

Q. 1) What is the difference between `#include <file>` and `#include "file"` ?

Answer: 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:COMPILERINCLUDE;  
S:SOURCEHEADERS;
```

using the `#include <file>` version of file inclusion, the compiler first checks the `C:COMPILERINCLUDE` directory for the specified file. If the file is not found there, the compiler then checks the `S:SOURCEHEADERS` 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 <file>` 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.

Q. 2) C is a structural or high level or middle level language which one is correct?

Answer: C is a middle level language, because this language contains both the features of both high level language and low level languages, also can be called as structured programming language.

Q. 3) Is it possible to execute code even after the program exits the `main()` function?

Answer: 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 `atexit()` function.

Q. 4) how to create love flames program using c language?

Answer:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    int i,j,n,c,s=0,p,t,y;
    char a[25],b[25],m,q[25],w[25];
    clrscr();
    printf("Enter Ur Name");
    scanf("%s",a);

    printf("Enter Ur Partner's Name");
    scanf("%s",b);
    strcpy(q,a);
    strcpy(w,b);
    n=strlen(a);
    c=strlen(b);
    for(i=0;i<n;i++)
    {
        m=a[i];
        for(j=0;j<c;j++)
        {
            if(m==b[j])
            {
                a[i]=-1;
                b[j]=-1;
                s=s+2;
                break;
            }
        }
    }
    p=n+c;
    puts(a);
    puts(b);
    t=p-s;
```

```
printf("The count value is %d",t);
if(t==2 || t==4 || t==7 ||t==9 )
    printf("%s is ENEMY to %s ",q,w);
else if (t==3 || t==5 || t==14)
    printf("%s is FRIEND to %s ",q,w);
else if(t==6 || t==11 || t==15 )
    printf("%s is going to MARRY %s ",q,w);
else if(t==10)
    printf("%s is in LOVE with %s ",q,w);
else if(t==8)
    printf("%s has more AFFECTION on %s ",q,w);
else
    printf("%s and %s are SISTERS/BROTHERS ",q,w);
getch();
}
```

Q. 5) How do you override a defined macro?

Answer: Assume 'NULL' is framework defined MACRO. If you wish to override, then,

```
#ifdef NULL
#undef NULL
#define NULL ((void*)0)
#endif
```

above will override already defined MACRO in the scope.

Q. 6) Can a file other than a .h file be included with #include?

Answer: The preprocessor will include whatever file you specify in your #include statement. Therefore, if you have the line #include <macros.inc> 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.

Q. 7) What is the difference between `NULL` and `NUL`?

Answer: `NULL` is a macro defined in `<stddef.h>` 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 `'\0'`.

Q. 8) Write the equivalent expression for `x%8`?

Answer: `x-(8 * (x/8))`

Q. 9) How can you avoid including a header more than once?

Answer: 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.

Q. 10) How to write a C program for displaying a sentence without output command?

Answer: we can do it in many ways

i) you will get only this sentence

```
#include<stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    int a=10,b;
    b=a/0;
}
```

output: some text

ii) in this you can get any output

```
void display()
{
    printf("kishore reddy");
}
```

save above program in file and save it as filename.h

```
#include<stdio.h>
#include "filename.h"
void main()
{
    display();
}
```

Q. 11) is there anything you can do in C++ that you cannot do in C?

Answer: The thing which are not supported by C are:

- i) Inheritance
 - ii) Polymorphism
 - iii) Operator overloading
 - iv) Abstraction & Encapsulation (data hiding)
- and other oop's feature.

Q. 12) How are portions of a program disabled in demo versions?

Answer: 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 program!\n");
    return(0);
    #endif
    ...
}
```

Q. 13) What will be printed as the result of the operation below:

```
main()
{
    char *ptr = " Cisco Systems";
    *ptr++;
    printf("%s\n", ptr);
    ptr++;
    printf("%s\n", ptr);
}
```

Answer: Cisco Systems
isco systems

Q. 14) is it possible to print graphics output in C?

Answer: yes, we can print graphics output using c by using APIs defined in `#include<graphics.h>`

Q. 15) What is the benefit of using `#define` to declare a constant?

Answer: 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.

Q. 16) What are the different storage classes in C?

Answer: 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.

Q. 17) What is the difference between `#define` and `constant` in C?

Answer: A `const` identifier allows a variable to be constant throughout the file in which it's declared. Where as `#define` is a macro which replaces the macro-name with a user-defined text statement. Also, a macro can be deleted, if we wish, using `#undef` whereas a `const` variable once defined can't be done so.

Q. 18) What is a `pragma`?

Answer: 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)`

Q. 19) Difference between C and C++?

Answer: C is a procedure oriented language.
C++ is an object oriented language.

Q. 20) What is the use of semicolon (;) at the end of every statement?

Answer: If semicolon comes in a statement means its ending point of the statement.

Q. 21) Define structural language and procedural language?

Answer: There are two types of structural language.
i) Procedure language like c
ii) Object Oriented like c++, java

Q. 22) The following variable is available in file1.c, who can access it?

```
static int average;
```

Answer: all the functions in the file1.c can access the variable.

Q. 23) Can you define which header file to include at compile time?

Answer: 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 <alloc.h>  
#else  
#include <malloc.h>  
#endif
```

Q. 24) How can we use data connectivity in 'c' language?

Answer: Using Pro *C we can do the data connectivity.

Q. 25) How can C programs run without header files?

Answer: C programming run without header file write example given below.

```
void main()
{
    clrscr();
    printf("hello");
    getch();
}
```

Q. 26) if compiler for C is written in C language, which is used to compile the compiler?

Answer: C compiler was written in C language, that is true. But that does not say in what language first C compiler was written. I think first C compiler might have used assembly language instructions. And after that further compilers might have used this (or other compilers that was built the same way) compiler to compile the other newly build compilers.

Q. 27) what is Preprocessor?

Answer: 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.

Q. 28) What are the standard predefined macros?

Answer: 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.

__DATE__ 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.

Q. 29) Can you write a c program without using main function?

Answer: we can write the program without main but it can't run because linking is from main().

Q. 30) What will be printed as the result of the operation below:

```
main()
{
    int a=0;
    if(a==0)
        printf("Cisco Systems\n");
    printf("Cisco Systems\n");
}
```

Answer: Two lines with "Cisco Systems" will be printed.

Q. 31) What is object file? How can you access object file?

Answer: When a c program is compiled the source code is converted into object file i.e., a.out this file is known as object file.

Q. 32) What are storage class in c?

Answer: There are of 4 type of storage class in C

- Static
- auto
- register
- extern

Q. 33) Write a C program to print that a number is odd or even?

Answer:

```
main()
{
    int arr[10];
    int i, oddcount=0, evencount=0;
    for(i=0; i<10; i++)
    {
        scanf("%d", &arr[i]);
    }
    for(i=0; i<10; i++)
    {
        if(arr[i]%2)
            oddcount++;
        else
            evencount++;
    }
    printf("odd count=%d and even count=%d", oddcount, evencount);
    return 0;
}
```

Q. 34) What is #line used for?

Answer: 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.

Q. 35) what is the difference between int and long int in C?

Answer: int allocate 2bytes memory
long int allocate 4 bytes memory

Q. 36) How to find a given number is Armstrong number or not in "c"?

Answer: given no
n=153, n1=n;
while(n>=0)
{
 r=n%10;
 sum+=r*r*r;
 n=n/10;
}
if(n1==sum)
 then Armstrong
else
 not

Q. 37) How to print value without using any output statements?

Answer: We can also print the output of a program in another way.
First you save the program with filename.h
later use this filename in another program using this statement.

```
#include "filename.h"
```

so that now we can print the output of that program.

For example:

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

save above program in add.h

in another file

```
#include<stdio.h>  
#include"add.h"  
#include<conio.h>  
int process(int &,int &);  
void main()  
{  
    clrscr();  
    int a,b;  
    printf("enter two numbers");  
    scanf("%d%d",a,b);  
    printf("%d",process(a,b);  
    getch();  
}
```

save above program as filename2.c

Q. 38) How can type-insensitive macros be created?

Answer: 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 <stdio.h>
#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);
    ...
}
```

Q. 39) What will print out?

```
main()
{
    char *p1="name";
    char *p2;
    p2=(char*)malloc(20);
    memset(p2, 0, 20);
    while(*p2++ = *p1++);
    printf("%s\n", p2);
}
```

Answer: Empty string.

Q. 40) What is modular programming?

Answer: 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.

Q. 41) What is Assembler, compiler, Preprocessor, lexical analysis, parsing ?

Answer:

Assembler: It converts the assembly language into machine level.

Compiler: It converts the high level program into machine level.

Preprocessor: Already defined using #define and used later in the program.

Lexical analysis: It analyze the expression from left to right and separate into tokens.

Parsing: It groups the token collected from the lexical analysis and constructs parse tree.

Q. 42) what will be the result of the following code?

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

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

Q. 43) What are the characteristics of arrays in C?

Answer:

- 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.
- 6) While declaring the 2D array, the number of columns should be specified and its a mandatory. whereas for number of rows there is no such rule.

Q. 44) What is the difference between compile time error and run time error?

Answer: If the program has the compile time error then the program won't execute.

Ex: syntax or semantic errors.

If the program has the runtime error then the program would compile successfully with no errors. But the output was not desired.

Ex: dividing with zero.

Q. 45) How to perform addition, subtraction of 2 numbers without using addition and subtraction operators?

Answer: yes we can

- 1) using \times or
- 2) $a--b$
- 3) $a2-b2/a-b$

Q. 46) How do you write a program which produces its own source code as its output?

Answer: open the selfcopy.c file inside its code---selfcopy.c---

```
main()
{
    FILE *fopen(), *fp;
    int c ;
    printf("THE OUTPUT IS THE PROGRAM ITSELF\n");
    fp = fopen("selfcopy.c","r");
    c= getc( fp ) ;
    while ( c != EOF )
    {
        putchar( c );
        c = getc ( fp );
    }
    fclose( fp );
}
```

Q. 47) Can we execute printf statement without using semicolon?

Answer: `if(printf("print anything")){}`

Q. 48) What is a macro, and how do you use it?

Answer: 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.

Q. 49) What will the preprocessor do for a program?

Answer: 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.

Q. 50) Can any other language completely replace c?

Answer:

- The primary design of C is to produce portable code while maintaining performance and minimizing footprint, as is the case for operating systems or other programs where a "high-level" interface would affect performance.

- It is a stable and mature language whose features are unlikely to disappear for a long time and has been ported to most, if not all, platforms.

For example, C programs can be compiled and run on the HP 50g calculator (ARM processor), the TI-89 calculator (68000 processor), Palm OS Cobalt smart phones (ARM processor). While nearly all popular programming languages will run on at least one of these devices, C may be the only programming language that runs on more than 3 of these devices.

- One powerful reason is memory allocation. Unlike most computer languages, C allows the programmer to write directly to memory.
- C gives control over the memory layout of data structures. Moreover dynamic memory allocation is under the control of the programmer, which inevitably means that memory de-allocation is the burden of the programmer.
- While Perl, PHP, Python and Ruby may be powerful and support many features not provided by default in C, they are not normally implemented in their own language. *Rather, most such languages initially relied on being written in C (or another high-performance programming language), and would require their implementation be ported to a new platform before they can be used.
- As with all programming languages, whether you want to choose C over another high-level language is a matter of opinion and both technical and business requirements.

Q. 51) What will be printed as the result of the operation below:

```
main()
{
    char *p1;
    char *p2;
    p1=(char *)malloc(25);
    p2=(char *)malloc(25);
    strcpy(p1,"Cisco");
    strcpy(p2,"systems");
    strcat(p1,p2);
    printf("%s",p1);
}
```

Answer: Ciscosystems

Q. 52) When should the register modifier be used? Does it really help?

Answer: 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.

Q. 53) How do you print an address?

Answer: 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( "%P\n", (void*) buffer );
```