Problem Solving with C

Compiled by

M S Anand (anandms@pes.edu)

Text Book(s):

1. “How To Solve It By Computer”, R G Dromey, Pearson, 2011.
2. “The C Programming Language”, Brian Kernighan, Dennis Ritchie, 2nd Edition, Prentice Hall PTR, 1988.

Reference Book(s):

1. “Expert C Programming; Deep C secrets”, Peter van der Linden
2. “ The C puzzle Book”, Alan R Feuer

Unions

User defined data type just like a structure.

```
union sample
{
    char name [4];
    long  length;
    short s1;
};

union sample u1, u2;
```

A union, like a structure consists of one or more members, possibly of different types. However, the compiler allocates only enough space for the largest of the members, which overlay each other within this space.

Assigning a new value to one member alters the value of other members as well.

How do you access the members of a union?
Just like members of a structure

u1.name
u1.length
etc

Union can be looked at as a place to store any one of its members (not all (as in structures)).

Properties of unions

Almost the same as for structures:

Union tags and union types can be declared.

Unions can be copied using the = operator

Can be passed to functions

Can be returned from functions

Unions can be initialized like structures. Only the first member can be given an initial value.

Designated initializers just like in structures – only one member can be initialized but it need not be the first one.

Similarites between structures and unions

1. Both are user-defined data types used to store data of different types as a single unit.
2. Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
3. Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
4. A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
5. **‘.’** operator is used for accessing members.

Differences between structures and unions

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Illustrate the difference between a structure and a union in terms of memory layout.

```
struct samp
{
    int i;
    double d;
} s1;
```

```
union samp1
{
    int i;
    double d;
} d1;
```

The structure will be allocated 16 bytes whereas the union will be allocated 8 bytes only.

The structure members i and d will start at separate addresses while the union members i and d will start at the same address.

A few sample programs

Program1

Program2

Program3

Using unions to save space

We often use unions to save space in structures.

Problem

3 items are sold through a gift catalog – books, mugs and shirts.
Each item has a stock number and a price and other data depending on the item:

Books – Title, Author, no of pages
Mugs – Design
Shirts – Design, colors available, sizes available

If we are asked to design a data structure to handle this, our first attempt would result in a structure like this (next slide)

```
struct catalog_item
{
    int stock_number;
    double price;
    int item_type;
    char title [50];
    char author [40];
    int num_pages;
    char design [40];
    int colors;
    int sizes;
};
```

item_type would be one of BOOK, MUG or SHIRT
The colors and sizes members would store encoded combination of colors and sizes.
This structure wastes space – HOW?

Only part of the information is common to all the items in the catalog.

If the item is a book, colors, sizes and design members are not relevant.

How do we bring in union to save space?

```
struct catalog_item
{
    int stock_number;
    double price;
    int item_type;
    union
    {
        struct
        {
            char title [50];
            char author [40];
            int num_pages;
        } book;
        struct
        {
            char design [40];
        } mug;
        struct
        {
            char design [40];
            int colors;
            int sizes;
        } shirt;
    } item;
};
```

How do you access the members of this structure?

If c is a catalog_item structure

Book's title can be accessed as:

c.item.book.title

Using unions to build mixed data structures

Suppose we need an array whose elements are a mixture of int and double values.

Impossible?

Use unions

```
typedef union
{
    int i;
    double d;
} Number;

Number Number_array [10];
Number_array[0].i = 100;
Number_array[1].d = 1234.56;
```

Adding a tag field to a union

Unions suffer from a major problem:

No easy way to tell which member of the union was modified last and therefore contains a meaningful value.

In order to keep track of this information, we can embed the union within a structure that has one other member: a tag field or a **discriminant** – its purpose is to let us know which member of the union was modified last and hence contains a meaningful value;

```
#define INT_KIND            0
#define DOUBLE_KIND         1
typedef struct
{
    int kind;
    union
    {
        int i;
        double d;
    } u;
} Number;
```

How do we use the kind field?

```
void print_number (Number n)
{
    if (n.kind == INT_KIND)
        printf (“The integer stored is %d\n”, n.u.i)
    else
        printf (“The double value is %f\n”, n.u.d);
}
```

So, it is the programmer’s responsibility to update the “kind” field every time one of the members of the union is modified.

Bit fields

In C, we can specify size (in bits) of **structure and union members**. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.

A sample program without using bit fields for date is here.

The above representation of 'date' takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, value of m is from 1 to 12, we can optimize the space using bit fields.

Program using bit fields is here.

Some interesting facts about bit fields

We cannot have pointers to bit field members as they may not start at a byte boundary. Sample program.

Array of bit fields is not allowed. For example, the below program fails in compilation. The program.

Storage classes in C

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:
auto, **extern**, **static** and **register**.

auto

This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language.

Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared.

However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables reside.

They are assigned a garbage value by default whenever they are declared.

extern:
Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used.

Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block.

Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

static

This storage class is used to declare static variables which are popularly used while writing programs in C language.

Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared.

Their scope is local to the function to which they were defined.

Global static variables can be accessed anywhere in the program.

By default, they are assigned the value 0 by the compiler.

register

This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.

If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program.

An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

volatile

C's volatile keyword is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time--without any action being taken by the code the compiler finds nearby. The implications of this are quite serious.

Syntax of C's volatile Keyword

To declare a variable volatile, include the keyword volatile before or after the data type in the variable definition.

```
volatile int x;
int volatile y;
```

If you apply volatile to a struct or union, the entire contents of the struct or union are volatile. If you don't want this behavior, you can apply the volatile qualifier to the individual members of the struct or union.

Proper Use of C's volatile Keyword

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

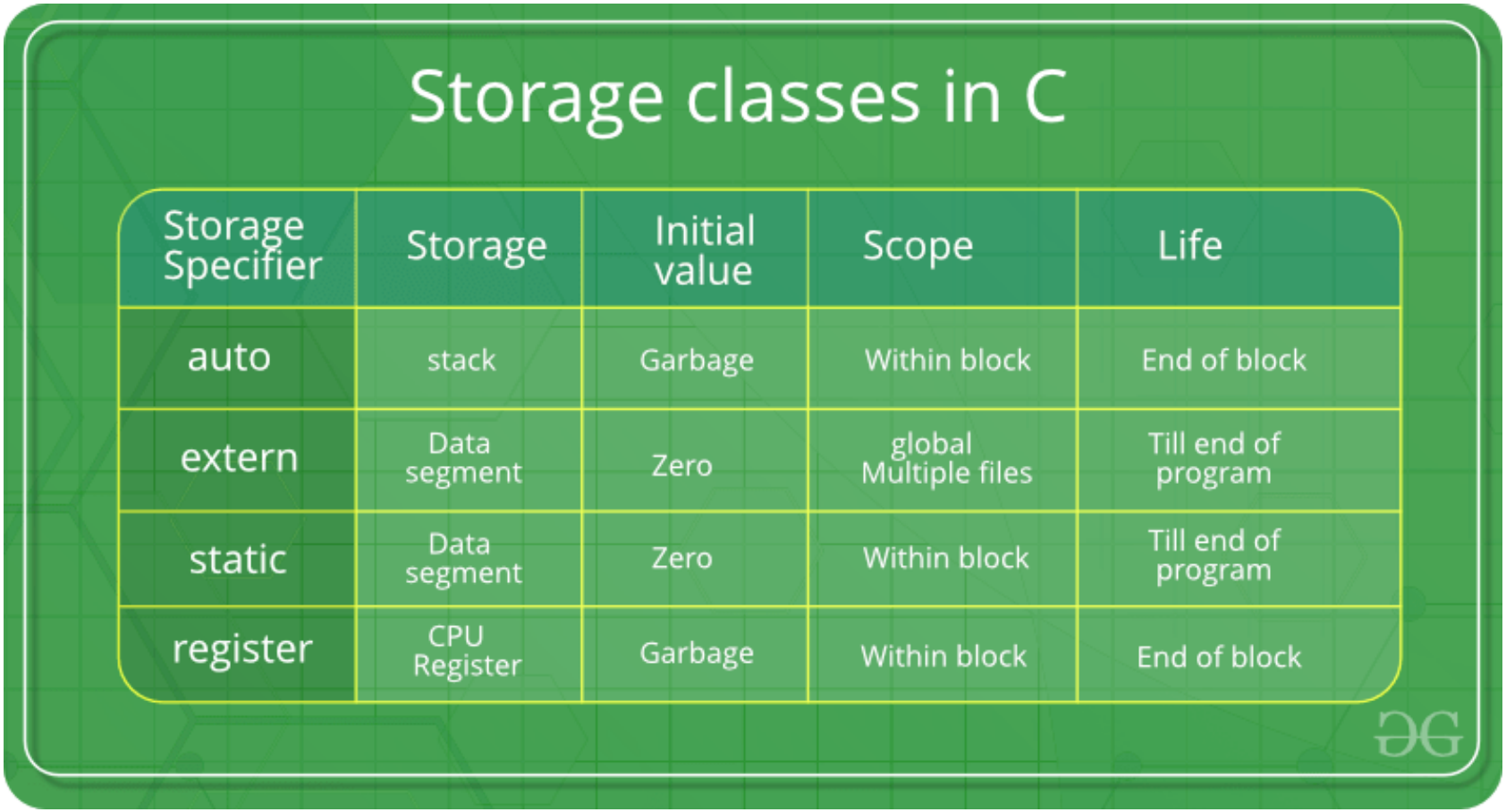
1. Memory-mapped peripheral registers
2. Global variables modified by an interrupt service routine
3. Global variables accessed by multiple tasks within a multi-threaded application

To specify the storage class for a variable, the following syntax is to be followed:

Syntax:
storage_class var_data_type var_name;

A program to demonstrate these concepts is here and here.

Summary


Storage classes in C
Storage Specifier
Storage
Initial value
Scope
Life
auto
stack
Garbage
Within block
End of block
extern
Data segment
Zero
global Multiple files
Till end of program
static
Data segment
Zero
Within block
Till end of program
register
CPU Register
Garbage
Within block
End of block

Function pointers in C

How to declare a function pointer?

function_return_type(*Pointer_name)(function argument list)

For example:
double (*p2f)(double, char)

Here double is a return type of function, p2f is name of the function pointer and (double, char) is an argument list of this function - the first argument of this function is of double type and the second argument is char type.

```
int sum (int num1, int num2)
{
    return num1+num2;
}

int main(void)
{
    int (*f2p) (int, int);
    f2p = sum;
    //Calling function using function pointer
    int op1 = f2p(10, 13);
    //Calling function in normal way using function name
    int op2 = sum(10, 13);
    printf("Output1: Call using function pointer: %d",op1);
    printf("\nOutput2: Call using function name: %d", op2);
    return 0;
}
```

Callbacks in C

A callback is any executable code that is **passed as an argument to other code, which is expected to call back (execute) the argument at a given time** [Source : Wiki].

In simple language, if a reference of a function is passed to another function as an argument to call it, then it will be called as a Callback function.

In C, a callback function is a function that is called through a function pointer.

```
// A simple C program to demonstrate callback
#include<stdio.h>
void A()
{
    printf("I am function A\n");
}
// callback function
void B(void (*ptr)())
{
    ptr (); // callback to A
}
int main(void)
{
    void (*ptr)() = A;
     // calling function B and passing
    // address of the function A as argument
    B(ptr);
    return 0;
}
```

Here is a program which uses an array of function pointers

Program

Command line arguments

The arguments that we pass on to main() are called command line arguments.

The full declaration of main looks like this:
int main (int argc, char *argv[])

The function main() can have two arguments, traditionally named as argc and argv. Out of these, argv is an array of pointers to strings and argc is an int whose value is equal to the number of strings to which argv points.

What is argv [0]?

Program to print the command line arguments

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

    if (argc < 2)
    {
        printf (“The name of the program is %s\n”, argv[0]);
    }
    else
    {
        for (i = 0;i < argc; i ++)
            printf (“Argc %d is %s\n”, i, argv[i]);
    }
    return 0;
}
```

Environment variables

Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer.

C program to print environment variables

```
#include <stdio.h>
int main(int argc, char *argv[], char * envp[])
{
    int i;

    for (i = 0; envp[i] != NULL; i++)
        printf("\n%s", envp[i]);
    return 0;
}
```

Second way to get the list of environment list is using the environ variable.

```
extern char **environ;
```

Functions to get and set environment variables

#include <stdlib.h>

char *getenv(const char *_name_);

The **getenv**() function searches the environment list to find the environment variable *name*, and returns a pointer to the corresponding *value* string.

The **getenv**() function returns a pointer to the value in the environment, or NULL if there is no match.

#include <stdlib.h>

int putenv(char *_string_);

The **putenv**() function adds or changes the value of environment variables. The argument *string* is of the form *name*=*value*. If *name* does not already exist in the environment, then *string* is added to the environment. If *name* does exist, then the value of *name* in the environment is changed to *value*. The string pointed to by *string* becomes part of the environment, so altering the string changes the environment.

The **putenv**() function returns zero on success, or nonzero if an error occurs.

A sample program is here.

Portable program development

A portable application is a software product designed to be easily moved from one computing environment to another.

C preprocessor

It is a piece of software which edits the C programs just before compilation.

The working of the preprocessor

The behavior of the preprocessor is controlled by preprocessing directives are commands that begin with a # character.
You are familiar with 2 of them: #define and #include

#define – defines a **macro** a name that represents something else, such as a constant or a frequently used expression.

The preprocessor responds to a #define directive by storing the name of the macro together with its definition.

When the macro is used later in the program, the preprocessor “expands” the macro, replacing it by its defined value.

The #include directive tells the preprocessor to open a particular file and “include” its contents as part of the file being compiled.

The input to the preprocessor is a C program, possibly containing directives. The preprocssor executes these directives, removing them in the process. The output of the preprocessor is another C program: an edited version of the earlier program with no directives. The output goes to the compiler as input.

You can see the output from the preprocessor by using the following command:

gcc –E sample.c

Preprocessing directives

Most preprocessing directives fall into one of the three following categories:

Macro definition: The **#define** directive defines a macro; **#undef** directive removes a macro definition

File inclusion: The **#include** directive causes the contents of a specified file to be included in a program.

Conditional compilation: The **#if**, #**ifdef**, #**ifndef**, #**else**, #**elif** and #**endif** directives allow blocks of text to be either included in or excluded from a program, depending on conditions that can be tested by the preprocessor.

Rules that apply to all directives:

1. **Directives always begin with the '#' symbol** – The # symbol need not be at the beginning of a line, as long as only white spaces precede it. After the # comes the name of the directive, followed by any other information the directive requires.

2. **Any number of spaces and horizontal tabs may separate the tokens in a directive**. For example

```
#     define          NUMBER        100
```

3. **Directives always end at the first new-line character, unless explicitly continued** To continue a directive to the next line, we must end the current line with a \ character. An example:

```
#define   DISK_CAPACITY   (SIDES*\
                          TRACKS_PER_SIDE * \
                           SECTORS_PER_TRACK *\
                           BYTES_PER_SECTOR)
```

4. **Directives can appear anywhere in a program –**

5. **Comments may appear on the same line as a directive**

```
#define  RATE     3.5f /* Rate of interest */
```

Macro definitions

Simple macros

#define identifier replacement-list

replacement-list is any sequence of pre-processing tokens

A macro's replacement list may include identifiers, keywords, numeric constants, character constants, string literals, operators and punctuation. When it encounters a macro definition, the preprocessor makes a note that identifier represents replacement-list; whenever identifier appears later in the file, the preprocessor substitutes the replacement-list.

Caution:
Don't put any extra symbols in a macro definition – they will become part of the macro definition.

```
#define  N=100    /* is a common mistake */

int a [N]   becomes   int a [=100]
```

Another common mistake

```
#define  N 100;       /* is a second common mistake */
int a [N] becomes int a [100;];
```

Simple macros are primarily used for giving names to numeric, character and string values.

Standard Predefined Macros

The standard predefined macros are specified by the relevant language standards, so they are available with all compilers that implement those standards. Their names all start with double underscores.

__FILE__ This macro expands to the name of the current input file, in the form of a C string constant. This is the path by which the preprocessor opened the file, not the short name specified in '#include' or as the input file name argument. For example, "/usr/local/include/myheader.h" is a possible expansion of this macro.

__LINE__ This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

__FILE__ and __LINE__ are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected.

__DATE__ This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like "Feb 12 1996". If the day of the month is less than 10, it is padded with a space on the left.

__TIME__ This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like "23:59:01".

__STDC__ In normal operation, this macro expands to the constant 1, to signify that this compiler conforms to ISO Standard C.

__STDC_VERSION__ This macro expands to the C Standard's version number, a long integer constant of the form *yyyymm*L where *yyyy* and *mm* are the year and month of the Standard version. This signifies which version of the C Standard the compiler conforms to.

Sample program is here.

Parameterized macros:
The definition of parameterized macro (also known as a **function-like macro**) has the form:

#define identifier(x1,x2,x3, ...x10,..) replacement-list

Where x1, x2, .. are identifiers (the macro's parameters). The parameters may appear as many times as desired in the replacement list.

Note:
#define MAX(x,y) ((x) > (y) ? (x): (y))

Advantages of parameterized macros over true functions:
The program may be slightly faster – A function call involves some overhead during program execution – context information must be saved, arguments copied and so forth.

Macros are 'generic' – Macro parameters unlike function parameters have no particular type. As a result a macro can accept arguments of any type, provided that the resulting program – after preprocessing is valid. We could use the MAX macro to find the larger of two values int, long, float, double, ...

Disadvantages
The compiled code will often be larger – Each macro invocation causes the insertion of the macro's replacement list, thereby increasing the size of the source program. The more often the macro is used, the more pronounced this effect is. The problem is compounded when the macro invocations are nested.

Example

When we use macro MAX to find the maximum of three integers

```
n = MAX (i, MAX(j, k));
```

This statement, after preprocessing would look like this:

```
n = ((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```

Arguments aren't type-checked – Macro arguments are neither checked by the preprocessor nor are they converted.

It's not possible to have a pointer to a macro -

A macro may evaluate its arguments more than once – Evaluating an argument more than once can cause unexpected behaviour if the argument has side-effects.

```
n = MAX (i++, j);
```

After preprocessing, the same line looks like this:

```
n = (i++ ) > (j) ? (i++):j);
```

What is the problem here?

If i is larger than j, then i will be (incorrectly) incremented twice and n will be assigned an unexpected value.

Conditional Compilation (#if, #ifdef, #ifndef, #else, #elif, #endif and defined operator)

Six directives are available to control conditional compilation. They delimit blocks of program text **that are compiled only if a specified condition is true**. These directives can be nested. The program text within the blocks is arbitrary and may consist of preprocessor directives, C statements, and so on.

The beginning of the block of program text is marked by one of three directives:
#if
#ifdef
#ifndef
Optionally, an alternative block of text can be set aside with one of two directives:
#else
#elif
The end of the block or alternative block is marked by the #endif directive.

If the condition checked by #if , #ifdef , or #ifndef is true (nonzero), then all lines between the matching #else (or #elif) and an #endif directive, if present, are ignored.

If the condition is false (0), then the lines between the #if , #ifdef , or #ifndef and an #else , #elif , or #endif directive are ignored.

The #if Directive

The #if directive has the following syntax:

#if *constant-expression newline*

This directive checks whether the *constant-expression* is true (nonzero). The operand must be a constant integer expression that does not contain any increment (++), decrement (- -), sizeof , pointer (*), address (&), and cast operators.

The #ifdef Directive

The #ifdef directive has the following syntax:

#ifdef *identifier newline*

This directive checks whether the identifier is currently defined. Identifiers can be defined by a #define directive or on the command line. If such identifiers have not been subsequently undefined, they are considered currently defined.

The #ifndef Directive

The #ifndef directive has the following syntax:

#ifndef *identifier newline*

This directive checks to see if the identifier is not currently defined.

The #else Directive

The #else directive has the following syntax:

#else *newline*

This directive delimits alternative source text to be compiled if the condition tested for in the corresponding #if , #ifdef , or #ifndef directive is false. An #else directive is optional.

The #elif Directive

The #elif directive has the following syntax:

#elif *constant-expression newline*

The #elif directive performs a task similar to the combined use of the else-if statements in C. This directive delimits alternative source lines to be compiled if the constant expression in the corresponding #if , #ifdef , #ifndef , or another #elif directive is false and if the additional constant expression presented in the #elif line is true. An #elif directive is optional.

The #endif Directive

The #endif directive has the following syntax:

#endif *newline*

This directive ends the scope of the #if, #ifdef, #ifndef, #else or #elif directive

The defined Operator

Another way to verify that a macro is defined is to use the **defined** unary operator. The **defined** operator has one of the following forms:

defined *name*
defined (*name*)

An expression of this form evaluates to 1 if *name* is defined and to 0 if it is not.

The **defined** operator is especially useful for checking many macros with just a single use of the #if directive. In this way, you can check for macro definitions in one concise line without having to use many #ifdef or #ifndef directives.

For example, consider the following macro checks:

```
#ifdef macro1
        printf( "Hello!\n" );
#endif
#ifndef macro2
        printf( "Hello!\n" );
#endif
#ifdef macro3
        printf( "Hello!\n" );
#endif
```

Another use of the **defined** operator is in a single #if directive to perform similar macro checks:

```
#if defined (macro1) || !defined (macro2) || defined (macro3)
        printf( "Hello!\n" );
#endif
```

Note that **defined** operators can be combined in any logical expression using the C logical operators. However, defined can only be used in the evaluated expression of an #if or #elif preprocessor directive.

What would be output of following program?

```
#include <stdio.h>
#define A 10
#define B 40
int main(void)
{
#if A==B
    printf("Hello");
#elif A>B
    printf("World");
#else
    printf("CodingFox");
#endif
   return 0;
}
```

What would be output of following program?

```
#include <stdio.h>
#define X 45
int main(void)
{
#if !X
     printf("Hello");
#else
     printf("World");
#endif
    return 0;
}
```

What would be the output of following program?

```
#include <stdio.h>
#define X 45.25
int main(void)
{
#if !X
 printf("Hello");
#else
 printf("World");
#endif
    return 0;
}
```

What would be the output of following program?

```
#include <stdio.h>
int main(void)
{
#if sizeof(short)==2
 printf("Hello");
#else
 printf("World");
#endif
    return 0;
}
```

What would be the output of following program?

```
#include <stdio.h>
#define MAX
int main(void)
{
#if defined(MAX)
 printf("Hello");
#else
 printf("World");
#endif
 printf("\n");
#if !defined(MIN)
 printf("Fox");
#else
 printf("Duck");
#endif
    return 0;
}
```

Miscellaneous directives

The #error directive

It has the form:

#error **message**

If the preprocessor encounters a #error directive, it prints an error message which must include **message**.

Encountering an #error directive indicates a serious flaw in the program. #error directives are usually used in conjunction with conditional compilation to check for situations that shouldn't arise during a normal compilation.

```
#if  INT_MAX  < 100000
#error int type is too small
#endif
```

Attempting to compile the program on a machine whose integers are stored in 16 bits will produce a message such as:

Error directive: int type is too small

#pragma directive

The #pragma directive provides a way to request special behaviour from the compiler.

It has the form

```
#pragma     tokens
```

Where tokens are arbitrary tokens.

An example

```
#pragma  data (heap_size => 1000, stack_size => 2000)
```

The set of commands that can appear in #pragma directive is different for each compiler.

Another example
#pragma GCC poison: This directive is supported by the GCC compiler and is used to remove an identifier completely from the program. If we want to block an identifier then we can use the **#pragma GCC poison** directive.

A program which uses the #pragma directive is here.

The above program will give the below error when you try to compile

```
poison_prog.c: In function 'main':
poison_prog.c:13:9: error: attempt to use poisoned "printf"
        printf("PES University");
        ^
poison_prog.c:16:9: error: attempt to use poisoned "printf"
        printf("bye");
        ^
```

The preprocessor must ignore any #pragma directive that contains an unrecognized command; it is not permitted to give an error message.

Functions with variable number of arguments

You may like to have a function that will accept any number of values and then return the average. You don't know how many arguments will be passed in to the function.

One way you could make the function would be to accept a pointer to an array.

Another way would be to write a function that can take any number of arguments.

So you could write avg(4, 12.2, 23.3, 33.3, 12.1); or you could write avg(2, 2.3, 34.4); The advantage of this approach is that it's much easier to change the code if you want to change the number of arguments.

Whenever a function is declared to have an indeterminate number of arguments, in place of the last argument you should place an ellipsis (which looks like '...'), so, int a_function (int x, ...); would tell the compiler the function should accept however many arguments that the programmer uses, as long as it is equal to at least one, the one being the first, x.

We'll need to use some macros from the stdarg.h header file to extract the values stored in the variable argument list -- **va_start**, which initializes the list, **va_arg**, which returns the next argument in the list, and **va_end**, which cleans up the variable argument list.

#include <stdarg.h>
void va_start(va_list *ap*, *last*);
type **va_arg(va_list** *ap*, *type*);
void va_end(va_list *ap*);

Description
A function may be called with a varying number of arguments of varying types.

The include file <*stdarg.h*> declares a type *va_list* and defines three macros for stepping through a list of arguments whose number and types are not known to the called function.

The called function must declare an object of type *va_list* which is used by the macros **va_start**(), **va_arg**(), and **va_end**().

va_start()

The **va_start**() macro initializes *ap* for subsequent use by **va_arg**() and **va_end**(), and **must be called first**.

The argument *last* is the name of the last argument before the variable argument list, that is, the last argument of which the calling function knows the type.

Because the address of this argument may be used in the **va_start**() macro**, it should not be declared as a register variable, or as a function or an array type**.

va_arg()

The **va_arg**() macro expands to an expression that has the type and value of the next argument in the call. The argument *ap* is the *va_list ap* initialized by **va_start**(). Each call to **va_arg**() modifies *ap* so that the next call returns the next argument. The argument *type* is a type name specified so that the type of a pointer to an object that has the specified type can be obtained simply by adding a * to *type*.The first use of the **va_arg**() macro after that of the **va_start**() macro returns the argument after *last*. Successive invocations return the values of the remaining arguments.

If there is no next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), random errors will occur.

If *ap* is passed to a function that uses **va_arg(***ap***,***type***)** then the value of *ap* is undefined after the return of that function.

va_end()

Each invocation of **va_start**() must be matched by a corresponding invocation of **va_end**() **in the same function**.

After the call **va_end(***ap***)** the variable *ap* is undefined. Multiple traversals of the list, each bracketed by **va_start**() and **va_end**() are possible. **va_end**() may be a macro or a function.

To use these functions, we need a variable capable of storing a variable-length argument list--this variable will be of type va_list. va_list is like any other type. For example, the following code declares a list that can be used to store a variable number of arguments.

```
va_list a_list;
```

va_start is a macro which accepts two arguments, a va_list and the name of the variable that directly precedes the ellipsis ("...").

So in the function a_function, to initialize a_list with va_start, you would write va_start (a_list, x);

```
int a_function ( int x, ... )
{
    va_list a_list;
    va_start( a_list, x );
}
```

va_arg takes a va_list and a variable type, and returns the next argument in the list in the form of whatever variable type it is told. It then moves down the list to the next argument.

For example, va_arg (a_list, double) will return the next argument, assuming it exists, in the form of a double. The next time it is called, it will return the argument following the last returned number, if one exists.

Note that you need to know the type of each argument--that's part of why printf requires a format string! Once you're done, use va_end to clean up the list: va_end(a_list);

Program1
Program2