C LANGUAGE INTERVIEW QUESTIONS

What is C language?
The C programming language is a standardized programming language developed in the early 1970s by Ken Thompson and Dennis Ritchie for use on the UNIX operating system. It has since spread to many other operating systems, and is one of the most widely used programming languages. C is prized for its efficiency, and is the most popular programming language for writing system software, though it is also used for writing applications.

What does static variable mean?

there are 3 main uses for the static.
1. If you declare within a function:
It retains the value between function calls

2.If it is declared for a function name:
By default function is extern..so it will be visible from other files if the function declaration is as static..it is invisible for the outer files

3. Static for global variables:
By default we can use the global variables from outside files If it is static global..that variable is limited to with in the file

What are the different storage classes in C ?

C has three types of storage: automatic, static and allocated.

Variable having block scope and without static specifier have automatic storage duration.

Variables with block scope, and with static specifier have static scope. Global variables (i.e, file scope) with or without the the static specifier also have static scope.

Memory obtained from calls to malloc(), alloc() or realloc() belongs to allocated storage class.

What is hashing?
To hash means to grind up, and that's essentially what hashing is all about. The heart of a hashing algorithm is a hash function that takes your nice, neat data and grinds it into some random-looking integer.

The idea behind hashing is that some data either has no inherent ordering (such as images) or is expensive to compare (such as images). If the data has no inherent ordering, you can't perform comparison searches.

If the data is expensive to compare, the number of comparisons used even by a binary search might be too many. So instead of looking at the data themselves, you'll condense (hash) the data to an integer (its hash value) and keep all the data with the same hash value in the same place. This task is carried out by using the hash value as an index into an array.

To search for an item, you simply hash it and look at all the data whose hash values match that of the data you're looking for. This technique greatly lessens the number of items you have to look at. If the parameters are set up with care and enough storage is available for the hash table, the number of comparisons needed to find an item can be made arbitrarily close to one.

One aspect that affects the efficiency of a hashing implementation is the hash function itself. It should ideally distribute data randomly throughout the entire hash table, to reduce the likelihood of collisions. Collisions occur when two different keys have the same hash value.

There are two ways to resolve this problem. In open addressing, the collision is resolved by the choosing of another position in the hash table for the element inserted later. When the hash table is searched, if the entry is not found at its hashed position in the table, the search continues checking until either the element is found or an empty position in the table is found.

The second method of resolving a hash collision is called chaining. In this method, a bucket or linked list holds all the elements whose keys hash to the same value. When the hash table is searched, the list must be searched linearly.

Can static variables be declared in a header file?

You can't declare a static variable without defining it as well (this is because the storage class modifiers static and extern are mutually exclusive). A static variable can be defined in a header file, but this would cause each source file that included the header file to have its own private copy of the variable, which is probably not what was intended.

Can a variable be both const and volatile?

Yes. The const modifier means that this code cannot change the value of the variable, but that does not mean that the value cannot be changed by means outside this code. For instance, in the example in FAQ 8, the timer structure was accessed through a volatile const pointer. The function itself did not change the value of the timer, so it was declared const. However, the value was changed by hardware on the computer, so it was declared volatile. If a variable is both const and volatile, the two modifiers can appear in either order.

Can include files be nested?

Answer Yes. Include files can be nested any number of times. As long as you use precautionary measures , you can avoid including the same file twice. In the past, nesting header files was seen as bad programming practice, because it complicates the dependency tracking function of the MAKE program and thus slows down compilation. Many of today's popular compilers make up for this difficulty by implementing a concept called precompiled headers, in which all headers and associated dependencies are stored in a precompiled state.
Many programmers like to create a custom header file that has #include statements for every header needed for each module. This is perfectly acceptable and can help avoid

potential problems relating to #include files, such as accidentally omitting an #include file in a module.

What is a null pointer?

There are times when it's necessary to have a pointer that doesn't point to anything. The macro NULL, defined in , has a value that's guaranteed to be different from any valid pointer. NULL is a literal zero, possibly cast to void* or char*. Some people, notably C++ programmers, prefer to use 0 rather than NULL.
The null pointer is used in three ways:
1) To stop indirection in a recursive data structure
2) As an error value
3) As a sentinel value

printf() Function
What is the output of printf("%d")?

1. When we write printf("%d",x); this means compiler will print the value of x. But as here, there is nothing after %d so compiler will show in output window garbage value.

2. When we use %d the compiler internally uses it to access the argument in the stack (argument stack). Ideally compiler determines the offset of the data variable depending on the format specification string. Now when we write printf("%d",a) then compiler first accesses the top most element in the argument stack of the printf which is %d and depending on the format string it calculated to offset to the actual data variable in the memory which is to be printed. Now when only %d will be present in the printf then compiler will calculate the correct offset (which will be the offset to access the integer variable) but as the actual data object is to be printed is not present at that memory location so it will print what ever will be the contents of that memory location.

3. Some compilers check the format string and will generate an error without the proper number and type of arguments for things like printf(...) and scanf(...).
malloc()

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

1. calloc(...) allocates a block of memory for an array of elements of a certain size. By default the block is initialized to 0. The total number of memory allocated will be (number_of_elements * size).

malloc(...) takes in only a single argument which is the memory required in bytes. malloc(...) allocated bytes of memory and not blocks of memory like calloc(...).

2. malloc(...) allocates memory blocks and returns a void pointer to the allocated space, or NULL if there is insufficient memory available.

calloc(...) allocates an array in memory with elements initialized to 0 and returns a pointer to the allocated space. calloc(...) calls malloc(...) in order to use the C++ _set_new_mode function to set the new handler mode.

printf() Function- What is the difference between "printf(...)" and "sprintf(...)"?

sprintf(...) writes data to the character array whereas printf(...) writes data to the standard output device.

Compilation How to reduce a final size of executable?

Size of the final executable can be reduced using dynamic linking for libraries.

Linked Lists -- Can you tell me how to check whether a linked list is circular?

Create two pointers, and set both to the start of the list. Update each as follows:

```
while (pointer1) {
pointer1 = pointer1->next;
pointer2 = pointer2->next;
if (pointer2) pointer2=pointer2->next;
if (pointer1 == pointer2) {
print ("circular");
}
}
```

If a list is circular, at some point pointer2 will wrap around and be either at the item just before pointer1, or the item before that. Either way, its either 1 or 2 jumps until they meet.

"union" Data Type What is the output of the following program? Why?

```
#include
main() {
typedef union {
int a;
char b[10];
float c;
}
Union;

Union x,y = {100};
x.a = 50;
strcpy(x.b,"hello");
x.c = 21.50;
printf("Union x : %d %s %f n",x.a,x.b,x.c);
printf("Union y : %d %s %f n",y.a,y.b,y.c);
}
```

String Processing --- Write out a function that prints out all the permutations of a string. For example, abc would give you abc, acb, bac, bca, cab, cba.

```
void PrintPermu (char *sBegin, char* sRest) {
int iLoop;
char cTmp;
char cFLetter[1];
```

```
char *sNewBegin;
char *sCur;
int iLen;
static int iCount;

iLen = strlen(sRest);
if (iLen == 2) {
iCount++;
printf("%d: %s%s\n",iCount,sBegin,sRest);
iCount++;
printf("%d: %s%c%c\n",iCount,sBegin,sRest[1],sRest[0]);
return;
} else if (iLen == 1) {
iCount++;
printf("%d: %s%s\n", iCount, sBegin, sRest);
return;
} else {
// swap the first character of sRest with each of
// the remaining chars recursively call debug print
sCur = (char*)malloc(iLen);
sNewBegin = (char*)malloc(iLen);
for (iLoop = 0; iLoop < iLen; iLoop ++) {
strcpy(sCur, sRest);
strcpy(sNewBegin, sBegin);
cTmp = sCur[iLoop];
sCur[iLoop] = sCur[0];
sCur[0] = cTmp;
sprintf(cFLetter, "%c", sCur[0]);
strcat(sNewBegin, cFLetter);
debugprint(sNewBegin, sCur+1);
}
}
}

void main() {
char s[255];
char sIn[255];
printf("\nEnter a string:");
scanf("%s%*c",sIn);
memset(s,0,255);
PrintPermu(s, sIn);
}
```

What will be the result of the following code?

#define TRUE 0 // some code while(TRUE) { // some code }

This will not go into the loop as TRUE is defined as 0.

What will be printed as the result of the operation below:

int x;

int modifyvalue()

```
{
return(x+=10);
}

int changevalue(int x)
{
return(x+=1);
}

void main()
{
int x=10;
x++;
changevalue(x);
x++;
modifyvalue();
printf("First output:%d\n",x);

x++;
changevalue(x);
printf("Second output:%d\n",x);
modifyvalue();
printf("Third output:%d\n",x);

}
```

Answer: 12 , 13 , 13

What will be printed as the result of the operation below:

```
main()
{
int x=10, y=15;
x = x++;
y = ++y;
printf("%d %d\n",x,y);

}
```

Answer: 11, 16

What will be printed as the result of the operation below:

```
main()
{
int a=0;
if(a==0)
```

printf("Tech Preparation\n");
printf("Tech Preparation\n");

}

Answer: Two lines with "Tech Preparation" will be printed.

What will the following piece of code do

```
int f(unsigned int x)
{
int i;
for (i=0; x!0; x>>=1){
if (x & 0X1)
i++;
}
return i;
}
```

Answer: returns the number of ones in the input parameter X

What will happen in these three cases?

```
if(a=0){
//somecode
}
if (a==0){
//do something
}
if (a===0){
//do something
}
```

What are x, y, y, u

```
#define Atype int*
typedef int *p;
p x, z;
Atype y, u;
```

Answer: x and z are pointers to int. y is a pointer to int but u is just an integer variable

Advantages of a macro over a function?
Macro gets to see the Compilation environment, so it can expand __ __TIME__ __FILE__ #defines. It is expanded by the preprocessor.

For example, you can't do this without macros
#define PRINT(EXPR) printf(#EXPR "=%d\n", EXPR)

PRINT(5+6*7) // expands into printf("5+6*7=%d", 5+6*7);

You can define your mini language with macros:
#define strequal(A,B) (!strcmp(A,B))

Macros are a necessary evils of life. The purists don't like them, but without it no real work gets done.

What is the difference between strings and character arrays?
A major difference is: string will have static storage duration, whereas as a character array will not, unless it is explicity specified by using the static keyword.

Actually, a string is a character array with following properties:

* the multibyte character sequence, to which we generally call string, is used to initialize an array of static storage duration. The size of this array is just sufficient to contain these characters plus the terminating NUL character.

* it not specified what happens if this array, i.e., string, is modified.

* Two strings of same value[1] may share same memory area. For example, in the following declarations:

char *s1 = "Calvin and Hobbes";
char *s2 = "Calvin and Hobbes";

the strings pointed by s1 and s2 may reside in the same memory location. But, it is not true for the following:

char ca1[] = "Calvin and Hobbes";
char ca2[] = "Calvin and Hobbes";

[1] The value of a string is the sequence of the values of the contained characters, in order.

Write down the equivalent pointer expression for referring the same element a[i][j][k][l]?
a[i] == *(a+i)
a[i][j] == *(*(a+i)+j)
a[i][j][k] == *(*(*(a+i)+j)+k)
a[i][j][k][l] == *(*(*(*(a+i)+j)+k)+l)

Which bit wise operator is suitable for checking whether a particular bit is on or off?
The bitwise AND operator. Here is an example:enum {
KBit0 = 1,
KBit1,
...
KBit31,
};

if (some_int & KBit24)
printf ("Bit number 24 is ON\n");

else
printf (“Bit number 24 is OFF\n”);

Which bit wise operator is suitable for turning off a particular bit in a number?
The bitwise AND operator, again. In the following code snippet, the bit number 24 is reset to zero.
some_int = some_int & ~KBit24;

Which bit wise operator is suitable for putting on a particular bit in a number?
The bitwise OR operator. In the following code snippet, the bit number 24 is turned ON:
some_int = some_int | KBit24;

Does there exist any other function which can be used to convert an integer or a float to a string?
Some implementations provide a nonstandard function called itoa(), which converts an integer to string.

#include

char *itoa(int value, char *string, int radix);

DESCRIPTION
The itoa() function constructs a string representation of an integer.

PARAMETERS
value:
Is the integer to be converted to string representation.

string:
Points to the buffer that is to hold resulting string.
The resulting string may be as long as seventeen bytes.

radix:
Is the base of the number; must be in the range 2 - 36.

A portable solution exists. One can use sprintf():

char s[SOME_CONST];
int i = 10;
float f = 10.20;

sprintf (s, “%d %f\n”, i, f);

Why does malloc(0) return valid memory address ? What's the use ?
malloc(0) does not return a non-NULL under every implementation.
An implementation is free to behave in a manner it finds
suitable, if the allocation size requested is zero. The
implmentation may choose any of the following actions:

* A null pointer is returned.

* The behavior is same as if a space of non-zero size
was requested. In this case, the usage of return
value yields to undefined-behavior.

Notice, however, that if the implementation returns a non-NULL
value for a request of a zero-length space, a pointer to object
of ZERO length is returned! Think, how an object of zero size
should be represented?

For implementations that return non-NULL values, a typical usage
is as follows:

```
void
func ( void )
{
int *p; /* p is a one-dimensional array,
whose size will vary during the
the lifetime of the program */
size_t c;

p = malloc(0); /* initial allocation */
if (!p)
{
perror (”FAILURE” );
return;
}

/* ... */

while (1)
{
c = (size_t) ... ; /* Calculate allocation size */
p = realloc ( p, c * sizeof *p );

/* use p, or break from the loop */
/* ... */
}
return;
}
```

Notice that this program is not portable, since an implementation
is free to return NULL for a malloc(0) request, as the C Standard
does not support zero-sized objects.

Difference between const char* p and char const* p

in const char* p, the character pointed by ‘p’ is constant, so u cant change the value of character pointed by p but u can make ‘p’ refer to some other location.

in char const* p, the ptr ‘p’ is constant not the character referenced by it, so u cant make ‘p’ to reference to any other location but u can change the value of the char pointed by ‘p’.

How can method defined in multiple base classes with same name can be invoked from derived class simultaneously
ex:

```
class x
{
public:
m1();

};

class y
{
public:
m1();

};

class z :public x, public y
{
public:
m1()
{
x::m1();
y::m1();
}

};
```

Write a program to interchange 2 variables without using the third one.

```
a=7;
b=2;
a = a + b;
b = a - b;
a = a - b;
```

What is the result of using Option Explicit?
When writing your C program, you can include files in two ways.
The first way is to surround the file you want to include with the angled brackets < and >.
This method of inclusion tells the preprocessor to look for the file in the predefined default location.
This predefined default location is often an INCLUDE environment variable that denotes the path to your include files.
For instance, given the INCLUDE variable
INCLUDE=C:\COMPILER\INCLUDE;S:\SOURCE\HEADERS;
using the #include version of file inclusion, the compiler first checks the
C:\COMPILER\INCLUDE

directory for the specified file. If the file is not found there, the compiler then checks the S:\SOURCE\HEADERS directory. If the file is still not found, the preprocessor checks the current directory.
The second way to include files is to surround the file you want to include with double quotation marks. This method of inclusion tells the preprocessor to look for the file in the current directory first, then look for it in the predefined locations you have set up. Using the #include file version of file inclusion and applying it to the preceding example, the preprocessor first checks the current directory for the specified file. If the file is not found in the current directory, the C:COMPILERINCLUDE directory is searched. If the file is still not found, the preprocessor checks the S:SOURCEHEADERS directory.
The #include method of file inclusion is often used to include standard headers such as stdio.h or
stdlib.h.
This is because these headers are rarely (if ever) modified, and they should always be read from your compiler's standard include file directory.
The #include file method of file inclusion is often used to include nonstandard header files that you have created for use in your program. This is because these headers are often modified in the current directory, and you will want the preprocessor to use your newly modified version of the header rather than the older, unmodified version.

What is the benefit of using an enum rather than a #define constant?
The use of an enumeration constant (enum) has many advantages over using the traditional symbolic constant style of #define. These advantages include a lower maintenance requirement, improved program readability, and better debugging capability.
1) The first advantage is that enumerated constants are generated automatically by the compiler. Conversely, symbolic constants must be manually assigned values by the programmer.
For instance, if you had an enumerated constant type for error codes that could occur in your program, your enum definition could look something like this:

```
enum Error_Code
{
OUT_OF_MEMORY,
INSUFFICIENT_DISK_SPACE,
LOGIC_ERROR,
FILE_NOT_FOUND
};
```

In the preceding example, OUT_OF_MEMORY is automatically assigned the value of 0 (zero) by the compiler because it appears first in the definition. The compiler then continues to automatically assign numbers to the enumerated constants, making INSUFFICIENT_DISK_SPACE equal to 1, LOGIC_ERROR equal to 2, and FILE_NOT_FOUND equal to 3, so on.
If you were to approach the same example by using symbolic constants, your code would look something like this:

```
#define OUT_OF_MEMORY 0
#define INSUFFICIENT_DISK_SPACE 1
#define LOGIC_ERROR 2
#define FILE_NOT_FOUND 3
```

values by the programmer. Each of the two methods arrives at the same result: four constants assigned numeric values to represent error codes. Consider the maintenance required, however, if you were to add two constants to represent the error codes

DRIVE_NOT_READY and CORRUPT_FILE. Using the enumeration constant method, you simply would put these two constants anywhere in the enum definition. The compiler would generate two unique values for these constants. Using the symbolic constant method, you would have to manually assign two new numbers to these constants. Additionally, you would want to ensure that the numbers you assign to these constants are unique.
2) Another advantage of using the enumeration constant method is that your programs are more readable and thus can be understood better by others who might have to update your program later.

3) A third advantage to using enumeration constants is that some symbolic debuggers can print the value of an enumeration constant. Conversely, most symbolic debuggers cannot print the value of a symbolic constant. This can be an enormous help in debugging your program, because if your program is stopped at a line that uses an enum, you can simply inspect that constant and instantly know its value. On the other hand, because most debuggers cannot print #define values, you would most likely have to search for that value by manually looking it up in a header file.

How can I open a file so that other programs can update it at the same time?
Your C compiler library contains a low-level file function called sopen() that can be used to open a file in shared mode. Beginning with DOS 3.0, files could be opened in shared mode by loading a special program named SHARE.EXE. Shared mode, as the name implies, allows a file to be shared with other programs as well as your own.
Using this function, you can allow other programs that are running to update the same file you are updating.
The sopen() function takes four parameters: a pointer to the filename you want to open, the operational mode you want to open the file in, the file sharing mode to use, and, if you are creating a file, the mode to create the file in. The second parameter of the sopen() function, usually referred to as the operation flag parameter, can have the following values assigned to it:
Constant Description O_APPEND Appends all writes to the end of the file
O_BINARY Opens the file in binary (untranslated) mode
O_CREAT If the file does not exist, it is created
O_EXCL If the O_CREAT flag is used and the file exists, returns an error
O_RDONLY Opens the file in read-only mode
O_RDWR Opens the file for reading and writing
O_TEXT Opens the file in text (translated) mode
O_TRUNC Opens an existing file and writes over its contents
O_WRONLY Opens the file in write-only mode
The third parameter of the sopen() function, usually referred to as the sharing flag, can have the following values assigned to it:
Constant Description
SH_COMPAT No other program can access the file
SH_DENYRW No other program can read from or write to the file
SH_DENYWR No other program can write to the file
SH_DENYRD No other program can read from the file
SH_DENYNO Any program can read from or write to the file

If the sopen() function is successful, it returns a non-negative number that is the file's handle. If an error occurs, 1 is returned, and the global variable errno is set to one of the following values:

Constant Description
ENOENT File or path not found
EMFILE No more file handles are available
EACCES Permission denied to access file
EINVACC Invalid access code
Constant Description

What is the quickest sorting method to use?

The answer depends on what you mean by quickest. For most sorting problems, it just doesn't matter how quick the sort is because it is done infrequently or other operations take significantly more time anyway. Even in cases in which sorting speed is of the essence, there is no one answer. It depends on not only the size and nature of the data, but also the likely order. No algorithm is best in all cases.

There are three sorting methods in this author's toolbox that are all very fast and that are useful in different situations. Those methods are quick sort, merge sort, and radix sort.

The Quick Sort

The quick sort algorithm is of the divide and conquer type. That means it works by reducing a sorting problem into several easier sorting problems and solving each of them. A dividing value is chosen from the input data, and the data is partitioned into three sets: elements that belong before the dividing value, the value itself, and elements that come after the dividing value. The partitioning is performed by exchanging elements that are in the first set but belong in the third with elements that are in the third set but belong in the first Elements that are equal to the dividing element can be put in any of the three setsthe algorithm will still work properly.

The Merge Sort

The merge sort is a divide and conquer sort as well. It works by considering the data to be sorted as a sequence of already-sorted lists (in the worst case, each list is one element long). Adjacent sorted lists are merged into larger sorted lists until there is a single sorted list containing all the elements. The merge sort is good at sorting lists and other data structures that are not in arrays, and it can be used to sort things that don't fit into memory. It also can be implemented as a stable sort.

The Radix Sort

The radix sort takes a list of integers and puts each element on a smaller list, depending on the value of its least significant byte. Then the small lists are concatenated, and the process is repeated for each more significant byte until the list is sorted. The radix sort is simpler to implement on fixed-length data such as ints.

when should the volatile modifier be used?

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 computer's 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.

When should the register modifier be used? Does it really help?

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.

How can you determine the size of an allocated portion of memory?
You can't, really. free() can , but there's no way for your program to know the trick free() uses. Even if you disassemble the library and discover the trick, there's no guarantee the trick won't change with the next release of the compiler.

What is page thrashing?
Some operating systems (such as UNIX or Windows in enhanced mode) use virtual memory. Virtual memory is a technique for making a machine behave as if it had more memory than it really has, by using disk space to simulate RAM (random-access memory). In the 80386 and higher Intel CPU chips, and in most other modern microprocessors (such as the Motorola 68030, Sparc, and Power PC), exists a piece of hardware called the Memory Management Unit, or MMU.
The MMU treats memory as if it were composed of a series of pages. A page of memory is a block of contiguous bytes of a certain size, usually 4096 or 8192 bytes. The operating system sets up and maintains a table for each running program called the Process Memory Map, or PMM. This is a table of all the pages of memory that program can access and where each is really located.
Every time your program accesses any portion of memory, the address (called a virtual address) is processed by the MMU. The MMU looks in the PMM to find out where the memory is really located (called the physical address). The physical address can be any location in memory or on disk that the operating system has assigned for it. If the location the program wants to access is on disk, the page containing it must be read from disk into memory, and the PMM must be updated to reflect this action (this is called a page fault). Because accessing the disk is so much slower than accessing RAM, the operating system tries to keep as much of the virtual memory as possible in RAM. If you're running a large enough program (or several small programs at once), there might not be enough RAM to hold all the memory used by the programs, so some of it must be moved out of RAM and onto disk (this action is called paging out). The operating system tries to guess which areas of memory aren't likely to be used for a while (usually based on how the memory has been used in the past). If it guesses wrong, or if your programs are accessing lots of memory in lots of places, many page faults will occur in order to read in the pages that were paged out. Because all of RAM is being used, for each page read in to be accessed, another page must be paged out. This can lead to more page faults, because now a different page of memory has been moved to disk.
The problem of many page faults occurring in a short time, called page thrashing, can drastically cut the performance of a system. Programs that frequently access many widely separated locations in memory are more likely to cause page thrashing on a system. So is running many small programs that all continue to run even when you are not actively using them. To reduce page thrashing, you can run fewer programs simultaneously. Or you can try changing the way a large program works to maximize the capability of the operating system to guess which pages won't be needed. You can achieve this effect by caching values or changing lookup algorithms in large data structures, or sometimes by

changing to a memory allocation library which provides an implementation of malloc() that allocates memory more efficiently. Finally, you might consider adding more RAM to the system to reduce the need to page out.

How do you override a defined macro?
You can use the #undef preprocessor directive to undefine (override) a previously defined macro.

How can you check to see whether a symbol is defined?
You can use the #ifdef and #ifndef preprocessor directives to check whether a symbol has been defined (#ifdef) or whether it has not been defined (#ifndef).
Can you define which header file to include at compile time? Yes. This can be done by using the #if, #else, and #endif preprocessor directives. For example, certain compilers use different names for header files. One such case is between Borland C++, which uses the header file alloc.h, and Microsoft C++, which uses the header file malloc.h. Both of these headers serve the same purpose, and each contains roughly the same definitions. If, however, you are writing a program that is to support Borland C++ and Microsoft C++, you must define which header to include at compile time. The following example shows how this can be done:

```
#ifdef _ _BORLANDC_ _
#include
#else
#include
#endif
```

Write the equivalent expression for x % 8?
x&7

When does the compiler not implicitly generate the address of the first element of an array?
Whenever an array name appears in an expression such as
- array as an operand of the sizeof operator
- array as an operand of & operator
- array as a string literal initializer for a character array

Then the compiler does not implicitly generate the address of the address of the first element of an array.

What is the benefit of using #define to declare a constant?
Using the #define method of declaring a constant enables you to declare a constant in one place and use it throughout your program. This helps make your programs more maintainable, because you need to maintain only the #define statement and not several instances of individual constants throughout your program.
For instance, if your program used the value of pi (approximately 3.14159) several times, you might want to declare a constant for pi as follows:

```
#define PI 3.14159
```

Using the #define method of declaring a constant is probably the most familiar way of declaring constants to traditional C programmers. Besides being the most common method of declaring constants, it also takes up the least memory. Constants defined in this manner are simply placed directly into your source code, with no variable space

allocated in memory. Unfortunately, this is one reason why most debuggers cannot inspect constants created using the #define method.

How can I search for data in a linked list?
Unfortunately, the only way to search a linked list is with a linear search, because the only way a linked list's members can be accessed is sequentially. Sometimes it is quicker to take the data from a linked list and store it in a different data structure so that searches can be more efficient.

Why should we assign NULL to the elements (pointer) after freeing them?
This is paranoia based on long experience. After a pointer has been freed, you can no longer use the pointed-to data. The pointer is said to dangle; it doesn't point at anything useful. If you NULL out or zero out a pointer immediately after freeing it, your program can no longer get in trouble by using that pointer. True, you might go indirect on the null pointer instead, but that's something your debugger might be able to help you with immediately. Also, there still might be copies of the pointer that refer to the memory that has been deallocated; that's the nature of C. Zeroing out pointers after freeing them won't solve all problems;

What is a null pointer assignment error? What are bus errors, memory faults, and core dumps?
These are all serious errors, symptoms of a wild pointer or subscript.
Null pointer assignment is a message you might get when an MS-DOS program finishes executing. Some such programs can arrange for a small amount of memory to be available "where the NULL pointer points to (so to speak). If the program tries to write to that area, it will overwrite the data put there by the compiler.
When the program is done, code generated by the compiler examines that area. If that data has been changed, the compiler-generated code complains with null pointer assignment. This message carries only enough information to get you worried. There's no way to tell, just from a null pointer assignment message, what part of your program is responsible for the error. Some debuggers, and some compilers, can give you more help in finding the problem.

Bus error: core dumped and Memory fault: core dumped are messages you might see from a program running under UNIX. They're more programmer friendly. Both mean that a pointer or an array subscript was wildly out of bounds. You can get these messages on a read or on a write. They aren't restricted to null pointer problems.
The core dumped part of the message is telling you about a file, called core, that has just been written in your current directory. This is a dump of everything on the stack and in the heap at the time the program was running. With the help of a debugger, you can use the core dump to find where the bad pointer was used.
That might not tell you why the pointer was bad, but it's a step in the right direction. If you don't have write permission in the current directory, you won't get a core file, or the core dumped message

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));

What is the difference between a string copy (strcpy) and a memory copy (memcpy)? When should each be used?

The strcpy() function is designed to work exclusively with strings. It copies each byte of the source string to the destination string and stops when the terminating null character () has been moved. On the other hand, the memcpy() function is designed to work with any type of data. Because not all data ends with a null character, you must provide the memcpy() function with the number of bytes you want to copy from the source to the destination.

How can I convert a string to a number?

The standard C library provides several functions for converting strings to numbers of all formats (integers, longs, floats, and so on) and vice versa.
The following functions can be used to convert strings to numbers:
Function Name Purpose
atof() Converts a string to a double-precision floating-point value.
atoi() Converts a string to an integer.
atol() Converts a string to a long integer.
strtod() Converts a string to a double-precision floating-point value and reports any leftover numbers that could not be converted.
strtol() Converts a string to a long integer and reports any leftover numbers that could not be converted.
strtoul() Converts a string to an unsigned long integer and reports any leftover numbers that could not be converted.

How can I convert a number to a string?

The standard C library provides several functions for converting numbers of all formats (integers, longs, floats, and so on) to strings and vice versa The following functions can be used to convert integers to strings:
Function Name Purpose
itoa() Converts an integer value to a string.
ltoa() Converts a long integer value to a string.
ultoa() Converts an unsigned long integer value to a string.
The following functions can be used to convert floating-point values to strings:
Function Name Purpose
ecvt() Converts a double-precision floating-point value to a string without an embedded decimal point.
fcvt() Same as ecvt(), but forces the precision to a specified number of digits.
gcvt() Converts a double-precision floating-point value to a string with an embedded decimal point.

Is it possible to execute code even after the program exits the main() function?

The standard C library provides a function named atexit() that can be used to perform cleanup operations when your program terminates. You can set up a set of functions you want to perform automatically when your program exits by passing function pointers to the at exit() function.

What is the stack?
The stack is where all the functions' local (auto) variables are created. The stack also contains some information used to call and return from functions.
A stack trace is a list of which functions have been called, based on this information. When you start using a debugger, one of the first things you should learn is how to get a stack trace.
The stack is very inflexible about allocating memory; everything must be deallocated in exactly the reverse order it was allocated in. For implementing function calls, that is all that's needed. Allocating memory off the stack is extremely efficient. One of the reasons C compilers generate such good code is their heavy use of a simple stack.
There used to be a C function that any programmer could use for allocating memory off the stack. The memory was automatically deallocated when the calling function returned. This was a dangerous function to call; it's not available anymore.

How do you print an address?
The safest way is to use printf() (or fprintf() or sprintf()) with the %P specification. That prints a void pointer (void*). Different compilers might print a pointer with different formats. Your compiler will pick a format that's right for your environment.
If you have some other kind of pointer (not a void*) and you want to be very safe, cast the pointer to a void*:
printf(%Pn, (void*) buffer);

Can a file other than a .h file be included with #include?
The preprocessor will include whatever file you specify in your #include statement. Therefore, if you have the line
#include
in your program, the file macros.inc will be included in your precompiled program. It is, however, unusual programming practice to put any file that does not have a .h or .hpp extension in an #include statement.
You should always put a .h extension on any of your C files you are going to include. This method makes it easier for you and others to identify which files are being used for preprocessing purposes. For instance, someone modifying or debugging your program might not know to look at the macros.inc file for macro definitions. That person might try in vain by searching all files with .h extensions and come up empty. If your file had been named macros.h, the search would have included the macros.h file, and the searcher would have been able to see what macros you defined in it.

What is Preprocessor?
The preprocessor is used to modify your program according to the preprocessor directives in your source code. Preprocessor directives (such as #define) give the preprocessor specific instructions on how to modify your source code. The preprocessor reads in all of your include files and the source code you are compiling and creates a preprocessed version of your source code. This preprocessed version has all of its macros and constant symbols replaced by their corresponding code and value assignments. If your source code contains any conditional preprocessor directives (such as #if), the preprocessor evaluates the condition and modifies your source code accordingly.
The preprocessor contains many features that are powerful to use, such as creating macros, performing conditional compilation, inserting predefined environment variables into your code, and turning compiler features on and off. For the professional programmer,

in-depth knowledge of the features of the preprocessor can be one of the keys to creating fast, efficient programs.

How can you restore a redirected standard stream?

The preceding example showed how you can redirect a standard stream from within your program. But what if later in your program you wanted to restore the standard stream to its original state? By using the standard C library functions named dup() and fdopen(), you can restore a standard stream such as stdout to its original state.
The dup() function duplicates a file handle. You can use the dup() function to save the file handle corresponding to the stdout standard stream. The fdopen() function opens a stream that has been duplicated with the dup() function.

What is the purpose of realloc( )?

The function realloc(ptr,n) uses two arguments.the first argument ptr is a pointer to a block of memory for which the size is to be altered. The second argument n specifies the new size. The size may be increased or decreased. If n is greater than the old size and if sufficient space is not available subsequent to the old region, the function realloc( ) may create a new region and all the old data are moved to the new region.

What is the heap?

The heap is where malloc(), calloc(), and realloc() get memory.
Getting memory from the heap is much slower than getting it from the stack. On the other hand, the heap is much more flexible than the stack. Memory can be allocated at any time and deallocated in any order. Such memory isn't deallocated automatically; you have to call free().
Recursive data structures are almost always implemented with memory from the heap. Strings often come from there too, especially strings that could be very long at runtime. If you can keep data in a local variable (and allocate it from the stack), your code will run faster than if you put the data on the heap. Sometimes you can use a better algorithm if you use the heap faster, or more robust, or more flexible. It's a tradeoff.
If memory is allocated from the heap, it's available until the program ends. That's great if you remember to deallocate it when you're done. If you forget, it's a problem. A memory leak is some allocated memory that's no longer needed but isn't deallocated. If you have a memory leak inside a loop, you can use up all the memory on the heap and not be able to get any more. (When that happens, the allocation functions return a null pointer.) In some environments, if a program doesn't deallocate everything it allocated, memory stays unavailable even after the program ends.

How do you use a pointer to a function?

The hardest part about using a pointer-to-function is declaring it.
Consider an example. You want to create a pointer, pf, that points to the strcmp() function. The strcmp() function is declared in this way:
int strcmp(const char *, const char *)
To set up pf to point to the strcmp() function, you want a declaration that looks just like the strcmp() function's declaration, but that has *pf rather than strcmp:
int (*pf)(const char *, const char *);
After you've gotten the declaration of pf, you can #include and assign the address of strcmp() to pf: pf = strcmp;

What is the purpose of main( ) function?
The function main( ) invokes other functions within it.It is the first function to be called when the program starts execution.
- It is the starting function
- It returns an int value to the environment that called the program
- Recursive call is allowed for main( ) also.
- It is a user-defined function
- Program execution ends when the closing brace of the function main( ) is reached.
- It has two arguments 1)argument count and 2) argument vector (represents strings passed).
- Any user-defined name can also be used as parameters for main( ) instead of argc and argv

Why n++ executes faster than n+1?
The expression n++ requires a single machine instruction such as INR to carry out the increment operation whereas, n+1 requires more instructions to carry out this operation.

What will the preprocessor do for a program?
The C preprocessor is used to modify your program according to the preprocessor directives in your source code. A preprocessor directive is a statement (such as #define) that gives the preprocessor specific instructions on how to modify your source code. The preprocessor is invoked as the first part of your compiler program's compilation step. It is usually hidden from the programmer because it is run automatically by the compiler.
The preprocessor reads in all of your include files and the source code you are compiling and creates a preprocessed version of your source code. This preprocessed version has all of its macros and constant symbols replaced by their corresponding code and value assignments. If your source code contains any conditional preprocessor directives (such as #if), the preprocessor evaluates the condition and modifies your source code accordingly.

What is the benefit of using const for declaring constants?
The benefit of using the const keyword is that the compiler might be able to make optimizations based on the knowledge that the value of the variable will not change. In addition, the compiler will try to ensure that the values won't be changed inadvertently.
Of course, the same benefits apply to #defined constants. The reason to use const rather than #define to define a constant is that a const variable can be of any type (such as a struct, which can't be represented by a #defined constant). Also, because a const variable is a real variable, it has an address that can be used, if needed, and it resides in only one place in memory

What is the easiest sorting method to use?
The answer is the standard library function qsort(). It's the easiest sort by far for several reasons:
It is already written.
It is already debugged.
It has been optimized as much as possible (usually).
Void qsort(void *buf, size_t num, size_t size, int (*comp)(const void *ele1, const void *ele2));

How many levels of pointers can you have?
The answer depends on what you mean by levels of pointers. If you mean How many levels of indirection can you have in a single declaration? the answer is At least 12.

```
int i = 0;
int *ip01 = & i;
int **ip02 = & ip01;
int ***ip03 = & ip02;
int ****ip04 = & ip03;
int *****ip05 = & ip04;
int ******ip06 = & ip05;
int *******ip07 = & ip06;
int ********ip08 = & ip07;
int *********ip09 = & ip08;
int **********ip10 = & ip09;
int ***********ip11 = & ip10;
int ************ip12 = & ip11;
************ip12 = 1; /* i = 1 */
```

The ANSI C standard says all compilers must handle at least 12 levels. Your compiler might support more.

Is it better to use a macro or a function?

The answer depends on the situation you are writing code for. Macros have the distinct advantage of being more efficient (and faster) than functions, because their corresponding code is inserted directly into your source code at the point where the macro is called. There is no overhead involved in using a macro like there is in placing a call to a function. However, macros are generally small and cannot handle large, complex coding constructs. A function is more suited for this type of situation. Additionally, macros are expanded inline, which means that the code is replicated for each occurrence of a macro. Your code therefore could be somewhat larger when you use macros than if you were to use functions.

Thus, the choice between using a macro and using a function is one of deciding between the tradeoff of faster program speed versus smaller program size. Generally, you should use macros to replace small, repeatable code sections, and you should use functions for larger coding tasks that might require several lines of code.

What are the standard predefined macros?

The ANSI C standard defines six predefined macros for use in the C language:

Macro Name Purpose

_ _LINE_ _ Inserts the current source code line number in your code.

_ _FILE_ _ Inserts the current source code filename in your code.

_ _ Inserts the current date of compilation in your code.

_ _TIME_ _ Inserts the current time of compilation in your code.

_ _STDC_ _ Is set to 1 if you are enforcing strict ANSI C conformity.

_ _cplusplus Is defined if you are compiling a C++ program.

What is a const pointer?

The access modifier keyword const is a promise the programmer makes to the compiler that the value of a variable will not be changed after it is initialized. The compiler will enforce that promise as best it can by not enabling the programmer to write code which modifies a variable that has been declared const.

A const pointer, or more correctly, a pointer to const, is a pointer which points to data that is const (constant, or unchanging). A pointer to const is declared by putting the word const at the beginning of the pointer declaration. This declares a pointer which points to data

that can't be modified. The pointer itself can be modified. The following example illustrates some legal and illegal uses of a const pointer:

```
const char *str = hello;
char c = *str  /* legal */
str++;  /* legal */
*str = 'a';  /* illegal */
str[1] = 'b';  /* illegal */
```

What is a pragma?

The #pragma preprocessor directive allows each compiler to implement compiler-specific features that can be turned on and off with the #pragma statement. For instance, your compiler might support a feature called loop optimization. This feature can be invoked as a command-line option or as a #pragma directive.

To implement this option using the #pragma directive, you would put the following line into your code:

```
#pragma loop_opt(on)
```

Conversely, you can turn off loop optimization by inserting the following line into your code:

```
#pragma loop_opt(off)
```

What is #line used for?

The #line preprocessor directive is used to reset the values of the _ _LINE_ _ and _ _FILE_ _ symbols, respectively. This directive is commonly used in fourth-generation languages that generate C language source files.

What is the difference between text and binary modes?

Streams can be classified into two types: text streams and binary streams. Text streams are interpreted, with a maximum length of 255 characters. With text streams, carriage return/line feed combinations are translated to the newline n character and vice versa. Binary streams are uninterpreted and are treated one byte at a time with no translation of characters. Typically, a text stream would be used for reading and writing standard text files, printing output to the screen or printer, or receiving input from the keyboard.

A binary text stream would typically be used for reading and writing binary files such as graphics or word processing documents, reading mouse input, or reading and writing to the modem.

How do you determine whether to use a stream function or a low-level function?

Stream functions such as fread() and fwrite() are buffered and are more efficient when reading and writing text or binary data to files. You generally gain better performance by using stream functions rather than their unbuffered low-level counterparts such as read() and write().

In multi-user environments, however, when files are typically shared and portions of files are continuously being locked, read from, written to, and unlocked, the stream functions do not perform as well as the low-level functions. This is because it is hard to buffer a shared file whose contents are constantly changing. Generally, you should always use buffered stream functions when accessing nonshared files, and you should always use the low-level functions when accessing shared files

What is static memory allocation and dynamic memory allocation?

Static memory allocation: The compiler allocates the required memory space for a declared variable.By using the address of operator,the reserved address is obtained and this address may be assigned to a pointer variable.Since most of the declared variable have static memory,this way of assigning pointer value to a pointer variable is known as static memory allocation. memory is assigned during compilation time.

Dynamic memory allocation: It uses functions such as malloc( ) or calloc( ) to get memory dynamically.If these functions are used to get memory dynamically and the values returned by these functions are assingned to pointer variables, such assignments are known as dynamic memory allocation.memory is assined during run time.

When should a far pointer be used?

Sometimes you can get away with using a small memory model in most of a given program. There might be just a few things that don't fit in your small data and code segments. When that happens, you can use explicit far pointers and function declarations to get at the rest of memory. A far function can be outside the 64KB segment most functions are shoehorned into for a small-code model. (Often, libraries are declared explicitly far, so they'll work no matter what code model the program uses.) A far pointer can refer to information outside the 64KB data segment. Typically, such pointers are used with farmalloc() and such, to manage a heap separate from where all the rest of the data lives. If you use a small-data, large-code model, you should explicitly make your function pointers far.

What is the difference between far and near?

Some compilers for PC compatibles use two types of pointers. near pointers are 16 bits long and can address a 64KB range. far pointers are 32 bits long and can address a 1MB range.

Near pointers operate within a 64KB segment. There's one segment for function addresses and one segment for data. far pointers have a 16-bit base (the segment address) and a 16-bit offset. The base is multiplied by 16, so a far pointer is effectively 20 bits long. Before you compile your code, you must tell the compiler which memory model to use. If you use a smallcode memory model, near pointers are used by default for function addresses. That means that all the functions need to fit in one 64KB segment. With a large-code model, the default is to use far function addresses. You'll get near pointers with a small data model, and far pointers with a large data model. These are just the defaults; you can declare variables and functions as explicitly near or far.

far pointers are a little slower. Whenever one is used, the code or data segment register needs to be swapped out. far pointers also have odd semantics for arithmetic and comparison. For example, the two far pointers in the preceding example point to the same address, but they would compare as different! If your program fits in a small-data, small-code memory model, your life will be easier.

When would you use a pointer to a function?

Pointers to functions are interesting when you pass them to other functions. A function that takes function pointers says, in effect, Part of what I do can be customized. Give me a pointer to a function, and I'll call it when that part of the job needs to be done. That function can do its part for me. This is known as a callback. It's used a lot in graphical user interface libraries, in which the style of a display is built into the library but the contents of the display are part of the application.

As a simpler example, say you have an array of character pointers (char*s), and you want

to sort it by the value of the strings the character pointers point to. The standard qsort() function uses function pointers to perform that task. qsort() takes four arguments,
- a pointer to the beginning of the array,
- the number of elements in the array,
- the size of each array element, and
- a comparison function, and returns an int.

How are pointer variables initialized?
Pointer variable are initialized by one of the following two ways
- Static memory allocation
- Dynamic memory allocation

How can you avoid including a header more than once?
One easy technique to avoid multiple inclusions of the same header is to use the #ifndef and #define
preprocessor directives. When you create a header for your program, you can #define a symbolic name that is unique to that header. You can use the conditional preprocessor directive named #ifndef to check whether that symbolic name has already been assigned. If it is assigned, you should not include the header, because it has already been preprocessed. If it is not defined, you should define it to avoid any further inclusions of the header. The following header illustrates this technique:

```
#ifndef _FILENAME_H
#define _FILENAME_H
#define VER_NUM 1.00.00
#define REL_DATE 08/01/94
#if _ _WINDOWS_ _
#define OS_VER WINDOWS
#else
#define OS_VER DOS
#endif
#endif
```

When the preprocessor encounters this header, it first checks to see whether _FILENAME_H has been defined. If it hasn't been defined, the header has not been included yet, and the _FILENAME_H symbolic name is defined. Then, the rest of the header is parsed until the last #endif is encountered, signaling the end of the conditional #ifndef _FILENAME_H statement. Substitute the actual name of the header file for FILENAME in the preceding example to make it applicable for your programs.

Difference between arrays and pointers?
- Pointers are used to manipulate data using the address. Pointers use * operator to access the data pointed to by them
- Arrays use subscripted variables to access and manipulate data. Array variables can be equivalently written using pointer expression.

What are the advantages of the functions?
- Debugging is easier
- It is easier to understand the logic involved in the program
- Testing is easier

- Recursive call is possible
- Irrelevant details in the user point of view are hidden in functions
- Functions are helpful in generalizing the program

Is NULL always defined as 0?
NULL is defined as either 0 or (void*)0. These values are almost identical; either a literal zero or a void pointer is converted automatically to any kind of pointer, as necessary, whenever a pointer is needed (although the compiler can't always tell when a pointer is needed).

What is the difference between NULL and NUL?
NULL is a macro defined in for the null pointer.
NUL is the name of the first character in the ASCII character set. It corresponds to a zero value. There's no standard macro NUL in C, but some people like to define it.
The digit 0 corresponds to a value of 80, decimal. Don't confuse the digit 0 with the value of '' (NUL)! NULL can be defined as ((void*)0), NUL as ''.

Can the sizeof operator be used to tell the size of an array passed to a function?
No. There's no way to tell, at runtime, how many elements are in an array parameter just by looking at the array parameter itself. Remember, passing an array to a function is exactly the same as passing a pointer to the first element.

Is using exit() the same as using return?
No. The exit() function is used to exit your program and return control to the operating system. The return statement is used to return from a function and return control to the calling function. If you issue a return from the main() function, you are essentially returning control to the calling function, which is the operating system. In this case, the return statement and exit() function are similar.

Can math operations be performed on a void pointer?
No. Pointer addition and subtraction are based on advancing the pointer by a number of elements. By definition, if you have a void pointer, you don't know what it's pointing to, so you don't know the size of what it's pointing to. If you want pointer arithmetic to work on raw addresses, use character pointers.

Can the size of an array be declared at runtime?
No. In an array declaration, the size must be known at compile time. You can't specify a size that's known only at runtime. For example, if i is a variable, you can't write code like this:
char array[i]; /* not valid C */
Some languages provide this latitude. C doesn't. If it did, the stack would be more complicated, function calls would be more expensive, and programs would run a lot slower. If you know that you have an array but you won't know until runtime how big it will be, declare a pointer to it and use malloc() or calloc() to allocate the array from the heap.

Can you add pointers together? Why would you?
No, you can't add pointers together. If you live at 1332 Lakeview Drive, and your neighbor lives at 1364 Lakeview, what's 1332+1364? It's a number, but it doesn't mean anything. If you try to perform this type of calculation with pointers in a C program, your compiler will

complain.
The only time the addition of pointers might come up is if you try to add a pointer and the difference of two pointers.

Are pointers integers?
No, pointers are not integers. A pointer is an address. It is merely a positive number and not an integer.

How do you redirect a standard stream?
Most operating systems, including DOS, provide a means to redirect program input and output to and from different devices. This means that rather than your program output (stdout) going to the screen; it can be redirected to a file or printer port. Similarly, your program's input (stdin) can come from a file rather than the keyboard. In DOS, this task is accomplished using the redirection characters, < and >. For example, if you wanted a program named PRINTIT.EXE to receive its input (stdin) from a file named STRINGS.TXT, you would enter the following command at the DOS prompt:
C:> PRINTIT <STRINGS.TXT
Notice that the name of the executable file always comes first. The less-than sign (<) tells DOS to take the strings contained in STRINGS.TXT and use them as input for the PRINTIT program.
The following example would redirect the program's output to the prn device, usually the printer attached on LPT1:
C :> REDIR > PRN
Alternatively, you might want to redirect the program's output to a file, as the following example shows:
C :> REDIR > REDIR.OUT
In this example, all output that would have normally appeared on-screen will be written to the file
REDIR.OUT.
Redirection of standard streams does not always have to occur at the operating system. You can redirect a standard stream from within your program by using the standard C library function named freopen(). For example, if you wanted to redirect the stdout standard stream within your program to a file named OUTPUT.TXT, you would implement the freopen() function as shown here:
... freopen(output.txt, w, stdout);
...
Now, every output statement (printf(), puts(), putch(), and so on) in your program will appear in the file OUTPUT.TXT.

What is a method?
Method is a way of doing something, especially a systematic way; implies an orderly logical arrangement (usually in steps).

What is the easiest searching method to use?
Just as qsort() was the easiest sorting method, because it is part of the standard library, bsearch() is the easiest searching method to use. If the given array is in the sorted order bsearch() is the best method.
Following is the prototype for bsearch():
void *bsearch(const void *key, const void *buf, size_t num, size_t size, int (*comp)(const void *, const void*));

Another simple searching method is a linear search. A linear search is not as fast as bsearch() for searching among a large number of items, but it is adequate for many purposes. A linear search might be the only method available, if the data isn't sorted or can't be accessed randomly. A linear search starts at the beginning and sequentially compares the key to each element in the data set.

Is it better to use a pointer to navigate an array of values, or is it better to use a subscripted array name?
It's easier for a C compiler to generate good code for pointers than for subscripts.

What is indirection?
If you declare a variable, its name is a direct reference to its value. If you have a pointer to a variable, or any other object in memory, you have an indirect reference to its value.

How are portions of a program disabled in demo versions?
If you are distributing a demo version of your program, the preprocessor can be used to enable or disable portions of your program. The following portion of code shows how this task is accomplished, using the preprocessor directives #if and #endif:

```
int save_document(char* doc_name)
{
#if DEMO_VERSION
printf(Sorry! You can't save documents using the DEMO version of this programming);
return(0);
#endif
...
}
```

What is modular programming?
If a program is large, it is subdivided into a number of smaller programs that are called modules or subprograms. If a complex problem is solved using more modules, this approach is known as modular programming.

How can you determine the maximum value that a numeric variable can hold?
For integral types, on a machine that uses two's complement arithmetic (which is just about any machine you're likely to use), a signed type can hold numbers from 2(number of bits 1) to +2(number of bits 1) 1. An unsigned type can hold values from 0 to +2(number of bits) 1. For instance, a 16-bit signed integer can hold numbers from 2^15 (32768) to +2^15 1 (32767).

How can you determine the maximum value that a numeric variable can hold?
How reliable are floating-point comparisons? Floating-point numbers are the black art of computer programming. One reason why this is so is that there is no optimal way to represent an arbitrary number. The Institute of Electrical and Electronic Engineers (IEEE) has developed a standard for the representation of floating-point numbers, but you cannot guarantee that every machine you use will conform to the standard.
Even if your machine does conform to the standard, there are deeper issues. It can be shown mathematically that there are an infinite number of real numbers between any two numbers. For the computer to distinguish between two numbers, the bits that represent them must differ. To represent an infinite number of different bit patterns would take an

infinite number of bits. Because the computer must represent a large range of numbers in a small number of bits (usually 32 to 64 bits), it has to make approximate representations of most numbers.
Because floating-point numbers are so tricky to deal with, it's generally bad practice to compare a floating-point number for equality with anything. Inequalities are much safer.

How can you determine the maximum value that a numeric variable can hold?
Which expression always return true? Which always return false? expression if (a=0) always return false
expression if (a=1) always return true

How many levels deep can include files be nested?
Even though there is no limit to the number of levels of nested include files you can have, your compiler might run out of stack space while trying to include an inordinately high number of files. This number varies according to your hardware configuration and possibly your compiler.

What is the difference between declaring a variable and defining a variable?
Declaring a variable means describing its type to the compiler but not allocating any space for it. Defining a variable means declaring it and also allocating space to hold the variable. You can also initialize a variable at the time it is defined.

How can I make sure that my program is the only one accessing a file?
By using the sopen() function you can open a file in shared mode and explicitly deny reading and writing permissions to any other program but yours. This task is accomplished by using the SH_DENYWR shared flag to denote that your program is going to deny any writing or reading attempts by other programs.
For example, the following snippet of code shows a file being opened in shared mode, denying access to all other files:
/* Note that the sopen() function is not ANSI compliant... */ fileHandle = sopen("C:DATASETUP.DAT", O_RDWR, SH_DENYWR);
By issuing this statement, all other programs are denied access to the SETUP.DAT file. If another program were to try to open SETUP.DAT for reading or writing, it would receive an EACCES error code, denoting that access is denied to the file.

How can I sort a linked list?
Both the merge sort and the radix sort are good sorting algorithms to use for linked lists.

Is it better to use malloc() or calloc()?
Both the malloc() and the calloc() functions are used to allocate dynamic memory. Each operates slightly different from the other. malloc() takes a size and returns a pointer to a chunk of memory at least that big:
void *malloc(size_t size);
calloc() takes a number of elements, and the size of each, and returns a pointer to a chunk of memory at least big enough to hold them all:
void *calloc(size_t numElements, size_t sizeOfElement);
There's one major difference and one minor difference between the two functions. The major difference is that malloc() doesn't initialize the allocated memory. The first time malloc() gives you a particular chunk of memory, the memory might be full of zeros. If

memory has been allocated, freed, and reallocated, it probably has whatever junk was left in it. That means, unfortunately, that a program might run in simple cases (when memory is never reallocated) but break when used harder (and when memory is reused). calloc() fills the allocated memory with all zero bits. That means that anything there you're going to use as a char or an int of any length, signed or unsigned, is guaranteed to be zero. Anything you're going to use as a pointer is set to all zero bits. That's usually a null pointer, but it's not guaranteed.Anything you're going to use as a float or double is set to all zero bits; that's a floating-point zero on some types of machines, but not on all.
The minor difference between the two is that calloc() returns an array of objects; malloc() returns one object. Some people use calloc() to make clear that they want an array.

What does it mean when a pointer is used in an if statement?
Any time a pointer is used as a condition, it means "Is this a non-null pointer?" A pointer can be used in an if, while, for, or do/while statement, or in a conditional expression.

Array is an lvalue or not?
An lvalue was defined as an expression to which a value can be assigned. Is an array an expression to which we can assign a value? 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 is an lvalue?
An lvalue is an expression to which a value can be assigned. The lvalue expression is located on the left side of an assignment statement, whereas an rvalue is located on the right side of an assignment statement. Each assignment statement must have an lvalue and an rvalue. The lvalue expression must reference a storable variable in memory. It cannot be a constant.

Diffenentiate between an internal static and external static variable?
An internal static variable is declared inside a block with static storage class whereas an external static variable is declared outside all the blocks in a file.An internal static variable

has persistent storage,block scope and no linkage.An external static variable has permanent storage,file scope and internal linkage.

What is the difference between a string and an array?

An array is an array of anything. A string is a specific kind of an array with a well-known convention to determine its length.

There are two kinds of programming languages: those in which a string is just an array of characters, and those in which it's a special type. In C, a string is just an array of characters (type char), with one wrinkle: a C string always ends with a NUL character. The "value" of an array is the same as the address of (or a pointer to) the first element; so, frequently, a C string and a pointer to char are used to mean the same thing.

An array can be any length. If it's passed to a function, there's no way the function can tell how long the array is supposed to be, unless some convention is used. The convention for strings is NUL termination; the last character is an ASCII NUL ('') character.

What is an argument ? differentiate between formal arguments and actual arguments?

An argument is an entity used to pass the data from calling function to the called function. Formal arguments are the arguments available in the function definition. They are preceded by their own data types. Actual arguments are available in the function call.

What are advantages and disadvantages of external storage class?

Advantages of external storage class

1)Persistent storage of a variable retains the latest value

2)The value is globally available

Disadvantages of external storage class

1)The storage for an external variable exists even when the variable is not needed

2)The side effect may produce surprising output

3)Modification of the program is difficult

4)Generality of a program is affected

What is a void pointer?

A void pointer is a C convention for a raw address. The compiler has no idea what type of object a void Pointer really points to. If you write

int *ip;

ip points to an int. If you write

void *p;

p doesn't point to a void!

In C and C++, any time you need a void pointer, you can use another pointer type. For example, if you have a char*, you can pass it to a function that expects a void*. You don't even need to cast it. In C (but not in C++), you can use a void* any time you need any kind of pointer, without casting. (In C++, you need to cast it).

A void pointer is used for working with raw memory or for passing a pointer to an unspecified type.

Some C code operates on raw memory. When C was first invented, character pointers (char *) were used for that. Then people started getting confused about when a character pointer was a string, when it was a character array, and when it was raw memory.

How can type-insensitive macros be created?

A type-insensitive macro is a macro that performs the same basic operation on different

data types.
This task can be accomplished by using the concatenation operator to create a call to a type-sensitive function based on the parameter passed to the macro. The following program provides an example:

```
#include
#define SORT(data_type) sort_ ## data_type
void sort_int(int** i);
void sort_long(long** l);
void sort_float(float** f);
void sort_string(char** s);
void main(void);

void main(void)
{
int** ip;
long** lp;
float** fp;
char** cp;
...
sort(int)(ip);
sort(long)(lp);
sort(float)(fp);
sort(char)(cp);
...
}
```

This program contains four functions to sort four different data types: int, long, float, and string (notice that only the function prototypes are included for brevity). A macro named SORT was created to take the data type passed to the macro and combine it with the sort_ string to form a valid function call that is appropriate for the data type being sorted. Thus, the string
sort(int)(ip);
translates into
sort_int(ip);
after being run through the preprocessor.

When should a type cast not be used?

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 programmer's intentions clearer.

When is a switch statement better than multiple if statements?

A switch statement is generally best to use when you have more than two conditional expressions based on a single variable of numeric type.

What is storage class and what are storage variable ?

A storage class is an attribute that changes the behavior of a variable. It controls the lifetime, scope and linkage.

There are five types of storage classes
1) auto
2) static
3) extern
4) register
5) typedef

What is a static function?
A static function is a function whose scope is limited to the current source file. Scope refers to the visibility of a function or variable. If the function or variable is visible outside of the current source file, it is said to have global, or external, scope. If the function or variable is not visible outside of the current source file, it is said to have local, or static, scope.

How can I sort things that are too large to bring into memory?
A sorting program that sorts items that are on secondary storage (disk or tape) rather than primary storage (memory) is called an external sort. Exactly how to sort large data depends on what is meant by too large to fit in memory. If the items to be sorted are themselves too large to fit in memory (such as images), but there aren't many items, you can keep in memory only the sort key and a value indicating the data's location on disk. After the key/value pairs are sorted, the data is rearranged on disk into the correct order. If too large to fit in memory means that there are too many items to fit into memory at one time, the data can be sorted in groups that will fit into memory, and then the resulting files can be merged. A sort such as a radix sort can also be used as an external sort, by making each bucket in the sort a file. Even the quick sort can be an external sort. The data can be partitioned by writing it to two smaller files. When the partitions are small enough to fit, they are sorted in memory and concatenated to form the sorted file.

What is a pointer variable?
A pointer variable is a variable that may contain the address of another variable or any valid address in the memory.

What is a pointer value and address?
A pointer value is a data object that refers to a memory location. Each memory location is numbered in the memory. The number attached to a memory location is called the address of the location.

What is a modulus operator? What are the restrictions of a modulus operator?
A Modulus operator gives the remainder value. The result of x%y is obtained by (x-(x/y)*y). This operator is applied only to integral operands and cannot be applied to float or double.

Differentiate between a linker and linkage?
A linker converts an object code into an executable code by linking together the necessary build in functions. The form and place of declaration where the variable is declared in a program determine the linkage of variable.

What is a function and built-in function?
A large program is subdivided into a number of smaller programs or subprograms. Each subprogram specifies one or more actions to be performed for a large program. such subprograms are functions.

The function supports only static and extern storage classes. By default, function assumes extern storage class. functions have global scope. Only register or auto storage class is allowed in the function parameters. Built-in functions that predefined and supplied along with the compiler are known as built-in functions. They are also known as library functions.

What is a macro, and how do you use it?

A macro is a preprocessor directive that provides a mechanism for token replacement in your source code. Macros are created by using the #define statement.
Here is an example of a macro: Macros can also utilize special operators such as the stringizing operator (#) and the concatenation operator (##).The stringizing operator can be used to convert macro parameters to quoted strings, as in the following example:

```
#define DEBUG_VALUE(v) printf(#v is equal to %d.n, v)
```

In your program, you can check the value of a variable by invoking the DEBUG_VALUE macro:

```
...
int x = 20;
DEBUG_VALUE(x);
...
```

The preceding code prints x is equal to 20. on-screen. This example shows that the stringizing operator used with macros can be a very handy debugging tool.

What is the difference between goto and longjmp() and setjmp()?

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.

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.

Why should I prototype a function?
A function prototype tells the compiler what kind of arguments a function is looking to receive and what kind of return value a function is going to give back. This approach helps the compiler ensure that calls to a function are made correctly and that no erroneous type conversions are taking place.

What is the quickest searching method to use?
A binary search, such as bsearch() performs, is much faster than a linear search. A hashing algorithm can provide even faster searching. One particularly interesting and fast method for searching is to keep the data in a digital trie. A digital trie offers the prospect of being able to search for an item in essentially a constant amount of time, independent of how many items are in the data set.
A digital trie combines aspects of binary searching, radix searching, and hashing. The term digital trie refers to the data structure used to hold the items to be searched. It is a multilevel data structure that branches N ways at each level.

What are the advantages of auto variables?
1)The same auto variable name can be used in different blocks
2)There is no side effect by changing the values in the blocks
3)The memory is economically used
4)Auto variables have inherent protection because of local scope

What are the characteristics of arrays in C?
1) An array holds elements that have the same data type
2) Array elements are stored in subsequent memory locations
3) Two-dimensional array elements are stored row by row in subsequent memory locations.
4) Array name represents the address of the starting element
5) Array size should be mentioned in the declaration. Array size must be a constant expression and not a variable.

How do you print only part of a string?
/* Use printf() to print the first 11 characters of source_str. */
printf(First 11 characters: '%11.11s'n, source_str);

In C, what is the difference between a static variable and global variable?
A static variable declared outside of any function is accessible only to all the functions defined in the same file (as the static variable). However, a global variable can be accessed by any function (including the ones from different files).

In C, why is the void pointer useful?
When would you use it? The void pointer is useful because it is a generic pointer that any pointer can be cast into and back again without loss of information.

What is Polymorphism ?

'Polymorphism' is an object oriented term. Polymorphism may be defined as the ability of related objects to respond to the same message with different, but appropriate actions. In other words, polymorphism means taking more than one form. Polymorphism leads to two important aspects in Object Oriented terminology - Function Overloading and Function Overriding. Overloading is the practice of supplying more than one definition for a given function name in the same scope. The compiler is left to pick the appropriate version of the function or operator based on the arguments with which it is called. Overriding refers to the modifications made in the sub class to the inherited methods from the base class to change their behavior.

What is Operator overloading ?

When an operator is overloaded, it takes on an additional meaning relative to a certain class. But it can still retain all of its old meanings.
Examples:
1) The operators >> and << may be used for I/O operations because in the header, they are overloaded.
2) In a stack class it is possible to overload the + operator so that it appends the contents of one stack to the contents of another. But the + operator still retains its original meaning relative to other types of data.

What are Templates

C++ Templates allow u to generate families of functions or classes that can operate on a variety of different data types, freeing you from the need to create a separate function or class for each type. Using templates, u have the convenience of writing a single generic function or class definition, which the compiler automatically translates into a specific version of the function or class, for each of the different data types that your program actually uses. Many data structures and algorithms can be defined independently of the type of data they work with. You can increase the amount of shared code by separating data-dependent portions from data-independent portions, and templates were introduced to help you do that.

What is the difference between run time binding and compile time binding?

Dynamic Binding :

The address of the functions are determined at runtime rather than @ compile time. This is also known as "Late Binding".

Static Binding :

The address of the functions are determined at compile time rather than @ run time. This is also known as "Early Binding"

What is Difference Between C/C++

C does not have a class/object concept.
C++ provides data abstraction, data encapsulation, Inheritance and Polymorphism.
C++ supports all C syntax.
In C passing value to a function is "Call by Value" whereas in C++ its "Call by Reference"
File extension is .c in C while .cpp in C++.(C++ compiler compiles the files with .c extension but C compiler can not!)
In C structures can not have contain functions declarations. In C++ structures are like

classes, so declaring functions is legal and allowed.
C++ can have inline/virtual functions for the classes.
c++ is C with Classes hence C++ while in c the closest u can get to an User defined data type is struct and union.

Why doesn't the following code give the desired result?

```
int x = 3000, y = 2000 ;
long int z = x * y ;
```

Here the multiplication is carried out between two ints x and y, and the result that would overflow would be truncated before being assigned to the variable z of type long int. However, to get the correct output, we should use an explicit cast to force long arithmetic as shown below:

```
long int z = ( long int ) x * y ;
```

Note that (long int)(x * y) would not give the desired effect.

Why doesn't the following statement work?

```
char str[ ] = "Hello" ;
strcat ( str, '!' ) ;
```

The string function strcat( ) concatenates strings and not a character. The basic difference between a string and a character is that a string is a collection of characters, represented by an array of characters whereas a character is a single character. To make the above statement work writes the statement as shown below:

```
strcat ( str, "!" ) ;
```

How do I know how many elements an array can hold?

The amount of memory an array can consume depends on the data type of an array. In DOS environment, the amount of memory an array can consume depends on the current memory model (i.e. Tiny, Small, Large, Huge, etc.). In general an array cannot consume more than 64 kb. Consider following program, which shows the maximum number of elements an array of type int, float and char can have in case of Small memory model.

```
main( )
{
int i[32767] ;
float f[16383] ;
char s[65535] ;
}
```

How do I write code that reads data at memory location specified by segment and offset?

Use peekb( ) function. This function returns byte(s) read from specific segment and offset locations in memory. The following program illustrates use of this function. In this program from VDU memory we have read characters and its attributes of the first row. The information stored in file is then further read and displayed using peek( ) function.

```
#include <stdio.h>
#include <dos.h>
main( )
{
char far *scr = 0xB8000000 ;
FILE *fp ;
```

```
int offset ;
char ch ;
if ( ( fp = fopen ( "scr.dat", "wb" ) ) == NULL )
{
printf ( "\nUnable to open file" ) ;
exit( ) ;
}
/ / reads and writes to file
for ( offset = 0 ; offset < 160 ; offset++ )
fprintf ( fp, "%c", peekb ( scr, offset ) ) ;
fclose ( fp ) ;
if ( ( fp = fopen ( "scr.dat", "rb" ) ) == NULL )
{
printf ( "\nUnable to open file" ) ;
exit( ) ;
}
/ / reads and writes to file
for ( offset = 0 ; offset < 160 ; offset++ )
{
fscanf ( fp, "%c", &ch ) ;
printf ( "%c", ch ) ;
}
fclose ( fp ) ;
}
```

What is conversion operator?
class can have a public method for specific data type conversions.
for example:

```
class Boo
{
double value;
public:
Boo(int i )
operator double()
{
return value;
}
};

Boo BooObject;
```

double i = BooObject; / / assigning object to variable i of type double. now conversion operator gets called to assign the value.

What is diff between malloc( ) / free( ) and new / delete?
malloc allocates memory for object in heap but doesn't invoke object's constructor to initiallize the object.

new allocates memory and also invokes constructor to initialize the object.

malloc() and free() do not support object semantics
Does not construct and destruct objects
string * ptr = (string *)(malloc (sizeof(string)))
Are not safe
Does not calculate the size of the objects that it construct
Returns a pointer to void
int *p = (int *) (malloc(sizeof(int)));
int *p = new int;
Are not extensible
new and delete can be overloaded in a class

"delete" first calls the object's termination routine (i.e. its destructor) and then releases the space the object occupied on the heap memory. If an array of objects was created using new, then delete must be told that it is dealing with an array by preceding the name with an empty []:-

```
Int_t *my_ints = new Int_t[10];

...

delete []my_ints;
```

what is the diff between "new" and "operator new" ?
"operator new" works like malloc.

What is difference between template and macro??
There is no way for the compiler to verify that the macro parameters are of compatible types. The macro is expanded without any special type checking.

If macro parameter has a postincremented variable (like c++), the increment is performed two times.

Because macros are expanded by the preprocessor, compiler error messages will refer to the expanded macro, rather than the macro definition itself. Also, the macro will show up in expanded form during debugging.

for example:
Macro:

```
#define min(i, j) (i < j ? i : j)
```

template:

```
template<class T>
T min (T i, T j)
{
return i < j ? i : j;
}
```