound statement. A special and rather trivial example of scope is function prototype scope where a declaration of a name extends only to the end of the function prototype. The scope of a name is completely independent of any storage class specifier that may be used in its declaration.

What are makefiles? Why are they used?

This questions will be answered soon :)

Gotcha! programs

Here are a few Gotcha! programs worth looking at....

- ```
  int main()
  {
    int const * p=5;
    printf("%d",++(*p));
  }
  ```

  Answer: "Cannot modify a constant value". p is a pointer to a "constant integer". But we tried to change the value of the "constant integer".

- ```
  int main()
  {
    char s[ ]="man";
    int i;
    for(i=0;s[ i ];i++)
    {
      printf("\n%c%c%c%c",s[ i ],*(s+i),*(i+s),i[s]);
    }
  }
  ```

  Answer:
  mmmm
  aaaa
  nnnn

  Thanks because, s[i], *(i+s), *(s+i), i[s] are all different ways of expressing the same idea. Generally array name is the base address for that array. Here s is the base address. i is the index number/displacement from the base address. So, indirecting it with * is same as s[i]. i[s] may be surprising. But in the case of C it is same as s[i].

- ```
  int main()
  {
    static int var = 5;
    printf("%d ",var--);
  ```
  ```

```
  if(var)
   main();
  return(0);
}
```

Answer: "5 4 3 2 1". When static storage class is given, it is initialized once. The change in the value of a static variable is retained even between the function calls. Main is also treated like any other ordinary function, which can be called recursively.

-

```
int main()
{
  extern int i;
  i=20;
  printf("%d",i);
  return(0);
}
```

Answer: "Linker Error : Undefined symbol '_i'". The extern storage class in the following declaration, extern int i; specifies to the compiler that the memory for i is allocated in some other program and that address will be given to the current program at the time of linking. But linker finds that no other variable of name i is available in any other program with memory space allocated for it. Hence a linker error has occurred .

-

```
int main()
{
  char string[]="Hello World";
  display(string);
  return(0);
}

void display(char *string)
{
  printf("%s",string);
}
```

Answer: "Compiler Error : Type mismatch in redeclaration of function display". In third line, when the function display is encountered, the compiler doesn't know anything about the function display. It assumes the arguments and return types to be integers, (which is the default type). When it sees the actual function display, the arguments and type contradicts with what it has assumed previously. Hence a compile time error occurs.

-

```
int main()
{
  struct xx
  {
   char name[]="hello";
  };
  struct xx *s;
  printf("%s",s->name);
  return(0);
```

```
}
```

Answer: "Compiler Error". You should not initialize variables in structure declaration.

-

```
int main()
{
  printf("%p",main);
}
```

Answer: Some address will be printed. Function names are just addresses (just like array names are addresses).main() is also a function. So the address of function main will be printed. %p in printf specifies that the argument is an address. They are printed as hexadecimal numbers.

-

```
enum colors {BLACK,BLUE,GREEN};
int main()
{
  printf("%d..%d..%d",BLACK,BLUE,GREEN);
  return(0);
}
```

Answer: "0..1..2". The enum tag assigns numbers starting from 0, if not explicitly defined.

-

```
#include <stdio.h>
void fun();

int main()
{
  int i=1;
  while (i<=5)
  {
     printf("%d",i);
     if (i>2)
     {
       goto here;
     }
     i++;
  }
}

void fun()
{
here:
  printf("PP");
}
```

Answer: "Compiler error: Undefined label 'here' in function main". Labels have functions scope, in other words The scope of the labels is limited to functions . The label 'here' is available in function fun() Hence it is not visible in function main.

-

```
#include <stdio.h>
int main()
{
  static char names[5][20]={"pascal","ada","cobol","fortran","per
l"};
  int i;
  char *t;

  t        = names[3];
  names[3] = names[4];
  names[4] = t;
  for (i=0;i<=4;i++)
  {
    printf("%s",names[i]);
  }
}
```

Answer: "Compiler error: Lvalue required in function main". Array names are pointer constants. So it cannot be modified.

-

```
#include <stdio.h>

int main()
{
  int i=1,j=2;
  switch(i)
  {
     case 1: printf("GOOD");
             break;
     case j: printf("BAD");
             break;
  }
}
```

Answer: "Compiler Error: Constant expression required in function main." The case statement can have only constant expressions (this implies that we cannot use variable names directly so an error). Note that enumerated types can be used in case statements.

-

```
int main()
{
  int i;
  printf("%d",scanf("%d",&i)); // value 10 is given as input here
  return(0);
}
```

Answer: "1". Scanf returns number of items successfully read.Here 10 is given as input which should have been scanned successfully. So number of items read is 1.

-

```
#define f(g,g2) g##g2

int main()
{
  int var12=100;
  printf("%d",f(var,12));
  return(0);
}
```

Answer: "100".

-

```
int main()
{
  extern int i;
  i=20;
  printf("%d",sizeof(i));
  return(0);
}
```

Answer: "Linker error: undefined symbol '_i'.". The extern declaration specifies that the variable i is defined somewhere else. The compiler passes the external variable to be resolved by the linker. So compiler doesn't find an error. During linking the linker searches for the definition of i. Since it is not found the linker flags an error.

-

```
#include <stdio.h>
int main()
{
  extern out;
  printf("%d", out);
  return(0);
}
int out=100;
```

Answer: "100". This is the correct way of writing the previous program.

-

```
int i,j;
for(i=0;i<=10;i++)
{
  j+=5;
  assert(i<5);
}
```

Answer: "Runtime error: Abnormal program termination. assert failed (i<5), <file name>,<line number>". The assert() statements are used during debugging to make sure that certain conditions are satisfied. If assertion fails, the program will terminate reporting the same. After debugging, use #undef NDEBUG and this will disable all the assertions from the source code. Assertion is a good debugging tool to make use of.

-

```
int main()
{
  main();
  return(0);
}
```

Answer: "Runtime error : Stack overflow.". The main function calls itself again and again. Each time the function is called its return address is stored in the call stack. Since there is no condition to terminate the function call, the call stack overflows at runtime. So it terminates the program and results in an error.

-

```
int main()
{
  char *cptr,c;
  void *vptr,v;
  c=10;
  v=0;
  cptr=&c;
  vptr=&v;
  printf("%c%v",c,v);
  return(0);
}
```

Answer: "Compiler error (at line number 4): size of v is Unknown." You can create a variable of type void * but not of type void, since void is an empty type. In the second line you are creating variable vptr of type void * and v of type void hence an error.

-

```
#include <stdio.h>
int i=10;
int main()
{
  extern int i;
  {
     int i=20;
     {
          const volatile unsigned i=30;
          printf("%d",i);
     }
     printf("%d",i);
  }
  printf("%d",i);
  return(0);
}
```

Answer: "30,20,10". The '{' introduces new block and thus new scope. In the innermost block i is declared as, const volatile unsigned which is a valid declaration. i is assumed of type int. So printf prints 30. In the next block, i has value 20 and so printf prints 20. In the outermost block, i is declared as extern, so no storage space is allocated for it. After compilation is over the linker

resolves it to global variable i (since it is the only variable visible there). So it prints i's value as 10.

-

```
int main()
{
  const int i=4;
  float j;
  j = ++i;
  printf("%d %f", i,++j);
}
```

Answer: "Compiler error". Here, i is a constant. you cannot change the value of constant

-

```
void aaa()
{
  printf("hi");
}

void bbb()
{
  printf("hello");
}

void ccc()
{
  printf("bye");
}

int main()
{
  int (*ptr[3])();
  ptr[0]=aaa;
  ptr[1]=bbb;
  ptr[2]=ccc;
  ptr[2]();
}
```

Answer: "bye". Here, ptr is array of pointers to functions of return type int.ptr[0] is assigned to address of the function aaa. Similarly ptr[1] and ptr[2] for bbb and ccc respectively. ptr[2]() is in effect of writing ccc(), since ptr[2] points to ccc.

-

```
int main()
{
  int * j;
  void fun(int **);
  fun(&j);
  return(0);
}

void fun(int **k)
```

```
{
  int a =0;
  /* add a stmt here*/
}
```

Answer : "*k = &a;". The argument of the function is a pointer to a pointer.In the following pgm add a stmt in the function fun such that the address of 'a' gets stored in 'j'.

-

```
int abc(int a,float b)
{
  /* some code */
}

int abc(a,b)
int a; float b;
{
  /* some code*/
}
```

Answer: The first one is called the "ANSI C notation" and the second one the "Kernighan & Ritche notation".

-

```
int main()
{
  static int i=5;
  if(--i)
  {
    main();
    printf("%d ",i);
  }
}
```

Answer: "0 0 0 0". The variable "I" is declared as static, hence memory for I will be allocated for only once, as it encounters the statement. The function main() will be called recursively unless I becomes equal to 0, and since main() is recursively called, so the value of static I ie., 0 will be printed every time the control is returned.

-

```
int main()
{
  void *v;
  int integer=2;
  int *i=&integer;
  v=i;
  printf("%d",(int*)*v);
  return(0);
}
```

Answer: "Compiler Error. We cannot apply indirection on type void*.". Void pointer is a generic pointer type. No pointer arithmetic can be done on it. Void pointers are normally used for, 1. Passing generic pointers to functions and returning such pointers. 2. As a intermediate pointer type. 3. Used when the exact pointer type will be known at a later point of time.

-

```
int main()
{
  float f=5,g=10;
  enum{i=10,j=20,k=50};
  printf("%d\n",++k);
  printf("%f\n",f<<2);
  printf("%lf\n",f%g);
  printf("%lf\n",fmod(f,g));
  return(0);
}
```

Answer: "Line no 5: Error: Lvalue required, Line no 6: Cannot apply leftshift to float, Line no 7: Cannot apply mod to float". Enumeration constants cannot be modified, so you cannot apply ++.Bit-wise operators and % operators cannot be applied on float values.fmod() is to find the modulus values for floats as % operator is for ints.

-

```
int main()
{
   while (strcmp(?some?,?some\0?))
   {
     printf(?Strings are not equal\n?);
   }
}
```

Answer: "No output". Ending the string constant with \0 explicitly makes no difference. So ?some? and ?some\0? are equivalent. So, strcmp returns 0 (false) hence breaking out of the while loop.

-

```
int main()
{
  char str1[] = {?s?,?o?,?m?,?e?};
  char str2[] = {?s?,?o?,?m?,?e?,?\0?};
  while (strcmp(str1,str2))
  {
    printf(?Strings are not equal\n?);
  }
  return(0);
}
```

Answer: ?Strings are not equal?, ?Strings are not equal?, ?. If a string constant is initialized explicitly with characters, ?\0? is not appended automatically to the string. Since str1 doesn?t have null termination, it treats whatever the values that are in the following positions as part of the string

until it randomly reaches a ?\0?. So str1 and str2 are not the same, hence the result.

-

```
int main()
{
  float i=1.5;
  switch(i)
  {
     case 1: printf("1");
     case 2: printf("2");
     default : printf("0");
  }
  return(0);
}
```

Answer: "Compiler Error: switch expression not integral". Switch statements can be applied only to integral types.