0 INTRODUCTION AND BASICS

Open sesame!

- The History of Ali Baba

0.0 C - An Overview

C is one of the widely used languages. It is a very powerful language suitable for system programming tasks like writing operating systems and compilers. For example, the operating systems UNIX and OS/2 are written in C and when speaking about compilers its easy to list out the compilers that are *not* written in C! Although it was originally designed as systems programming language, it is used in wide range of applications. It is used in the embedded devices with just 64-KB of memory and is also used in super computers and parallel computers that run at un-imaginable speeds. C and its successor C++ cover most of the programming areas and are predominant languages in the world of programming.

To put in the words of the creator of C++ Bjarne Stroustrup [Stroustrup 1986],

"C is clearly not the cleanest language ever designed nor the easiest to use, so why do many people use it?

It is flexible [to apply to any programming area]…

It is efficient [due to low-level semantics of the language]…

It is available [due to availability of C compilers in essentially every platform]…

It is portable [can be executed on multiple platforms, even though the language has many non-portable features]…".

C is a language for programmers and scientists and not for beginners and learners. So it's naturally the language of choice for them most of the times.

C is not a perfectly designed language. For example few of the operator precedence are wrong. But the effect is irreversible and the same operator precedence continues to be even in newer C based languages.

C concentrates on convenience, writability[†], workability and efficiency to safety and readability. This is the secret of its widespread success. Lets see a classic example for such code:

```
void strcpy(char *t, char *s)
{
        while(*t++ = *s++) ;
}
```

This code has less readability. It is curt and to the point. It is efficient (compared to the 'obvious' implementation). It gives power to the programmer. It is not verbose…

C is thus a language for the programmers by the programmers and that is the basic reason why it is so successful.

C is different from other programming languages by its design objectives itself and this fact is reflected in its standardization process also. Some of the facets of the spirit of C can be summarized in phrases like `[Rationale ANSI C 1999]`,

- Trust the programmer.
- Don't prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Make it fast, even if it is not guaranteed to be portable.

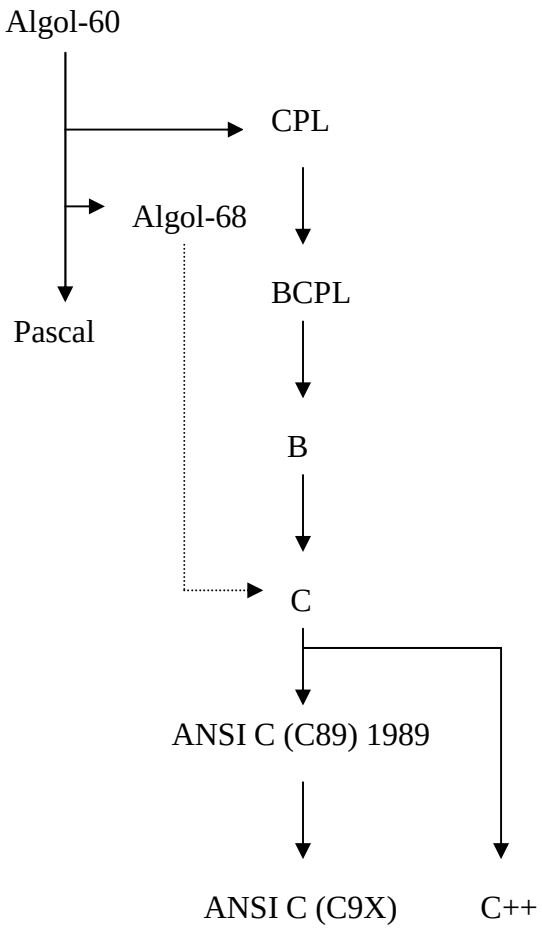
Understanding this design philosophy may help you understand some puzzling details of why C is like this in its present form.

Point to Ponder:

C is an attitude!

0.1 Brief history of C language

[†] There is no such jargon as 'writability' and here I refer it to as the ability to write programs lucidly



The evolution of C language

C language is the member of ALGOL-60 based languages. As I have already said, C is neither a language that is designed from scratch nor had perfect design and contained many flaws.

CPL (Combined programming language) was a language designed but never implemented. Later BCPL (Basic CPL) came as the implementation language for CPL by Martin Richards. It was refined to language named as B by Ken Thompson in 1970 for

the DEC PDP-7. It was written for implementing UNIX system. Later Dennis M. Ritche added types to the language and made changes to B to create the language what we have as C language.

C derives a lot from both BCPL and B languages and was for use with UNIX on DEC PDP-11 computers. The array and pointer constructs come from these two languages. Nearly all of the operators in B is supported in C. Both BCPL and B were type-less languages. The major modification in C was addition of types. [Ritchie 1978] says that the major advance of C over the languages B and BCPL was its typing structure. "The type-less nature of B and BCPL had seemed to promise a great simplification in the implementation, understanding and use of these languages… (but) it seemed inappropriate, for purely technological reasons, to the available hardware". It derives some ideas from Algol-68 also.

0.2 ANSI C Standard

Although K& R C had a rich set of features it was the initial version and C had a lot to grow. The [Kernighan and Ritchie 1978] was the reference manual for both the programmers and compiler writers for almost a decade. Since it is not meant for compiler writers, it left lot of ambiguity in its interpretation and many of the constructs were not clear. One such example is the list of library functions. Nothing significant is said about the header files in the [Kernighan and Ritchie 1978] and so each implementation had their own set of library functions. The compiler vendors had different interpretations and added more features (language extensions) of their own. This created many

inconsistencies between the programs written for various compilers and lot of portability and efficiency problems cropped up.

To overcome the problem of inconsistency and standardize the available language features ANSI formed a committee called X3J11. Its primary aim was to make “an unambiguous and machine-independent definition of C” while still retaining the spirit of C. The committee made a research and submitted a document and that was the birth of ANSI C standard. Soon the ISO committee adopted the same standard with very little modifications and so it became an international standard. It came to be called as ANSI/ISO C standard or more popularly as just ANSI C standard.

Even experienced C programmers also doesn’t know much about ANSI standard except what they frequently read or hear about what the standard says. When they get curious enough to go through the ANSI C document, they stumble a little to understand the document. The document is hard to understand by the programmers because it is meant for compiler writers and vendors ensures accuracy and describes the C language precisely. So the language used in the document is jocularly called as ‘standardese’. For example to describe side effects, the standard uses the idea of ‘sequence-points’ that may help confusing the reader more. L-value is not simply the ‘LHS (to =) value’. It is more properly a "locator value" designating an object.

ANSI standard is not a panacea for all problems. To give an example, ANSI C widened the difference between the C used as a ‘high-level language’ and as ‘portable assembly language’. The original `[Kernighan and Ritchie 1978]` is more preferred even now by the various language compilers to generate C as their target language. Because it is less-typed than ANSI C. To give another example, many think ‘sequence-

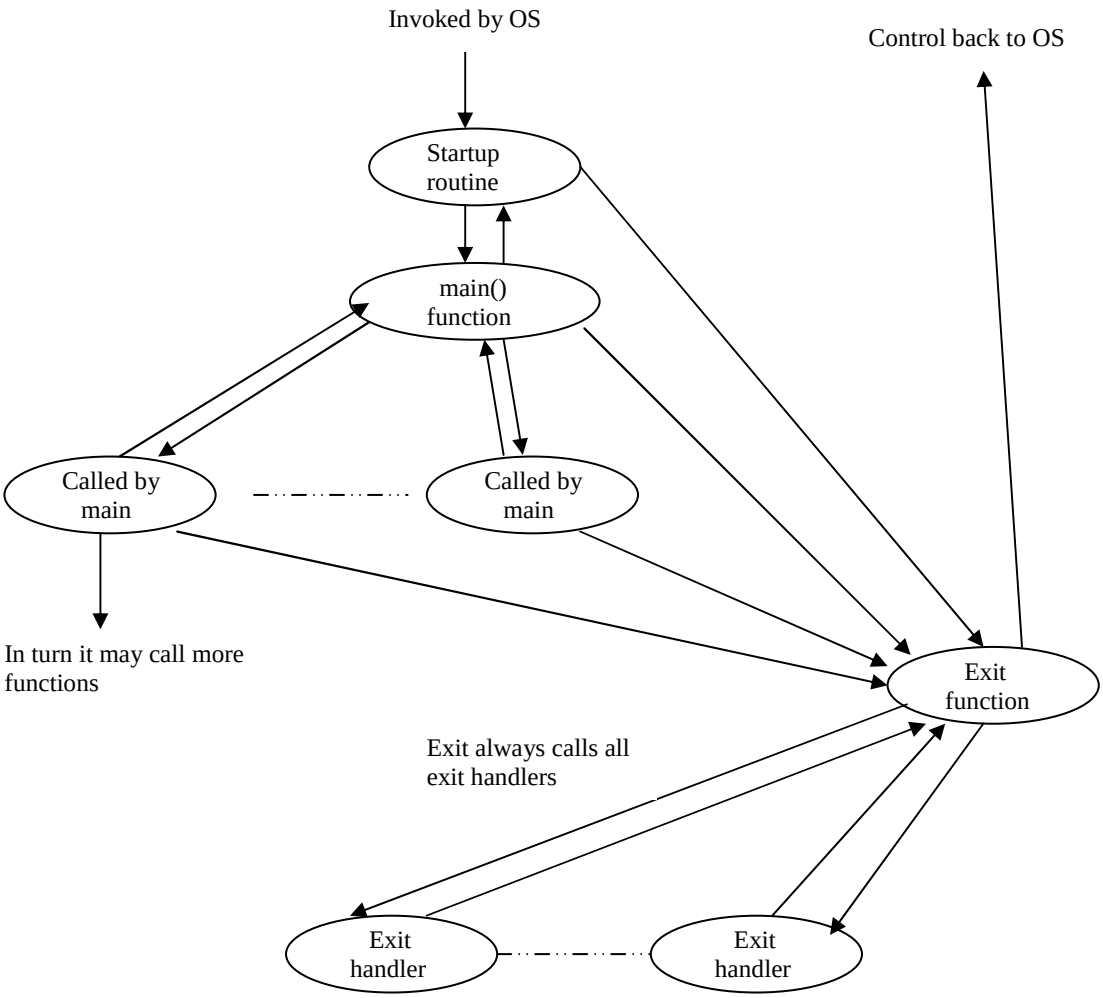
points' fully describe side-effects and the belief that knowing its mechanism will help to fully understand side-effects. This is a false notion about sequence-points of [ANSI C 1989]. Sequence points doesn't help fully understand side-effects.

0.3 The Future of C Language

Although the C may be a base for successful object oriented extensions like C++ and Java, C still continues to remain and be used with same qualities as ever. C is still a preferable language to write short efficient low-level code that interacts with the hardware and OS. The analogy may be the following one.

The old C programmers sometimes used assembly language for doing jobs that are tedious or not possible to do in C. In future, the programmers in other programming languages may do the same. They will write the code in their favorite language and for low-level routines and efficiency they will code in C using it as an assembly language.

0.4 The Lifetime of a C Program


Invoked by OS
Control back to OS
Startup routine
main() function
Called by main
Called by main
In turn it may call more functions
Exit function
Exit always calls all exit handlers
Exit handler
Exit handler

The life of a C program starts by being called by the OS. The space is allocated for it and the necessary data initializations are made. The start-up routine after doing the initialization work always calls the main function with the command line parameters passed as the arguments to it. The main function may in-turn call any function calls available in the code and the calling of functions continues if any such calls are there.

If nothing abnormally happens the control finally returns to main(). main() returns to start-up routine. Start-up routine calls exit() to terminate the program with the return value from main. It is as if the start-up routine has,

```
exit(main());   //or
exit(main(argc,argv));
```

The exit function calls all the exit handlers (i.e. the functions registered by atexit()). All files and stdout are flushed and the control returns back to OS.

If abort() is called by any of the functions, then the control directly returns to the OS. No other calls to other functions are made nor do the activities like flushing the files take place.

More information about this process and the functions involved are explained in the chapter on functions.

0.5 Source Files

Source files are of two types: interface source files and implementation source files. The interface source files are normally referred to as header files normally have .h extension and implementation files have .c extension.

The interface files contain the function prototypes, variable declarations, structure/union definitions etc.

The implementation source files contain the information like function definitions, other definitions and the information needed to generate the executable file, allocate and initialize data.

The standard header files are examples for the interface files and the code is available as .lib files and are linked at link-time by the linker to generate the .exe file. It should be noted that only the code for the functions used in the program gets into the .exe file even though many more functions are available in the header files.

0.6 Translation phases

To understand and resolve ambiguity with sequence in which the operations is done while translating the program, translation phases are available in ANSI C [ANSI C 1998]. The implementation may do this job in a single stretch, or combine the phases, but the effect is as if the programs are translated according to that sequence. For example, the implementation can have a preprocessor that does the work of all the phases intended for that in a single stretch.

1. multibyte characters are mapped to the source character set,
2. trigraph sequences are replaced by corresponding single-character internal representations,
3. backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines,

4. the source file is decomposed into preprocessing tokens and sequences of white-space characters (including comments),
5. preprocessing directives are executed, macro invocations are expanded, and _Pragma unary operator expressions are executed. All preprocessing directives are then deleted,
6. mapping from each source character set member and escape sequence in string literals is converted to the corresponding member of the execution character set,
7. adjacent string literal tokens are concatenated,
8. white-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token,
9. all external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation.

0.7 Start-up Module

In C, logically main is the function first called by the operating system in a program. But before main is executed OS calls another function called ‘start-up module’ to setup various environmental variables and other tasks that have to be done before calling main and giving control to it. This function is made invisible to the programmer.

Say you are writing code for an embedded system. In the execution environment, there is no OS to initialize data-structures used. In such cases, you may have to insert

your code in that ‘start-up module’. Compilers such as Turbo and Microsoft C provide facilities to add code in such cases for a particular target machine, for e.g. 8086.

0.8 main()

main is a special function and is logically the entry point for all programs. Returning a value from main() is equivalent to calling exit with the same value.

```
main()
{
int i;
static int j;
}
```

The variables i and j declared here have no difference because the scope, lifetime and visibility are all the same. In other words the local variables inside main() are created when the program starts execution and are destroyed only when the program is terminated. So it does not make much sense to declare any variable as static inside the main().

The other differences between main() and other ordinary functions are,

- the parameters with which main() can be declared are restricted,
- it is the only function that can be declared with either zero or two (or sometimes three) arguments. This is possible with main() function because it is declared implicitly, and is a special function. For other

functions, the number of arguments must match exactly between invocation and definition.

- parameters to main() are passed from command line,
- main() is the only function declared by the compiler and defined by the user,
- main() is by convention a unique external function,
- main() is the only function with implicit return 0; at the end of main(). When control crosses the ending '}' for main it returns control to the operating system by returning 0 to it (if there is no explicit return statement with a return value). The OS normally treats return 0 as the successful termination of the program.
- return type for main() is always is an int, (some compilers may accept void main() or any other return type, but they are actually treated as if it is declared as int main(). It makes the code non-portable. Always use int main()).

Standard C says that the arguments to main(), the argc and argv can be modified. The following program employs this idea to call the main() recursively.

```
// file name is recursion.c
// called from command line as,
// recursion 2
int main(int argc, char *argv[])
{
if(atoi(argv[1])>=0)
   {
   sprintf(argv[1],"%d", (atoi (argv[1]) - 1) );
```

```
    main(2,argv);
    }
}
// prints
// main is to be called 2 time(s) yet
// main is to be called 1 time(s) yet
// main is to be called 0 time(s) yet
```

0.9 Command line arguments

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

The name of the arguments is customary and you can use your own names. The first two arguments needed to be supported by the operating system. If numeric data is passed in command line, they are available as strings, so you must explicitly convert them back.

ANSI C assures that argv[argc]==0 is always true. So,

```
int main(int argc, char **argv, char **envp)
{
int i = 0;
while(i < argc)
     printf("%s\n",argv[i++]);
// and the following one are equivalent
while(*argv)
     printf("%s\n",*argv++);
}
```

The third argument char *envp is used widely to get the information about the environment and is nonstandard.

```
/* to show the environment */
int main(int argc, char **argv, char **envp)
{
while(*envp)
printf("%s\n",*envp++);
}
```

This program when executed in our machine it printed,

```
TEMP=C:\WINDOWS\TEMP
PROMPT=$p$g
winbootdir=C:\WINDOWS
COMSPEC=C:\WINDOWS\COMMAND.COM
PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;D:\SARAL\\BIN
windir=C:\WINDOWS
BLASTER=A220 I5 D1 T4
CMDLINE=noname00
```

Using the third argument in main is not strictly conforming to standard.

There is another widely used non-standard way of accessing the environmental variables and that is through the ‘environ’ external variable.

```
int i=0;
extern char ** environ;
while(environ[i])
```

```
printf("\n%s",environ[i++]);
```

The recommended way is to use the solution provided by ANSI as getenv() function for maximum portability:.

```
int main()
{
char * env = getenv(“PROMPT”));
// getenv is declared in stdlib.h
if(env)
     puts(env);
else
     puts(“The environmental variable not available”);
}
```

This program when executed in our machine it printed,

```
$p$g
```

Exercise 0.1:

argv[0] contains the name used to invoke the program. Is there any circumstance that it possible that it will contain null string “”?

0.10 Program Termination

The termination of the program may happen in one of the following ways,

Normal termination,

- by calling return explicitly from the main(),

- by reaching the end of main() (returns with implicit value 0),
- by calling exit(),

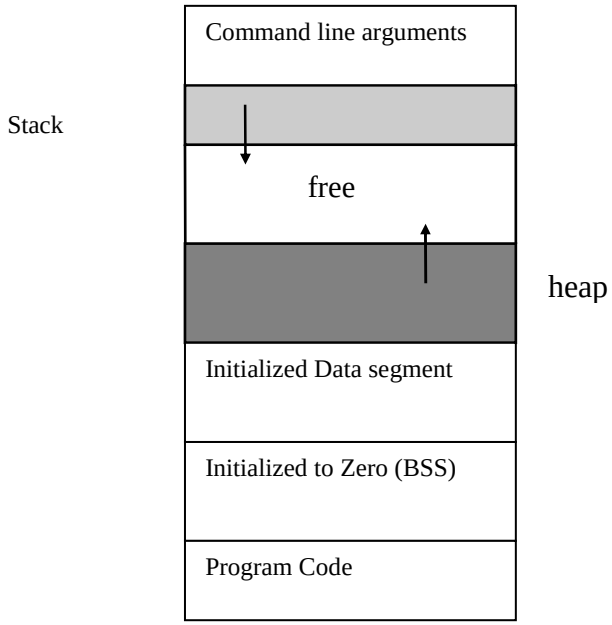
```
// yes, calling exit is a way for normal program termination
```

Abnormal termination,

- by calling abort(),
- by the occurrence of exception condition at runtime,
- by raising signals.

0.11 Structure of a C Program in Memory

The general way in which C programs are loaded into the memory is in the following format,



Structure of a C Program in Memory

0.12 Structure of a C Program in Memory

Major parts are,

- Data segment,
  - Initialized data segment(initialized to explicit initializers by programmers),
  - Uninitialized data segment (Initialized to zero data segment – BSS)
- Code segment,
- Stack and heap areas.

0.12.1 Data segment

The data segment contains the global and static data that are explicitly initialized by the users containing the initialized values.

The other part of data segment is called as BSS segment (standing for - Block Starting with Symbol – because of the old IBM systems had that segment initialized to zero) is the part of the memory where the operating system initializes it to Zeroes. That is how the uninitialized global data and static data get default value as zero. This area is fixed has static size (i.e. the size cannot be increased dynamically).

The data area is separated into two areas based on explicit initialization because the variables that are to be initialized can initialized one by one. However, the variables that are not initialized need not be explicitly initialized with zeros one by one. Instead of

that, the job of initializing the variables to zero is left to the operating system to be taken care of. This bulk initialization can greatly reduce the time required to load.

Mostly the layout of the data segment is in the control of the underlying operating system, still some loaders give partial control to the users. This information may be useful in applications such as embedded systems.

This area can be addressed and accessed using pointers from the code. Automatic variables have overhead in initializing the variables each time they are required and code is required to do that initialization. However, variables in data area does not have such runtime overhead because the initialization is done only once and that too at loading time.

0.12.2 Code segment

The program code is the code area where the executable code is available for execution. This area is also of fixed size. This can be accessed only be function pointers and not by other data pointers. Another important information to note here is that the system may consider this area as read only memory area and any attempt to write in this area leads to undefined behavior.

Constant strings may be placed either in code or data area and that depends on the implementation.

The attempt to write to code area leads to undefined behavior. For example the following code may result in runtime error or even crash the system (surprisingly, it worked well in my system!).

```
int main()
{
```

```
 static int i;
 strcpy((char *)main,"something");
 printf("%s",main);
 if(i++==0)
     main();
}
```

0.12.3 Stack and heap areas

For execution, the program uses two major parts, the stack and heap. Stack frames are created in stack for functions and heap for dynamic memory allocation. The stack and heap are uninitialized areas. Therefore, whatever happens to be there in the memory becomes the initial (garbage) value for the objects created in that space. These areas are discussed in detail in the chapter on functions.

Lets look at a sample program to show which variables get stored where,

```
int initToZero1;
static float initToZero2;
FILE * initToZero3;
// all are stored in initialized to zero segment(BSS)

double intitialized1 = 20.0;
// stored in initialized data segment

int main()
```

```
{
size_t (*fp)(const char *) = strlen;
// fp is an auto variable that is allocated in stack
// but it points to code area where code of strlen() is stored

char *dynamic = (char *)malloc(100);
// dynamic memory allocation, done in heap

int stringLength;
// this is an auto variable that is allocated in stack

static int initToZero4;
// stored in BSS

static int initialized2 = 10;
// stored in initialized data segment

strcpy(dynamic,"something");
// function call, uses stack

stringLength = fp(dynamic);
// again a function call
}
```

Or consider a still more complex example,

```
int main(int numOfArgs, char *arguments[])
```

```
{ // command line arguments may be stored in a separate area
static int i;
// stored in BSS
int (*fp)(int,char **) = main;
// points to code segment
static char *str[] = {"thisFileName","arg1",
     "arg2",0};
// stored in initialized data segment
while(*arguments)
     printf("\n %s",*arguments++);
if(!i++)
     fp(3,str);
}
// in my system it printed,
// temp.exe
// thisFileName
// arg1
// arg2
```

After seeing how a C program is organized in the memory, to cross check the validity of the idea you may try code like this,

```
void crossCheck()
{
int allocInStack;
// all auto variables are allocated in stack
void *ptrToHeap;
```

```
    ptrToHeap = malloc(8);
    // 8 bytes allocated in heap, pointed by a variable in stack
    if(ptrToHeap){
    assert(allocInHeap < &allocInStack);
    printf("Address of allocInStack %p and Address of heap
memory allocated %p\n", &allocInStack, ptrToHeap);
    crossCheck();
    }
    else
        printf("Memory exhausted of continous usage");
    }
    int main(){
        crossCheck();
    }
```

However, this program code suffers two major drawbacks,

- Comparison of two unrelated pointers (inside assert).

  ANSI says that the pointer comparison is valid only when the comparison is limited only to the limits of the array.

- Assuming some implementation dependent details.

  It is only a general case that stack and heap grow towards each other and stack is in higher memory locations than the heap. C does not assure anything as such.

This program is not portable. These kinds of problems are discussed throughout the book and you will be familiar with such ideas when you finish reading this book.

Exercise 0.2:

Consider the statement:

```
static int i = 0;
```

Where will be the variable i allocated space? Is it in BSS or initialized data segment?

Exercise 0.3:

The diagram doesn't show where the variables of storage class 'extern' and 'register' are stored. Could you tell where would they be stored?

0.13 Errors

Errors can occur anywhere in the compilation process. The possible errors are,

- preprocessor errors,
- compile time errors,
- linker errors.

Apart from these, runtime errors can also occur. If prevention is not taken for such run-time errors, it will terminate the program execution and so avoiding/handling them should be given utmost importance.

In C, if exceptions occur error flags kept by the system indicate them. A program may check for exceptions using these flags and perform corresponding patch up work. The program can also throw an exception explicitly using signals that are discussed under discussion on <signal.h>. A different method of error indication is available through errno defined in <errno.h>. More discussion about these header files is in later chapters.

Run-time errors are different from exceptions. Errors indicate the fatality of the problem and not meant to be handled.

Exercise 0.4:

The following code makes flags a “Divide by zero” error. Is it a compile or runtime error?

```
int i = 1/0;
```

1 PROGRAM DESIGN

High thoughts must have high language

- Aristophanes

Clear, efficient and portable programs require careful design. Design of programs involves so many aspects including the programmer's experience and intuition. Thus it is an art rather than a science. This chapter explores various issues involved in program design.

1.1 Portability

Portability is an important issue in the program design and the ANSI committee has dedicated an appendix to portability issues. ISO defines portability as "A set of attributes that bear on the ability of the software to be transferred from one environment to another"[Henricson and Nyquist 1997].

Therefore, a portable program should produce same output across various environments that differ in:

- Operating Systems
- Hardware
- Compiler
- user's natural language
- presentation formats(date, time formats etc)

Although C was originally developed for only one platform, the PDP 11, it has been successfully implemented on almost all platforms available. However C still has some non-portable features. In other words, C has the reputation of a being a highly portable language, but it has some inherently non-portable features. In fact, special care should be taken for programs that are to be ported, and details about behavioral types, discussed below, must be known.

1.1.1 Behavioral Types

The way the program acts at runtime determined by the behavioral type. The various behavioral types are,

- well-defined behavior,
- implementation-defined behavior,
- unspecified behavior,
- undefined behavior,

Behavioral types are not to be confused with errors. Illegal code causes errors/exceptions to occur at either compile-time or run-time. But the above behavioral types occur in legal code and are defined only for the actions of the code at runtime.

You can write code without knowing anything about the behavioral types. But knowledge about this is very crucial if you want to make your code be portable and of high quality. The problems that arise out of portability are very hard to find and correct.

1.1.1.1 Well-defined behavior

When the language specification clearly specifies how the code behaves irrespective of the platforms or implementations, it is known as well-defined behavior. It is the most portable code and has no difference in its output across various platforms.

The [Kernighan and Ritchie 1988] and ANSI Standard documents are the closest documents available to a 'C language specification'. If the behavior of the construct/code is described in these documents then the construct/code is said to be of well-defined behavior.

Most of the code we write is of well-defined behavior. To give an 'obvious' example, the standard library function malloc(size) returns the starting address of the allocated memory if 'size' bytes are available in the heap, else it returns a NULL pointer. Both [Kernighan and Ritchie 1988] and ANSI describe how malloc behaves when sufficient memory is available and not available, so is a well-defined behavior. To see a 'non-obvious' example:

```
unsigned int i = UINT_MAX;
i++;
if(i==0)
     printf("This is a well defined behavior");
// now i rotates and so becomes 0
// prints
// This is a well defined behavior
```

The code behaves the same way irrespective of the implementation and the same output is printed.

1.1.1.2 Implementation defined behavior

When the behavior of the code is defined and documented by the implementers or compiler writers, the code is said to have implementation defined behavior. Therefore, the same code may produce different output on different compilers even if they reside on a same machine and on a same platform.

The best example for this could be the size of the data types. The documentation of the compiler would specify the size of the data types.

Since it is almost impossible to write code without implementation defined code. For our example if you declare,

```
int i;
// this has implementation-defined behavior - sizeof (int) = ?
```

then your program has such behavior. A programmer is free to use such code, but he should never rely on such behavior. For example:

```
char ch = -1;
```

This is implementation-defined behavior. The language specification leaves the decision of whether a char should be signed or unsigned to the implementor. So the above code is not recommended.

The list of the implementation-defined behaviors given by ANSI is given in appendix.

1.1.1.3 Unspecified behavior

The designers of the language have understood that it is natural that the implementations vary for various constructs depending on the platform. This makes the implementation efficient and fit for that particular platform. Some of these details are too implementation specific that the programmer need not understand that. These are need not be documented by the implementation. The behavior of such code is known as unspecified behavior. One such example is the sequence in which the arguments are evaluated in a function call.

```
someFun( i += a , i + 2);
callTwoFuns( g(), f() );
```

The arguments of a function call can be evaluated in any order. The expression `i +=a` may be evaluated before `i + 2` and vice-versa.

You should not write code that relies upon such behavior.

Implementation defined behavior and unspecified behavior are similar. Both specifies that the behavior that is implementation specific. The main difference is that the implementation-defined behavior is to be documented by the vendor and are features that the user generally accesses directly. Whereas in unspecified behavior the compiler vendor may not document it and are implementation details that are generally not accessed by the users.

The standard committee did not define the constructs of these two behavioral types intentionally to have full access to underlying hardware and efficient implementation.

1.1.1.4 Undefined behavior

If neither the language specification nor the implementation specifies the behavior of an erroneous code, then the code is said to be have undefined behavior. The behavior of the code in the environment cannot the said precisely.

So the code that contains such behavior should not be used and is incorrect because of erroneous code or data. Undefined behavior may lead to any effect from giving erroneous results to system crash.

```
int i=0, j=1;
(&i+1) = 10;          // assign the value 10 to j
```

Here the variable j is assigned with exploiting the fact that in that environment the variables i and j are stored in adjacent locations.

```
int *i;
*i = 10;
```

i is a wild-pointer and the result and behavior of the code of applying indirection operator on it undefined.

These are examples of using undefined behavior. Code with undefined behavior is always undesirable and should be strictly avoided. In such cases, either use assert to make sure that you don’t use that accidentally or remove such behavior from the code.

1.1.2 Language extensions

The compiler vendors make language extensions for various reasons,

- to extend the language itself as adding extra features to the language (this happens naturally as the language evolves and normally before the standardization takes place),
- sometimes to make it possible for code to be generated for a particular platform,
- to make the code generated for a particular platform to be more efficient. (E.g. near, far and huge pointer types in Microsoft and Borland compilers for x86 platform).

Let's see an instance for a requirement of language extension and how that request is satisfied.

In writing programs like device drivers and graphical libraries the speed is crucial. Access to the hardware registers and other system resources may be required sometimes. There are instances where manipulation of registers and execute instructions that are inaccessible through C but are accessible through assembly language (C has low-level features but not this much low level at the cost of portability). In C the assignment of one array/string to another is not supported. But the assembly language for that hardware may have instructions that may do these operations atomically (block copy) which will require C code to do element-by-element copy. Providing standard library functions, which may be implemented in C or in assembly language, recognises the need for such access to the special cases. Examples for such library functions are getchar(), memcpy() etc.

Thus there is a need that the assembly code be directly written in C. This will help the programmer to code in assembly language in C programs wherever greater efficiency is required/ low-level interaction is needed.

This feature is available in many implementations as asm statement.

```
asm(assembly_instruction);
```

will insert the assembly_instruction be directly injected into the assembly code generated.

Let's say we have to install a new I/O device. How the interfacing to that device be made? This can be done using C code now and using assembly code wherever it is required.

This feature is also useful for time-critical applications where an overhead of even a function call may be high.

Using assembly code for efficiency has many disadvantages. The programmers who update the code may not be familiar with the particular assembly language used. Moreover porting the code to other systems requires the code be rewritten in that particular assembly language. This feature (and as in the case of all language extensions) compromises portability for efficiency.

Avoid using language extensions unless you are writing code only for a particular environment and the efficiency is of top priority. Stay within the mainstream and well-defined constructs of the language to avoid portability problems.

1.1.3 Steps for Writing Portable Code

Writing portable code is not done automatically and it is only by conscious effort as far as C is concerned. The following steps are recommended when writing any serious C code:

1. Analyze the portability objectives of any program before writing any C code.

2. Write code that conforms to the standard C. This should be done even if your compiler or platform has lot of extra features to use (like language extensions). Using such features when writing standard C code possibly will harm the portability of the code. Use standard C library whenever possible. Avoid using third party libraries when achieving the same functionality through the standard library is possible.
3. When the support for the functionality is not available in the standard library look for the functionality in the library provided by your compiler vendor. See if that functionality is available in the source code form.
4. When the functionality you want is not available even in the library provided by your compiler vendor, look for any such library in the market preferably in the source code form.
5. Only after failing to have such functionality in the third-party libraries, decide to develop your own code, that too keeping portably in mind. Try to do it in C code and only if not possible go to the options like using assembly code for your programs.

Lets look at an example of how this can be applied systematically for a problem-at-hand. XYZ company wants a tool for storing, retrieving and displaying the photographs of their employees in a database form. The company already has acquired a special hardware for scanning the photographs. It is already using software developed in C for office automation and they have the source code for the same.

For the problem C suits well because they are already have the application running in C and source code is also available and the tool for scanning and storing the photographs can be done in C very well.

On the first hand examine the scope of the problem. This is a requirement that may be required in many companies and so it has lot of scope for being used outside the company. The places where it may be required may have to interface with different hardware (like scanners) and may require running on different platforms. Therefore, the gains due to portability seem to be attractive, even if portable code is not possible, the non-portable code will serve the purpose at hand.

As the next step you see if the code can be written completely in standard C. The platform you work is UNIX and so for storing the data, low-level files can be used. Doing so will harm portability, so use standard library functions for doing that. For this problem, interfacing with the hardware is required and for displaying the photos graphics support is needed. Even though writing complete code in standard C is not possible, most of the code can still be written in standard C. Make sure to keep the non-portable code easy to find and isolate it to separate files.

For interfacing with external hardware devices your compiler provides special header files and the source code is also available for you. The scanner is accompanied with software for interfacing it with your code. You observe that the same functionality is achievable by using the library provided by your vendor, without using the interfacing software from the scanner. Hence, you resort to using the library since this can work for any other scanners also although you need to write some more code.

The standard C does not have any graphics library. Unfortunately, your compiler vendor also happens to not provide one such library. You have a good assembler, also you are an accomplished assembly language programmer, and your compiler has options to integrate the assembly code in your code. However, you observe that a portable graphics package available by a third-party software vendor. You have to spend a little for purchasing that and that graphics package does not perform as good as your assembly code. You end up by buying the graphics package because it has better portability options.

Thus you end up writing the code that is maximally portable without using language extensions, platform dependent code or assembly code. In addition, you make lot of money selling the package to other companies with little or no modifications. So it is always preferable to write maximally portable code, if not fully portable code.

1.1.4 Writing non-portable code

Throughout the book I stress on the importance of portability and writing portable code. This *doesn't* mean that you should *never* write non-portable code. My point is that writing portable code helps you to have maximum benefit by distributing the code to various platforms. It also minimizes your effort to port to new-platforms.

Sometimes it is necessary for you to write non-portable code (for example a graphics package/library or hardware interface). In such cases:

- make non-portable code easy to identify and locate,
- use conditional compilation (to make it possible to have code depending on the platform supported).

- use typedefs (to hide/abstract such platform dependant details),
- isolate/group all the platform specific code to few files (if the code is to be ported to other platforms it is enough to change only the code in those files)

The ability to write non-portable and platform specific code is actually a one of the reasons for widespread success of C.

As the [ANSI-98] puts it as one of the underlying principles of standardization of C itself as "C code can be non-portable". Since C can be effectively used to write code for a particular platform, you can reap the maximum benefit from the available underlying platform. For example lets see an example of using system calls of UNIX for executing one program from within another.

The system calls used for low-level process creation are execlp() and execvp(). The execlp call overlays the existing program with the new one , runs that and exits. The original program gets back control only when an error occurs.

```
execlp(path, file_name,arguments...);
//last argument must be NULL
```

A variant of execlp called execvp is used when the number of arguments is not known in advance:

```
execvp(path, argument_array);
//argument array should be NULL terminated
```

System calls are further discussed under the chapter in "Unix and Windows programming in C".

1.2 Language Features to Avoid

Every language has its own strengths and weaknesses. They have strongholds, traps and pitfalls. So, language supports a feature doesn't mean that that feature should be used. This is true for even a small language like C with less features. For example, the language supports 'pragmas', but using that leads to non-portable code.

Sometimes you have to avoid using some language features, depending on the environment you program. For example while programming for embedded systems, normally, the use of dynamic memory allocation is prohibited.

C is a language where you can code in different ways to solve the same problem. So careful decision should be made in selecting the language features that are harmless, well understood and less error-prone. For example, take a simple task of finding the biggest of three numbers. Depending on the requirement and situation, you can either opt for macros or functions, but in general, it is better to avoid macros and go for functions (I discuss a situation where macros is preferable to functions in the chapter on "preprocessor").

So be cautious in selecting and using the features supported by the language.

1.3 Performance and Optimization Considerations

For serious scientific applications, performance is an important criterion and slight difference in speed can make a big difference. C was, of course, designed keeping efficiency in mind, but the problem is that it was based on PDP machines. One such example is the memory access techniques in C that are based on PDP Machines.

One cannot fully rely on the compiler to optimize and it is always good to hand-optimize the code as much as possible particularly in time-critical and scientific applications. Because the programmer knows his intentions clearly and can optimize better while writing the code to the compiler analyzing the code and make the code efficient.

The optimizations that are possible can vary with requirements. In some cases, the readability of the code needs to be slightly affected for optimizing the code. In addition, optimizing depends on the platform, the minute hardware details, and many implementation details and knowledge of such details is sometimes necessary to write a much-optimized code.

For example, infinite loop for(;;) generates faster code than the while(1) even though both intends to do the same. This is because for(;;) is a specialized condition for the ‘for’ loop that is allowed by C language to indicate infinite loop so the compiler can generate code directly for the infinite loop. Whereas for ‘while’ the code has to be generated such that the condition has to be checked and transferred after always checking the condition.

Some machines handle unsigned values faster than the signed values. Due to its desirable properties like they values never overflow, making explicit that the value can never go negative through the code itself etc., makes usage of unsigned over signed whenever possible. Copying bulk of data from one location to another location can be efficient if it is done in block multiples of eight bytes than byte by byte. Such an example of copying optimization is the Duffs device (discussed later).

Recursion is acceptable to map the problem directly to solution but can be costly

if the function has lot of auto variables occupying lot of space. In such cases avoid recursion and try the iterative equivalents.

1.3.1 Role of Optimizers

In the early days of C, it was used mostly for systems programming only. Initially the system programmers were reluctant to do programming in C to assembly language since it is widely believed that doing programming in high-level languages have the cost of efficiency. Soon the C compilers became available in multiple platforms and they were written such that they generated specialized code to fit the underlying machines. Importantly optimizers did a good job and became an important part in almost every C compiler. Optimizers can do some optimizations (like register optimizations) that are not always possible or tedious to do in doing assembly programming directly. Programmers can concentrate on other aspects of programming by leaving low-level programming to be taken care by the compiler.

Efficiency is not just a design goal but a driving force in C's design. So writing efficient code is natural in C (and most of us, the C programmers even do it sometimes unconsciously).

So the programmers started preferring C code to assembly language programming and that is an interesting transition standing as a testimony of C's commitment to efficient code. Efficiency is thus the combined quality of both the language and its implementation.

Although the optimizers do a good deal of work in improving the efficiency of the code, it is not good to write code that depends on optimization be done by it. Most of the

optimizations can be done by good programming practices, careful and good designing. There are numerous techniques to write optimal code and it is always better to write optimal and efficient code by us.

1.3.2 Size of the Executable File

The size of the executable code may be unnecessarily large due to many reasons. The primary reasons are,

- repetition/ duplication of the code,
- unnecessary functions that have been added

The reuse of the code is good in the sense it makes use of already available code that is normally a tested one. It reduces the development time also. However, it has a trade-off too. Large amount of code duplication takes place if code reuse is not done carefully. It makes the code harder to maintain (as opposed to the popular belief that reuse makes maintenance easier. Of course, this is true if care is taken while reusing code) because the original code is not tailored to solve the current need.

The tradeoff for the program size is the performance. If the file is too big, the whole program cannot reside in the memory. Therefore, frequent swapping of pages has to take place to give space for new pages[†]. The overall effect is the performance degradation.

[†] In case of paged memory management systems (like DOS); not in every operating system. My idea is to convey that making .exe files unnecessarily big affects performance.

1.3.3 Memory Management

Whenever possible prefer automatic storage as opposed to dynamic storage. This is because the code has to be written to take care of dynamic storage allocation failures and runtime overhead is involved in calling the memory allocation functions that may sometimes take more time. Managing the allocation and deallocation of memory explicitly by the programmer is error-prone and even experienced programmers stumble on this sometimes. Examples are the deallocation of memory twice and using the memory area that has already been deallocated. For these reasons, automatic storage must be preferred to dynamic storage whenever possible.

2 CONSTANTS, TYPES and TYPE CONVERSIONS

C provides you with different flavors of types[1] that can be tailored to suit any particular need. The language does not specify any limit on the range of the data types. So depending on the hardware, the compilers can implement them efficiently. This means that integer can be implemented with the native word size of the processor, which makes the operations faster. In addition, the library code or the math co-processor, depending on the availability, can do the floating-point operations.

In C the types may be broadly classified into scalar, aggregate, function and void. There are further sub-divisions, which can be understood from the diagram. Before knowing about constants and types lets see about variables.

2.1 Variables

Variables are names given to the memory locations, a way to identify and use the area to store and retrieve values. It is for the programmer, and so they do not exist after the executable code is created. Whereas the constants live up to the compilation process only and have no memory locations associated with them.

```
int i, *ip = &i;
// &i is allowed because i has a memory location
```

[1] I want to clarify the difference between 'type' and 'data type'. Data type specifies a set of values and a set of operations on those values. However, type is a super set of data type, obtained by using existing data types to come out with a composite set of values and a set of operations on those values (e.g. using typedefs). Hereafter I use 'type' synonymously with 'data type'.

```
// and so can take address of it.
int cp = &10;
// is not allowed because the '10' is not stored
// anywhere and so you cannot apply & to it.
```

That is the same reason why constants cannot be used in the case of passing to functions,

```
void intSwap(int *i, int *j)
{
int temp = *i;
*i = *j;
*j = temp;
}
```

for this function call like,

```
intSwap(&i, &j);
// is perfectly acceptable
intSwap(&10,&20);
// is illegal because integer constant doesn't
// reserve memory space
```

One obvious exception is the string constants that are stored in the memory. For example, you should have used the code like this using this fact,

```
int i = strcmp("string1","string2");
// pass the addresses of string1 and string2
// which are stored somewhere in the memory.
char *str = "this string is available in memory";
// address of the string constant is stored in str.
printf("%p","someString");
```

```
// prints the address of the string constant "someString"
```

In other words variables are addressable whereas literal constants are non-addressable and that is why you can apply unary & operator only to variables and not for constants.

2.2 Types of variables

Variables can be classified by the nature with which the value it stores changes.

2.2.1 Synchronous variables

The value of these variables can only be changed through program code (like assign statements, which changes the value stored in that variable). All the variables used in C programs are synchronous unless otherwise explicitly specified (by const or volatile qualifiers)

```
int syn, *synp;
// and any other variables without the qualifiers const or volatile
// are synchronous
```

2.2.2 Asynchronous variables

These variables represent the memory locations where the value in that location is modified by the system and is in the control of the system. For example, the storage location that contains the current time in that system that is updated by the timer in the system. To indicate that the variable as asynchronous use volatile qualifier.

```
volatile float asyn = 10.0;
// this indicates to the compiler that the variable asyn is not an
```

```
// ordinary variable and its value may be changed by external factors
```

2.2.3 Read-Only variables

These are initialized variables that can only be read but not modified. The const qualifier indicates the variable of this type.

```
const int rov = 10;
// means that the variable rov may be used for reading purposes only
// and not for writing into it.
```

More about const and volatile qualifiers is discussed later.

This classification of variables was not there in the original K&R C because there were no const or volatile qualifiers then. This is due to ANSI C, which introduced these two qualifiers (called as cv-qualifiers standing for const and volatile qualifiers).

2.3 Constants

Constants are naming of internal representation of the bit pattern of the objects[2]. It means that the internal representation may change, but the meaning of constant never does. In C, the words 'constant' and 'literal' are used interchangeably.

[2] 'object' is a region of memory that can hold a fixed or variable value or set of values. This use of word 'object' is different from the meaning used in object-oriented languages. Hereafter the word 'object' is used to mean the variable and its associated space.

2.3.1 Prefixes and suffixes

Prefixes and suffixes force the type of the constants. The most common prefixes are ‘0x’ and ‘0’, used in hexadecimal and octal integers, respectively. Prefix ‘L’ is used to specify that a character constant is from a runtime wide character set, which is available in some implementations.

The suffixes used in integers are L/l, U/u (order immaterial). L denotes long and U for unsigned. In addition to the suffix L/l, the floating constants can have F/f suffix. If no suffixes are there, the floating-point constant is stored as double, the F/f forces it to be a float and L/l forces it to be long double.

Point to Ponder:

In the absence of any overriding suffixes, the data type of an integer constant is derived from its value

2.3.2 Escape characters

Escape characters are the combination of the \ and a character from the set of characters given below or an integer equivalent of the character, which has a special meaning in C. They are of two types:

2.3.2.1 Character escape code

If we use a character to specify the code then it is called a character escape code. They are

```
\a, \b, \f, \n, \r, \t, \v, \?, \\, \’, \”
```

2.3.2.2 Numeric escape code

If we specify the escape character with the \integer form, then it is called numeric escape code.

Exercise 2.1:

Escape characters (in particular, numeric codes) allow the mapping supported by the target computer. Justify.

2.4 Scalar Type

If all the values of a data type lie along a linear scale, then the data type is said to be of scalar data type. I.e. the values of the data type can be used as an operand to the relational operators.

2.4.1 Arithmetic Type

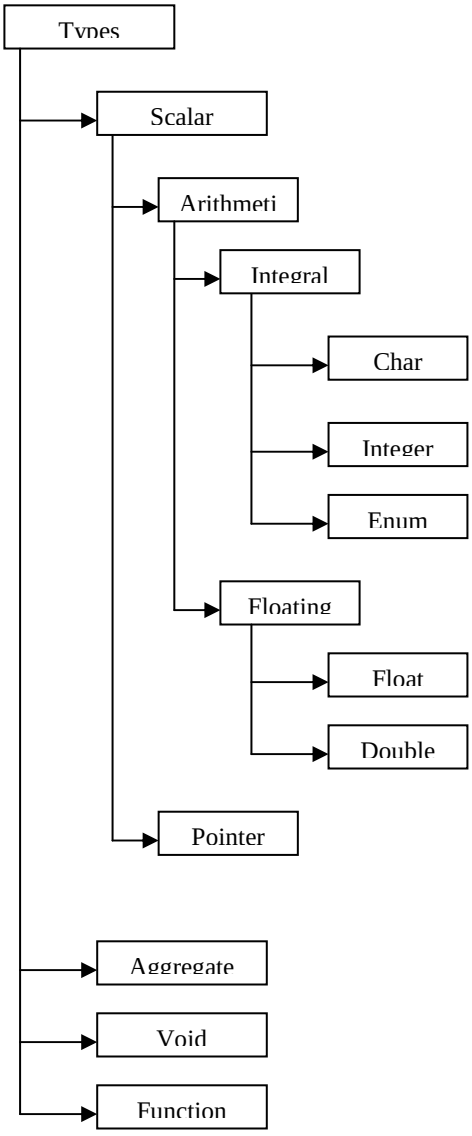
These are the types, which can be interpreted as numbers.

2.4.1.1 Integral Type

These are the types, which are basically integers.

2.4.1.2 Character Type

Character type is derived from integer and is capable of storing the execution character set. The size should be at least one byte. If a character from the execution character set is stored, the equivalent non-negative integer code is stored.



Various Datatypes available in C

We should not assume anything about the underlying hardware support for characters.

Version 1:

`ch >= 65 && ch <=91` is intended to check if the character is an upper case alphabet. It is not portable because the hardware may support some other character set (say EBCDIC) and so becomes wrong.

Version 2:

`ch >='A' && ch <='Z'` Assuming that the sequence A-Z is continuous. This may not be the case in every character set, so may fail.

Version 3:

`isupper(ch)`. This gives the required result, as it is sure to be portable.

If you want to print the ASCII character set (supposing your system supports it), you may write a code segment like this,

```
char ch;
for(ch=0;ch<=127;ch++)
     printf("%c   %d \n", ch, ch);
```

to your surprise this code segment may not work! The simple reason for this is that the char may be signed or unsigned by default. If it is signed then ch++ is executed after ch reaches 127 and rotates back to -128. Thus ch is always smaller than 127.

Exercise 2.2:

Can we use char as ‘tiny’ integer? Justify your answer. If yes, does the fact that the char may be signed or unsigned will affect your answer?

2.4.1.2.1 Character constants

The constants represented inside the single quotes are referred to as character constants. In ANSI C, a character constant is of type integer.

ANSI C allows multi-byte constants. Since the support from the implementations may vary, the use of multi-byte constants makes the program non-portable (multi-byte characters are different from wide characters).

```
int ch = ‘xy’;
// say, here sizeof(int) == 2 bytes.
// This is a multibyte-char
```

Prefix L signifies that the following is a multi-byte character where long type is used to store the information of more than one byte available.

```
wchar_t ch = L‘xy’;
// this is a wide character taking 2 bytes.
```

Exercise 2.3 **:**

Both of the following are equivalent:

```
char name1[] = “name”;
char name2[] = {‘n’,’a’,’m’,’e’,’\0’};
```

But you know that it takes two bytes for a character constant. Then why doesn’t name2 take more space because it is made up of character constants?

2.4.1.2.2 Multi-byte and Wide characters

ANSI C provides a way to represent the character set in various languages by a mechanism called multi-byte characters. When used, the runtime environment interprets contiguous bytes as a character. The number of bytes interpreted, as a single character, is implementation defined.

```
long ch = 'abcd';
// where long holds four characters and treats as a single multi-byte
// character.
```

Wide character may occupy 16 bits or more and are represented as integers and may be defined as follows,

```
typedef unsigned short wchar_t;
```

To initialize a character of type wchar_t, just do it as usual as for a char,

```
wchar_t ch = 'C';    // or
wchar_t ch = L'C'    // prefix L is optional.
```

Prefix L indicates that the character is of type wide-character and two bytes are allocated for that character.

For the wide-character strings, similar format is to be followed. Here the prefix L is mandatory.

```
wchar_t * wideStr = L"a wide string";    // or
wchar_t  wideStr[] = L"a wide string";
```

the same idea applies to array of strings etc.

The wide-character strings are null terminated by two bytes. As you can see, you cannot apply the same string functions for ordinary chars to strings of wide-chars.

```
strlen(wideStr);     // will give wrong results
```

For this, ANSI provides equivalent wide character string library functions to plain chars.

For e.g.

```
wcslen(wideStr)
// for finding the length of the wide character string
```

this is equivalent to strlen() for plain chars and wprintf for printf etc.

You can look it this way. Plain chars take 1-byte and wide-characters normally 2-bytes. Both co-exist with out problems (as int and long co-exist) and both have similar library functions of their own.

Multi-byte characters are different from wide characters. Multi-byte characters are made-up of multiple single byte characters and are interpreted as a single character at *runtime* in an implementation defined way. Whereas in wide character is a type (wchar_t) and is internally represented as an integer.

Library functions support is available for the wide characters but not for the multi-byte characters. For wide-characters, it is in an implementation-defined library and not much support is available for wide character manipulation for its full-fledged use. Portability problems will arise by the byte order or by the encoding scheme supported (say for Unicode UTF). If you want your software to be international, you may need this facility, but unfortunately, the facilities provided by the wide characters is not adequate.

The run-time library routines for translating between multibyte and wide characters include mbstowcs, mbtowc, wcstombs, and wctomb. For example:

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

this function converts the wide-character string to the multi-byte character string (it returns the number of characters success-fully converted).

```
char mbbuf[100];
wchar_t *wcstring = L"Some wide string";
wcstombs ( mbbuf, wcstring, 10 );
```

Similarly,

```
int wctomb(char *s, wchar_t wc);
```

This function tells number of bytes required to represent the wide-character ‘wc’ where ‘s’ is the multi-byte character string.

2.4.1.2.3 C and Unicode

ASCII is only for English taking seven bits to represent each character. The other European languages use extended ASCII that takes 8-bits to represent the characters that too with lot of problems. The languages such as Japanese, Chinese etc. used a coding scheme called as Double Byte Coding Scheme (DBCS). Because the character set for such languages are quite large, complex, and 8-bits are not sufficient to represent such character sets. For multilingual computing lot of coding schemes proliferated that lead to lots of inconsistencies. To have a universal coding scheme for all the world languages (character sets) Unicode was introduced. Unicode takes 16-bits to uniquely represent each character.

ANSI C inherently supports Unicode in the form of wide characters. Even though wide-characters are not meant for Unicode they match with the representation of Unicode.

We already saw about multi-byte characters that are composed of sequence of single bytes. The preceding bytes can modify the meaning of successive bytes and so are not uniform. They are strictly compiler dependent. Comparatively wide-characters are uniform and are thus suitable to represent Unicode characters. As I have said, facilities available for use of wide-characters for Unicode not adequate but is that is the solution offered by ANSI C.

2.4.1.2.4 Execution Character Set

The execution character set is not necessarily the same as the source character set used for writing C programs. The execution character set includes all characters in the source character set as well as the null character, new-line character, backspace, horizontal tab, vertical tab, carriage return, and escape sequences. The source and execution character sets may differ and in implementations.

2.4.1.2.5 Trigraphs

Not all characters used in the C source code, like the character '{', are available in all other character sets. The important character set that does not have these characters to represent is ISO invariant character set. Some keyboards may also be missing some characters to type in C source code. To

solve these problems the idea of trigraph sequences were introduced in ANSI C as alternate spellings of some characters.

Character sequence	C Source Character
??	#
??(	[
??/	\
??)	]
??’	^
??<	{
??!	|
??>	}
??-	~

Trigraph Sequences

2.4.1.3 Integer Type

Integer is the most natural representation of numbers in a computer. Therefore, it is the most efficient data type in terms of speed. The size of an integer is usually the word size of the processor, although the compiler is free to choose the size. However, ANSI C does not permit an integer, which is less than 16 bits.

2.4.1.3.1 Integer constant

Integer constants can be denoted in three notations, decimal, octal or hexadecimal. Octal constants (ANSI C) begin with 0 and should not contain the digits 8 or 9. Hexadecimal constant begins with 0x or 0X, followed by the combination of 0 to 9, A to F (in either case). The constant, which starts with a non-zero number, is a decimal constant. If the constant is beyond the range of the integer then it is automatically promoted to the next available size, say unsigned or long.

```
int i = 12;

int j = 012;

// beware; octal number.
```

It is not only the beginners who easily forget that 012 and 12 are different and that the preceding 0 has special meaning. Octal constants start with 0 is certainly non-intuitive and history shows that it has lead to many bugs in programs.

Exercise 2.4:

Have you ever thought of if 0 an octal constant or decimal constant. Does the information if 0 is decimal or not make any difference in its interpretation/usage?

2.4.1.4 Enumeration Type

Enumeration is a set of named constants. These constants are called enumerators. Enumeration types are internally represented as integers. Therefore, they can take part in expressions as if it were of integral type. If the variables of enumeration type are assigned

with a value other than that of its domain the compiler may check it and issue a warning or error.

The use of enums is superior to the use of integer constants or #defines because the use of enums makes the code more readable and self-documenting.

Exercise 2.5:

Is it possible to have the same size for short, int, long in some machine?

2.4.1.5 Floating-Point Type

These types can represent the numbers with decimal points. Floats are of single precision and as the name indicates, doubles are of double precision. The usual size of double type is 64 bits.

All the floating-point types are implicitly signed by definition (so ‘unsigned float’ is meaningless). Depending on the required degree of efficiency and available memory, we can choose between float and double.

ANSI C does not specify any representation standard for these types. Still it provides a model, whose characteristics are guaranteed to be present in any implementation. The standard header file <float.h> defines macros that provide information about the implementation of floating point arithmetic.

All floating-point operations are done in double precision to reduce the loss in precision during the evaluation of expressions [Kernighan and Ritchie 1978]. However, ANSI C suggests that it can be done in single precision itself, as the type conversion may be costly in terms of processor time.

2.4.1.5.1 A little bit of history

Since C was originally designed for writing UNIX (system programming), the nature of its application reduced the necessity for floating point operations. Moreover, in the hardware of the original and initial implementations of C (PDP-11) floating point arithmetic was done in double precision only. For writing library functions seemed to be easy if only one type was handled. For these reasons the library functions involving mathematics (<math.h>) was done for double types and all the floating point calculations were promoted and was done in double precision only.

To some extent it improved efficiency and made the code simple. However, this suffered many disadvantages. In later implementations, most of the implementations had most efficient calculations in single precision only. Later the C became popular in engineering applications which placed great importance on floating point operations. For these reasons the ANSI made a change that for floating point operations implementations may choose to do it in single precision itself.

Pains should be taken in understanding the floating-point implementation. Although the actual representation may vary with implementations, the most common representation is the IEEE standard.

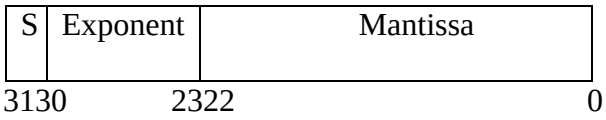
2.4.1.5.2 IEEE Standard

The floating point arithmetic was one of the weak points in K&R C. As indicated previously, one of the changes suggested by the ANSI committee is the recommended use of IEEE floating point standard.

2.4.1.5.2.1 Single Precision Standard

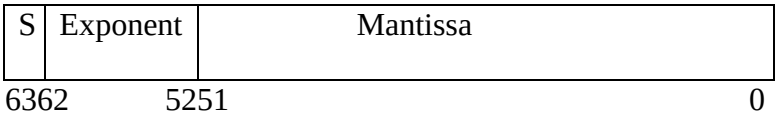
This standard uses 32 bits (4 byte) for representing the floating point. The format is explained below.

- The first bit reserved for sign bit.
- The next 8 bits are used to store the exponent (e)in the unsigned form
- The remaining 23 bits are used to store mantissa(m)



2.4.1.5.2.2 Double Precision Standard

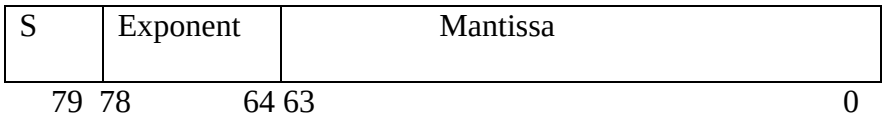
- The first bit reserved for sign bit.
- The next 11 bits are used to store the exponent (e)in the unsigned form
- The remaining 52 bits are used to store mantissa(m)



2.4.1.5.2.3 Format of Long Double

For long double the IEEE extended double precision standard of 80 bits may be used.

- The first bit reserved for sign bit.
- The next 15 bits are used to store the exponent (e)in the unsigned form
- The remaining 64 bits are used to store mantissa(m)



2.4.1.5.3 Limits in <float.h>

There are four limits in specifying the floating-point standard. They are minimum and maximum values that can be represented, the number of decimal digits of precision and the delta/epsilon value, which specifies the minimal possible change of value that affects the type(FLT_MIN, FLT_MAX, FLT_DIG and FLT_EPSILON respectively.

Care should be taken in using the floating points in equality expressions since floating values cannot exactly be represented. However, the multiples of 2's can be represented accurately without loss of any information in a float/double(i.e. 1,2,4,8,16... can be represented accurately).

```
float f1 = 8.0;
double d1 = 8.0;
if(f1 == d1)
```

```
printf("this will certainly be printed");
```

It is usual to check floating-point comparisons like this,

```
if(fp1 == fp2)
    // do something
```

As we have seen, this may not work well (since the values cannot be exactly represented). Can you think of any other way to check the equality of two floating points that is better than this one?

```
if (fabs  (fp1 - fp2) <=  FLT_EPSILON)
    // do something
```

Where FLT_EPSILON is defined in <float.h> and stands for the smallest possible change that can be represented in the floating point number. You check for the small change between the two numbers and if the change is very small you can ignore it and consider that both are equal. If this still does not work, try casting to double to check the equality. Of course, this will not work if you want to the values exactly.

What happens if a floating point constant that is very big to be stored in a floating-point variable (like when 1e+100 is assigned to a float variable)?

The number is a huge number that a float variable cannot contain. So, an overflow occurs and the behavior is not defined in ANSI C (since it is an undefined behavior, it may produce an exception or runtime error or even may continue execution silently).

Exercise 2.6:

Is it possible to have the same size for float, double and long double types in some machine?

2.4.1.5.4 Floating constant

Floating point constants can either by represented with ordinary notation or scientific notation. In ordinary notation, we will include a decimal point between the numbers. The scientific notation will be in the form mantissa |E/e| exponent. The mantissa can optionally contain a decimal point also. A floating-point constant is never expressed in octal or hexadecimal notation.

2.4.1.6 *Pointer Type*

A pointer is capable of holding the address of any memory location. Pointers fall into two main categories,

- pointers to functions and
- pointers to objects.

A function pointer is different from data pointers. Data pointers just store plain address where the variable is located. On the other hand, the function pointers have several components such as return type of the function and the signature of the function.

Pointers are discussed in the chapter dedicated for it.

2.4.1.6.1 Pointer constants

Constants, which store pointers (address of data), should be called as pointer constants. Pointer constants are not supported in C because giving the user the ability to

manipulate addresses makes no sense. However, there is one such address that can be given access to freely. That is NULL pointer constant. This is the only explicit pointer constant in C.

In DOS (and Windows) based systems, the memory location 0x417 holds the information about the status of the keyboard keys like CAPS lock, NUM lock etc. The sixth bit position holds the status of the NUM lock. If that bit is on (1) it means that the NUM lock is on in the keyboard and 0 means it is off. The program code (non-portable, DOS based code) to check the status looks as follows,

```
char far *kbdptr = (char far *)0x417;
if(*kbdptr&32)
        printf("NUM lock is ON");
else
     printf("NUM lock is OFF");
```

Here the requirement of pointer constant is there and that role is taken by the integer constant and the casting simulates a pointer constant to store the address 0x417 in ‘kbptr’.

2.5 Aggregate Type

The aggregate types are composite in nature. They contain other aggregate or scalar types. Here logically related types are organized at physically adjacent locations. It consists of array, structure and union types, these will be discussed in detail later.

2.6 Void Type

Void specifies non-existent/empty set of values. Since it specifies non-existent value, one cannot create a variable of type void.

2.7 Function Type

The function types return (specific) data types.

Why should functions be considered as a separate variable type?. The following facts make it reasonable,

- The operators *, & can be applied to functions as if they are variables,
- Pointers to functions is available,
- They can participate in expressions as if they are variables,
- Function definitions reserve space,
- The type of the function is its return type.

For the close relationship between the variables and functions, functions are also considered as a variable type.

2.8 Derived Types

Arrays and pointers are sometimes referred to as derived data types because they are not data types of their own but are of some base data types.

2.9 Incomplete Types

If some information about the type is missing, that will possibly given later is referred to as incomplete type.

```
struct a;
// incomplete type
int i = sizeof(a);
// error(as sizeof is applied to a incomplete type)
```

Here the structure ‘a’ is declared and not yet defined. Therefore, ‘a’ is an incomplete type. The definition may appear in the later part of the code like this:

```
struct a{ int i };
// filling the information of the incomplete type
int i = sizeof(a);
// o.k. now necessary information required for struct a is known.
```

Consider,

```
typedef struct stack stackType;
```

Here the struct stack can be of incomplete type.

```
stackType fun1();
struct stack fun2();
```

are function declarations that make use of this feature that the struct stack and stackType are used before its definition. This serves as an example of the use of forward declarations.

Another example for such incomplete type is in case of arrays:

```
typedef int TYPE[];
```

```
TYPE t = {1,2,3};
printf("%d",sizeof(t));
// acceptable. necessary information about it is known.
printf("%d",sizeof(TYPE));
// error. Sizeof TYPE is unknown.
```

In these two examples, it is evident that some information is missing to the compiler and so it issues some error. Lets now move to the case of pointers, an example for logical incomplete type, where it is not evident that some information is not available.

```
int *i = 0x400;
// i points to the address 400
*i = 0;
// set the value of memory location pointed by i;
```

The second statement is problematic, because it points to some location whose value may not be available for modification. This is an example for 'Incomplete type' in case of pointers in which there is non-availability of the implementation of the referenced location. Using such incomplete types leads to undefined behavior.

Point to Ponder

The void type is an incomplete type that cannot be completed.

2.10 Type Specifiers

Type specifiers are used to modify the data type's meaning. They are unsigned, signed, short and long.

2.10.1 Unsigned and Signed

Whenever we want non-negative constraint to be applied for an integral type, we can use the unsigned type specifier. The idea of having unsigned and signed types separately started with the requirement of having a larger range of positive values within the same available space.

Unsigned types sometimes become essential in cases where low-level access is required like encryption, data from networks etc.

The signed on other hand operates in another way, making the MSB to be a sign bit; it allows the storage of the negative number. It also results in a side effect by reducing the range of positive values. If we do not explicitly mention whether an integral type is signed or not, signed is taken as default (except char, which is determined by the implementation).

The way signed and unsigned data types are implemented is same. The only difference is that the interpretation of the MSB varies.

The following example finds out if the default type of character type in your system is signed or unsigned. In addition, the property of arithmetic and logical fill by using right shift operator is demonstrated.

```
{
 char ch1=128;
 unsigned char ch2=128;
 ch1 >>= 1;
 ch2 >>= 1;
```

```
 printf("Default char type in your system is %s“,
 (ch1==ch2) ? “unsigned " : “signed ");
}
```

If you are very serious about the portability of the characters, use characters for the range, which is common for both the unsigned and signed (i.e. the values 0 to 127). If the range exceeds that limit, use integers instead.

Unsigned types obey the laws of arithmetic modulo (congruence) 2n, where n is the number of bits in the representation. So unsigned integral types can never overflow. However, it is not in the case of floating point types. This is one of the desirable properties of unsigned types.

Exercise 2.7:

Predict the output of the program :

```
main(){
int i= -3,j=i;
i>>=2;
i<<=2;
if(i == j) printf(“U are smart”);
else printf(“U are not smart enough”);
}
```

2.10.2 Short and Long

Short, long and int are provided to represent various sizes of possible integral types supported by the hardware. The ANSI C tells that the size of these types are implementation defined, but assures that the non-decreasing order of char, short, int, long is preserved (non-decreasing order means that the sizes are char <= short <= int <= long and not the other way).

2.11 Type Qualifiers

If we need to add some special properties to the types we can use the type qualifiers. The available type qualifiers are const and volatile. ANSI C added these qualifiers to the language. The idea of const objects is from Pascal.

2.11.1 Const Qualifier

Whenever we want some value of an object to be unchanged throughout the execution of the program, we can use the const qualifier. An expression evaluating to an const object should not be used as lvalue. The objects declared are also sometimes called as symbolic constants.

Constness is a compile time concept. It just ensures that the object is not modified and is documentation about the idea that it is a non-modifiable one. It helps compiler catch such attempts to modify the const variables.

The default value for uninitialized const variable is 0. Also if declared as a global one its default linkage is extern.

```
extern int i;
```

```
// implicitly initialized to 0.
// If in global scope it has extern linkage
```

Using symbolic constants sometimes may be useful in compile time operation sometimes called as *constant folding* (Not to be confused with constant-expression evaluation).

```
const float PI = 3.14;
for( i = 0 ; i < 10 ; i++ )
     area = 2 * PI * r;
```

In this code, the compiler may replace PI with 3.14, which helps creating efficient code. (still smarter compilers may treat 2 * 3.14 as a constant expression and evaluate the expression at compile time itself).

Note : *const is not a command to the compiler, rather it is a guideline that the object declared as const would not be modified by the user. The compiler is free to impose or not impose this constraint strictly.*

Exercise 2.8:

Can we change the value of the const in the following manner? If yes then what is the effect of such changing of value?

```
*(&constVar) = var?
```

Exercise 2.9:

What is the difference between the constness as in const int i = 10 and ‘10’?

2.11.2 Volatile Qualifier

The compiler usually makes optimization on the objects.

```
while ( id < 100 )
{
flag = 0;  // set flag to false
a[i]  = i++;
}
```

Here the optimization part of the compiler may think that the setting of flag to 0 is repeated 100 times unnecessarily. So it may modify the code such that the effect is as follows,

```
flag = 0;  // set flag to false
while ( i < 100 )
{
a[i]  = i;
}
```

where both the loops are equivalent. However, the second is optimized version and executes faster. While making optimization, it assumes that the value of the object will not change without the knowledge of the compiler. But in some cases, the object may be modified without the knowledge (control/detection) of the compiler (read about types of variables in the beginning of the chapter. 'without knowledge of the compiler' means it is an asynchronous object). In those cases, the required objective may not be reached if optimization is done on those objects. If we want to prevent any optimizations on those objects, then we can use volatile qualifier.

The objective is to delay the program for a considerable amount of time and print the final time later. The code uses a location 0x500 where the current time is updated and stored in this location in the system.

```
const int SIGNIFICANT = 60;
int *timer = 0x500;
// asynchronous variable
// assume that at location 0x500 the current time is available
int startTime = *timer, currTime= *timer;
// initialize both variables with current time
while( (currTime – startTime) < SIGNIFICANT )
{   // loop until the difference is SIGNIFICANT
   currTime = *timer;     //update currTime
}
printf("%d",currTime);
```

The compiler thinks that the assignment

```
currTime = *timer;
```

is executed again and again without any necessity and puts it (optimizes the code) out of the loop and the code looks as follows,

```
const int SIGNIFICANT = 60;
int *timer = 0x500;
int startTime = *timer, currTime= *timer;
if( (currTime – startTime) < SIGNIFICANT )
currTime = *timer;
// optimizes and executes the statement only once.
```

```
while( (currTime – startTime) < SIGNIFICANT )
{
// it goes to infinite loop now.
 }
printf("%d",currTime);
```

In addition, as you can see the problem is that the optimization is made on the asynchronous variable leading to problem. Qualifying the variable as volatile makes avoid such undesirable optimizations.

```
volatile  currTime = *timer;
// will prevent optimization done on currTime
```

Before seeing another example, lets see what it means to have both const and volatile qualifiers for a same variable. Say,

```
const volatile int i;
```

Here i is declared as the variable that the program(mer) should not modify but it can be modified by some external resources and so no optimizations should be done on it.

Let us see another example. Consider that your objective is to access the data from a serial port. Its port address is stored in a variable and using that you have to read the incoming data.

```
int * const portAddress = 0x400;
// assume that this is the port address.
// and you shall not modify the port address
while ( *portAddress != 0 ) //some terminating condition
 {
 *portAddress = 255;       //before reading it set it to 255
```

```
                                    // and this shouldn't be optimized
 *portAddress = readPort();    //  read from port
}
```

had optimization be done on the code, the code will look like this.

```
int * const portAddress = 0x400;
// assume that this is the port address.
// and you shall not modify the port address
while ( *portAddress != 0 ) //some terminating condition
 {
     *portAddress = readPort();     //  read from port
 }
```

the compiler may think that the assignment,

```
*portAddress = 255;
```

is a dead code because it has no effect on the code since *portAddress = readPort() is done immediately (like, if code is available like a = 5; a =10; then the first statement becomes meaningless).

Therefore, the optimized code will not work as expected. In these cases use volatile to specify that no optimizations to be done on that object.

So, to achieve this change the declaration to,

```
   volatile int * const portAddress = 0x400;
```

meaning that the address stored in the portAddress cannot be changed and the value pointed by the portAddress should not be optimized.

Volatile may be applied to any type of objects (like arrays and structures). If this is done then the object and all its constituents will be left unoptimized.

Other examples for such cases where volatile should be used are:

- the memory location whose value is used to get the current time, accessing the scan-code form a keyboard buffer using its address and in general - memory mapped devices,
- writing code for interrupt handling. There may be some variables that is accessible both by the interrupt servicing routine (ISR) and the regular code. In such cases the optimizations done by the compiler may lead to erroneous results,
- writing code where multithreading is done. For example, say two threads access a memory location. Both threads store the value of this variable in a register for optimization. Since both threads work independently, if one thread changes the value that is stored in a register, it remains unaffected to the variable stored in register in the another thread. If the variable is declared as volatile it will not be stored in a register and only one copy will be maintained irrespective of the number of threads,
- writing code in multiprocessing environment, and the idea is similar to multi-threaded environment. There may be shared memory locations and more than one processor may access and modify the value leading to inconsistent values.

In all such cases volatile must be used to prevent optimizations be done on those variables.

2.12 Limits of the Arithmetic Types

Limits are the constraints applied in the representation of the data type.

2.12.1 Translation Limits

Translation limits specify the constraints, with how the compiler translates the sequence of the characters in the source text to the internal representation.

E.g. ANSI C defines that the compiler should give the support to at least 509 characters in a string literal after concatenation.

2.12.2 Numerical Limits

The range of values, which the data type can represent, is specified by the numerical limits.

The standard header files <limits.h> and <float.h> defines the numerical limits for the particular implementation.

However, the values in the <limits.h> define the minimal and maximal limits for the types. To find out the actual limits in the system that you are working the following method can be used (although other implementations are possible this implementation seems to be direct, handy and works well).

```
#define INT_MAX (((unsigned )~0)>>1)
```

Similarly, the other macro constants can be defined. This is for integer where the size of integer is implementation defined. But for char the size is already known. So

writing our own versions of CHAR_MIN, CHAR_MAX is direct (But keeping the fact in the mind that the char implementation could be signed or unsigned by default).

```
# if (((int)((char)0x80)) <0 )
     #define CHAR_MAX 0x7f
     #define CHAR_MIN 0x80
#else
     #define CHAR_MAX 0xff
     #define CHAR_MIN 0x00
#endif
```

2.13 Creating Type Names

Typedefs create new type names. This adds a new name for an existing type rather than creating a new type.

Typedefs are subjected to the rules for scope.

```
{
 typedef char WORD;
 // WORD is char
 WORD w1;
 // w1 is an char
     {
     typedef int WORD;
     // WORD is int
```

```
        WORD w1;
        // w1 is an int now
        }
    }
```

2.13.1 #define and typedef

Consider the two ways to have a ‘byte’ type:

```
typedef char byte;
byte b1,b2;
// new type name byte is created.
```

It may seem that this is straightforward to use #define to do the same.

```
#define BYTE char
```

The ability to create new type names using #define and typedef seems to be similar and of same power. But this similarity is superficial. Lets start with a very simple example:

```
#define  ptr1 char *
typedef char * ptr2;
ptr1 p1 = “someStr”, p2 = “anotherStr”;
// error : cannot convert from char[11] to char
ptr2 p3 = “someStr”, p4 = “anotherStr”;
// o.k
```

Because the * applies only to p1 and not to p2. This problem doesn’t arise in the case of typedef. Now, consider:

```
int var;
```

```
#define  ptr1 char *
typedef char * ptr2;
const ptr1 myPtr1 = &var;
const ptr2 myPtr2 = &var;
// or ptr2 const myPtr1 both are equivalent.
myPtr1 = NULL;
// o.k.
myPtr2 = NULL;
// error !
```

myPtr1 is equivalent to be declared as const char * myPtr; which is simple textual replacement. But myPtr2 is equivalent to be declared as char * const myPtr; because ptr2 is of type char pointer. In addition, it shows typedef is not textual replacement.

The capability of creating a new type name using typedef is superior to #define.

```
typedef void(*fType)();
```

declares fType to be of type void (*) ()

i.e. fType is pointer to function with return type void and taking no arguments.

```
fType myPtr;
myPtr = clrscr;
// myPtr points to the function clrscr.
```

The strength of the typedef lies in the fact that they are efficiently handled by the parser. Since #define may result in hard-to-find errors, it is advisable to replace them with typedefs.

2.13.2 Some Interesting Problems

Consider the following code,

```
typedef int numTimes;

short numTimes times;
```

the code is very simple and direct but the compiler flags an error. Guess why? The following line is the culprit,

```
short numTimes times;

// Error : Cannot use modifiers for typedefs
```

the type that is defined by the typedef cannot be used with modifiers. The reason is that the types declared with typedef are treated as special types by the compiler and not as textual replacement. (If it were textual replacement, this code should be legal). So applying short to modify the type numTimes to declare times as short int fails.

To achieve the same result numTimes has to be again typedefed to the new type,

```
typedef short numTimes shortNumTimes;

shortNumTimes times;  // now o.k
```

typedefs also have some peculiarly qualities, for example, a typedef can declare a name to refer the same type more than once.

```
typedef int something;

typedef something something;
```

is valid!

Now consider the following example,

```
typedef char * charPtr;

const charPtr str = "something";
```

```
str = “another”;
```

issues an error stating that “l-value specifies a constant object”. What went wrong? The programmer expected to declare str as

```
const char *str = “something”;
```

and used a typedef instead to declare it indirectly as,

```
const charPtr str = “something”;
```

But this actually stands for,

```
char *const str = “something”;
```

which states str is a const pointer to a character, and so an attempt to change it using the assignment,

```
str = “another”;
```

flags an error.

To force what the programmer intended to do, the code should be modified as follows,

```
typedef const char * constCharPtr;
constCharPtr str = "something";
str = "another";
```

This is an another example to show that the typedef is not the same as the textual replacement as in #define.

As I have said, typedef names share the name space with ordinary identifiers. Therefore, a program can have a typedef name and a local-scope identifier by the same name.

```
typedef char T;
int main()
{
T  T=10;
printf("%d",T);
}
```

Here the compiler can distinguish between the type name and the variable name.

One more interesting problem arises with typedefs because of this property. Consider that I have declared a type T:

```
typedef char T;
```

you want to declare a const int variable named as T in the same scope as type T:

```
const int T = 10;
```

But you know that when the type name is missing in a declaration, the type int is assumed. So you can write this declaration of const int variable T like this:

```
const T = 10;
//ask compiler to assume the type int here as a
//shortcut for the previous declaration.
```

But you know that the name T also stands for char type since that name is typedefined. So for the compiler declaration looks like as if it is given as:

```
const char;
```

where the compiler thinks that variable name is missing. So it issues an error.

Therefore, the rule is that, *when declaring a local-scope identifier by the same name as a typedef the type name must be specified explicitly*.

Exercise 2.10:

Is there any possibility that sizeof (typeOrObject) operator return value 0 as the size of the type/object?

2.13.3 Abstracting Details

Typedefs are useful in abstracting the details from the users.

```
struct _iobuf {
   char *_ptr;
   int   _cnt;
   char *_base;
   int   _flag;
   int   _file;
   int   _charbuf;
   int   _bufsiz;
   char *_tmpfname;
   };  // one possible implementation
typedef struct _iobuf  FILE;
```

The detail behind FILE is abstracted and the user uses FILE freely as if it is a datatype.

Typedefs may be useful in cases like this and particularly in the complex declarations.

Consider that your requirement is to access the video buffer to manipulate screen. It would be very handy to access the whole screen as 25 * 80 array.

```
// This is machine specific implementation given here
// for demonstration of typedefs
#if defined (MONO)  // if mono monitor
     #define BASE 0xb0000000
     //for mono monitor video memory begins here
#else
      #define BASE 0xb8000000
     // for other monitors video memory begins here
#endif

#define ROWS 25
#define COLS  80
typedef  struct {
     unsigned char ch;
     // character takes one byte
     unsigned char attr;
     // its attribute takes one byte
     }unit;
     // this makes one unit

typedef unit videoMemory[ROWS][COLS];
//  The screen is 25 * 80 array of units.

videoMemory far * screen = (videoMemory far*) BASE;
// define the screen
```

```
#define SCREEN      (*screen)


void setChar(int xPos, int yPos, unsigned char ch,
unsigned char attr)
     {
     SCREEN[xPos][yPos].ch = ch;
          // set a character to corresponding x and y positions
     SCREEN[xPos][yPos].attr = attr;
          // set the character's attribute
     }
```

here typedefs abstract the detail of the type, and allows freely creating and manipulating objects as if they were built-in types.

2.13.4 Portability and Typedef

Typedef helps in increasing the portability! One of the main reasons for using typedef is to make it easy for a program to be more portable with no or minimal changes. The types declared by typedefs can be changed according to the target machine by the implementation.

```
size_t strlen(const char *);
```

notice that strlen returns size_t. In our compiler it was defined in <stdio.h> as,

```
typedef unsigned size_t
```

other examples are clock_t and time_t defined in <time.h>. One such implementation is,

```
typedef  long time_t
```

However, with equal probability can be implemented as,

```
typedef  unsigned long time_t
```

The choice is made based on the target machine and the compiler. Therefore, it effectively suits the needs providing portability across platforms without requiring the source code leaving untouched of change. In other words, if typedefs are used for data types that may be machine dependent, only the typedefs need change when the program is moved. If any up-gradation of the software is needed it is enough that the typedef is changed.

```
typedef unsigned int ptrdiff_t
```

when it is needed to be used in huge arrays can be modified and used as,

```
typedef unsigned long ptrdiff_t
```

Exercise 2.11:

What is year 2038 problem? (hint: it is related to typedefing time_t discussed here)

2.14 Type Equivalence

Two types are said to be equivalent if they have same type attributes.

```
{
  typedef int dummy;
  int a;
  dummy b;
  auto c;
```

```
}
```

Here a, b, c are said to be variables equivalent in type. More technically, two types are said to be of equivalent types if there is structural compatibility (in C). Structural compatibility applies for all types except structures, and for structures, name compatibility is required. Consider the two structure definitions,

```
struct div_t {
     int quot;
     int rem;
     }div1 = {1,2};
struct t_div {
     int quot;
     int rem;
     }div2;
div2 = div1;
// error, cannot convert from div_t to div-t
div2 = (t_div) div1;
// error, cannot cast from div_t to div-t
// name-compatibility is required for structure assignment.
```

As you can see in this example, the two structures only differ by name and so cannot be assigned to each other.

2.15 TYPE CONVERSIONS

C is a typed language. This means every variable and expression belongs to a particular type. It is not a strongly typed language[3]. We say it is not strongly typed because, the variables can be freely assigned with the variables of other types and strict type checking is not made. For e.g., it is almost universal among the C programmers to interchange 'int' and 'char' assignments.

Even the standard library sometimes follows it; as in the prototype,

```
int getchar( );
```

where it is customary to assign the value returned from getchar() to a character without explicit cast. Similarly conversions from void * to other pointer types are assumed and explicit casting is not normally made:

```
int * iObj = malloc(sizeof(int));
```

Type casting is of two types, explicit and implicit.

2.16 Explicit Conversions

Conversions, if made using a casting operator, are called as explicit conversions. Casting should not be done to just to escape from the errors and warnings issued by the compiler. The compiler actually wants to warn you that there is possibility of loss of information during conversion and to make sure that you really know what you want to do. Explicit casting is the more powerful (than assignment and method invocation

[3] It must be said here that viewpoints differ on whether C is strongly typed or not. The reader is encouraged to adopt the viewpoint that he/she is most comfortable with. I strongly suggest to treat C as not a strongly typed language and that is the viewpoint adopted and accepted widely.

conversions) and forces the compiler to change the type. It also improves the readability by saying that the type conversion is done on that point.

Casting sometimes indicates the bad design. But in some cases, casting is necessary. If type conversions are required, it is always recommended to use explicit casting.

Let us start with the mistakes the novice C programmers make. Consider the following code written by one such to calculate his/her percentage of marks.

```
float percentage;
int marksObtained, totalMarks;.
marksObtained = 973;
totalMarks = 1000;
percentage = ( marksObtained/totalMarks )*100;
     printf("Percentage  =  %3.2f%", percentage);
```

The programmer expects the program to print 97.30%, and is disappointed to see that it prints 0%. This is because the division operator performs an integer division that yields 0 for (973/1000). So to get proper results, he learns do an explicit conversion:

```
percentage = (float)( marksObtained/totalMarks )*100;
```

for his/her dismay, again gets 0%.

What went wrong? In the above example, marksObtained and totalMarks both are integers. So integer arithmetic is made on it resulting in 0 (of course, the resultant 0 will be type-casted to 0.0). Finally arriving at the correct solution:

```
percentage = (float)marksObtained/totalMarks*100;
```

2.17 Implicit Conversions

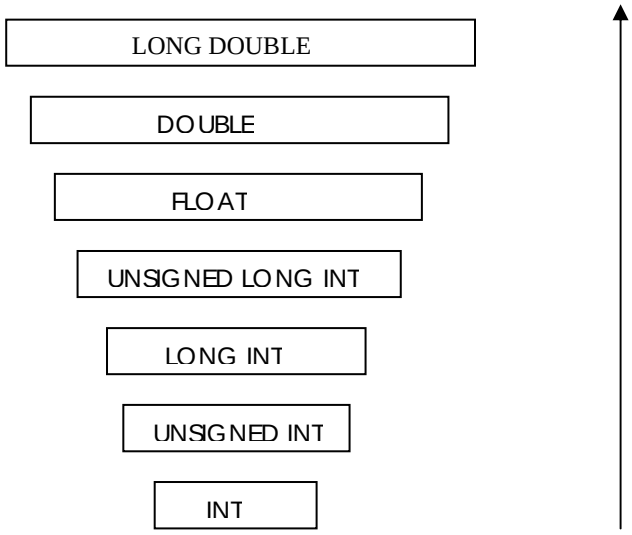
When operators in expressions involve different types, implicit conversions occur. Since the compiler automatically does implicit conversions, they may result in hard-to-find errors (since these conversions may not be well predicted).

Consider this problem involving implicit conversions.

```
char ch;
while( ( ch=getchar( )) != EOF )
       //   skip
```

If the char is unsigned by default, then the loop may not terminate. Because comparing for equality of EOF (usually -1), with an unsigned number yields a false.

The problem will be solved if the variable ‘ch’ is declared as an integer, which is signed.



Implicit type conversion hierarchy

2.17.1 Integral (Widening) Promotions

They are conversions from short, char, enum, and bitfields to int and float to double. These promotions do not lose any information regarding sign or order of magnitude (but for unsigned types read about unsigned & value preserving). These conversions may occur to improve efficiency (for e.g. integral types are internally represented as integers).

In the expressions (including the ones that have constants), when unsigned chars and unsigned shorts are involved, integral promotion takes place to accommodate the resulting value. Based on the resulting type, signed int or unsigned int, the results may differ considerably. This issue of promotion to either signed int or unsigned one is called as unsigned/value preserving.

It is to be noted that ANSI standard supports value preserving.

2.17.2 Unsigned Preserving

If the sign (the unsigned-ness) has to be preserved, i.e., if promoted to unsigned int, then it is called as unsigned preserving or just 'sign preserving' (since preserving the sign is given precedence of).

```
unsigned result = 1u - 2;  // suffix U/u means unsigned
if( result < 0  )
   printf("This will never be printed");
else
```

```
printf("%u", result);
// In our system, it prints 65535.
```

In this case, the unsigned and signed constants are mixed. So the signed value is promoted to unsigned resulting in a very large value. So always, the else part is executed.

You can force sign preserving by giving the U/u suffix for operands.(e.g. 2u-3u).

2.17.3 Value Preserving

On the other hand, if the promotion is to signed int to preserve the value rather than the sign, it is called as value preserving.

Sign preserving is supported in K&R C (and in UNIX C compilers) whereas ANSI C supports value preserving.

To solve this problem of signed and unsigned mixing, either cast the signed to unsigned if you are sure that it cannot take negative values. Or else, cast unsigned to signed if its value cannot exceed INT_MAX. This will help eliminate nasty bugs due to this problem.

Exercise 2.12:

Comment on the behavior of the following code segment:

```
int i = -1;
while( i++ < sizeof(char) )
        printf("%d",i);
```

2.17.4 Assignment Conversions

Assignment conversions change type of the RHS of an assignment statement to the type of the LHS if they are not the same. Either truncation or widening of values occurs.

2.17.5 Arithmetic Conversions

Arithmetic conversions occur in expressions. Certain operators require their operands to be of specific types. These conversions take place when the given type does not match with the required type. They follow data type hierarchy.

2.17.6 Conversion Due To Function Calls

When the formal parameter and the actual parameter differ in type, the actual parameter is automatically converted into actual parameter's type. The same rule applies while returning values. However, it is not recommended to rely on this facility.

2.17.7 Floating Point Conversions

Conversion of a floating-point value to an integer value results in truncation of the decimal position. If the number is negative, truncation is defined to be towards zero. If a floating-point variable is assigned with an integer constant, it need not exactly represent it (since exact precision of integer values is not possible).

2.17.8 Pointer Conversions

```
int number = (int)“This is the message to be displayed” ;
// number may hold the address of the string
printf(“%s”, number);
```

If you run this code, some of you will be amazed to see the string printed on the screen.

In [Kernighan and Ritchie 1988], pointers are treated as unsigned int when they involve in expressions and sign-preserving conversions are made. Conversions from one pointer type to another and back results in no loss of information. However, conversions from pointer type to int/long and back is not assured (it is also wrong to assume that always the size of int is enough to hold the pointer). This is what [Kernighan and Ritchie 1988] puts in words, "...A pointer can be assigned to any integer value, or a pointer value of another type. This can cause address exceptions when moved to another machine". [ANSI C 1988] also tells the same, but in other words, that there is no assurance of equality between pointers and integers.

The freedom given by C in many cases like the one given here may be harmful to portability of your program (one exception for this case is if the integer value is 0 which can be freely assigned).

A classical example is assigning the NULL to various types without explicit conversions. Normally NULL is defined as ((void*) 0) and it is ubiquitous to see the code like this,

```
char * c =NULL;
```

```
somePointer = NULL;
```

without explicit casting (this is another place where the type are freely mixed without explicit casting in C).

Generic pointer type void* may be used as an intermediate for conversion from one pointer to another. It may also be used in the cases when the pointer type is not known.

Exercise 2.13:

Comment on the following code:

```
int (*fp)();

int *p1 = (int *) (fp);

int *p2 = (int *) (int) (fp);
```

2.17.9 Void Conversions

Any object can be converted to void. This conversion is to explicitly specify that the value is discarded. Functions with return type void specifies the discarding value of the function call.

2.17.10 No Conversions

No changes of value representation occur while conversions within *equivalent* types occur.

3 DECLARATIONS AND INITIALIZATIONS

3.1 Declaration

Associating an object name with a type is known as declaration of an object.

```
int add(int, int);
```

is the function declaration for add.

```
extern int i;
```

is an external declaration that does not reserve or associate any space with. Various types of declarations are categorized as:

- Explicit Declarations

These are declarations made explicitly in the program with no extra information assumed.

- Implicit Declarations

These are default assumptions made by the compiler when type or relative information is missing.

- External Declarations

External declarations specify that the object names are declared in some other place and will be available at the later stage.

- Duplicate Declarations

Declarations for object names can be repeated without changing the meaning. Example: Tentative declarations.

3.2 Forward References

Forward references are using of variables before their declarations. In C, they are allowed in only three cases.

1. In structure / enumeration / union tag names can be used in declarations before their declaration.

So declarations like,

```
struct a { struct b *next;};
struct b { struct a *next;};
```

are valid.

2. Labels after goto statements where that label is declared later.

```
goto end;
….
….
end :
     over( );
// end label here is the forward reference
```

3. Function calls, where the name of the function is undeclared.

```
int main()
{
fun();
}
```

at the point the name fun is encountered the name is not known to the compiler. Since it resembles a function call, it assumes it to be the name of the function and generates code for calling that function. In other words, it assumes it to be declared as,

```
extern int fun();
```

i.e. the following details are assumed:

- fun() is extern, so it is assumed that it is defined somewhere else,
- fun() takes any number of arguments (declaration fun() doesn't mean that it takes no arguments but that it takes any number of arguments. This is explained in the chapter on functions),
- return type of fun() is integer.

Knowing these details of what the compiler assumes for the forward declaration of functions, now you can reason it out why the following code makes compiler issue error message such as "type mismatch in the redeclaration of fun".

```
int main()
{
fun();
}
float fun()
{
return 1.0;
}
```

Since the compiler assumes that return type of fun() is integer and later its assumption is contradicted by the definition of the fun() that it returns float it issues an error.

Writing code that depends on the implicit assumptions due to forward references may lead to unexpected bugs in code. Consider one such code,

```
int main()
{
int i = fun();
// assume return type as int and capture that returned
// garbage value in i.
}
fun(int i)
{
// doesn't issue an error because fun() is assumed to take any number
// of arguments. Some garbage value is passed.
printf("%d",i);
}
```

this is an example of how such assumptions may lead to buggy code and such code should not be written.

In short do not depend on the forward declaration of functions and in such cases declare them explicitly so that the intention will be clear to both the compiler and the reader of the program. This will also help the compiler to catch errors due wrong number of arguments or wrong type of arguments as in the following,

```
extern float fun(int );
// good. fun() is declared explicitly and doesn't depend on forward
// declaration.
int main()
```

```
{
float f = fun(10);
}
float fun(int i)
{
return (float)i;
}
```

3.3 Definition

As opposed to declaration, definition associates an object name with a memory address.

3.3.1 Tentative Definition

A tentative definition is a declaration that can become either a declaration or a definition. In the later parts of a program, if that variable is defined then that tentative definition becomes a declaration. Else, it becomes a definition.

E.g.
```
/* this is in global area */
 int i;
    /* tentative definition. This becomes declaration after seeing
the next definition */
 int i=0;    /* definition */
```

ANSI C uses the presence or absence of an initializer to determine whether a declaration is a definition or an allusion.

If no real definition is available up to the end of the file only one definition for that variable is created and is initialized to zero. E. g.

```
// contents of source.h
 int i;
 int i;
  main( )
 {  }
 int i;
// end of source.h
// definition not available. So i is automatically defined
```

3.3.2 One Definition Rule (ODR)

One definition rule states that *there may be any number of declarations for a variable / function but there can be only one definition for that variable / function which is associated with a memory address.*

ODR seems to be easy and straightforward but is tough for the compiler to follow in the case of multi-file programs, because it must keep track of so many declarations and definitions and check for type equivalence. The problems associated with ODR crop up only at the time of linking, aggravating the problem.

Point to Ponder:

Declaration introduces a name into the program and a definition introduces some code.

3.3.3 Difference between declarations and definitions

A declaration associates a name with a type, whereas a definition associates the name with its type and space. So the definition performs the function of the declaration also. The aim of the declaration is to introduce to the compiler of the type associated with the name used.

There can be any number of declarations for a name and all the declarations must agree on the type of the entity referred to (also refer to ‘type equivalence’), whereas a definition should be unique.

Declarations and definitions are distinct in the cases of struct, union, extern and typedef. They are declared separately and variables are defined later.

```
struct anything{
     int i,j;
};
// as you can see it is always a declaration
struct anything something;
//definition
```

extern is a very good example for the declaration of variables.

```
extern int i;
// it is clear that this is a declaration.
// i does not reserve any space but can be used in the code.
```

Difference between the declaration and definition is still more evident in case of functions,

```
// declaration of add function
```

```
int add(int , int );
// the following defines function add
int add(int a, int b)
{
     return a+b;
}
```

Although typedef stands for “type definition”, typedef always is a declaration (of types).

```
typedef int INT;
```

As you can see the INT is only an information to the compiler and reserves no space (except that the name is entered into the symbol table). So typedefs are and can only be declarations.

In most cases, declarations and definitions are combined.

```
int i=0;
// declares and defines i
```

Now, what about the statement,

```
extern int i = 10;
```

is it a declaration , definition or an error?

This is a definition (extern keyword is ignored in this case). Since the initial value is specified, this information takes precedence, i get space allocated and initialized to 10.

3.3.4 Declarators

```
int i,j;
```

Here the variables i and j are said to be declarators i.e. the variables that are declared/defined are said to be the declarators. *A declarator is available to be referred in the program code and used from the point where the declaration/definition is made* (Exception for this is forward references).

So declarations like,

```
int a = sizeof (a);

int i = 0, j = i, k = j;

enum some{Sunday = 1,holiday = Sunday};
```

are valid. These examples are meaningful and works correctly but not the following:

```
int a = 10;

int b = b + a;

// b now contains some garbage value, since it is a automatic

// variable and garbage in is garbage out(GIGO)
```

Consider the following example,

```
int main()

{

int local = global;

// error. The compiler does not know variable name 'global'

// at this point.

}

int global = 10;

void fun()

{

int local = global;
```

```
// o.k. The compiler knows variable name 'global' at this point.
}
```

to overcome this declare like this as we did in case of functions,

```
extern int global;
int main()
{
int local = global;
// o.k. The compiler now knows variable name 'global at this point.
}
int global = 10;
```

3.3.5 More about declarators

Consider the following declaration,

```
static int i,j;
```

this declares both i and j as static integers. But the following,

```
int* i,j;
```

declares i as integer pointer and j as an integer. In short the * declared is applied only to the immediate declarator. The space between the * and declarator doesn't make any difference,

```
int *i, j;
```

If you intend to declare both i and j as pointers to integers, do it explicitly as in,

```
int *i, *j;
```

If you want to make the distinction clear to the reader of the program, you can explicitly put it like this:

```
int (*i), j;
```

Consider this example:

```
int (**ipp), (*ip), (i);
```

Note that this also becomes an example for redundant parenthesis, where parenthesis is used for grouping. This may help and be clearer to a novice programmer.

In short take care to make sure that you declare the declarators correctly when more than one such is declared separated by commas.

3.4 Ordering In Declarations/ Definitions

Any order of type modifiers, storage class specifiers is allowed. Thus,

```
static int unsigned ch;
int signed extern i;
const volatile int unsigned static a;
```

are legal. The order may change and makes no difference. So,

```
const volatile int unsigned static a;
int const static volatile unsigned a;
unsigned const static volatile int a;
```

..... and all the 32 possible combinations are equivalent. Only legal combinations are allowed. Legal combinations mean that the semantics of the declaration/definition should be meaningful and valid.

```
unsigned int signed i;
```

```
int int short i;
```

declarations like this makes no sense and will flag error. Declarators occur only at the end of declarations/definitions and thus cannot be moved forward,

```
static long i signed;    //error
```

3.5 Scope, lifetime and visibility

Whenever you are declaring a variable, you automatically determine the scope, lifetime and visibility. These three are important issues associated with any variable. Understanding the proper meaning and usage of these concepts is essential for good programming.

3.6 Scope

Scope is defined as the area in which the object is active. There are five scopes in C. They are,

3.6.1 Program Scope

They are declarations at the top most layers. They are available up to the life of a program. All the functions have this scope. This is otherwise known as global scope.

3.6.2 File Scope

It has scope such that it may be accessed from that point to the end of the file.

```
void dummy(void) { }
// absence of static automatically gives program scope to dummy()
```

```
static  void dummy(void) { }
// static keyword here gives fn. dummy a file scope
```

3.6.3 Function Scope

Only labels have this scope. In this scope, they are active up to end of the function.

```
void printFun()
{
print:
     printf("i is less than j");
}
int main()
{
int i=1,j=2;
if(i < j)
     goto print;
}
```

This code will be flagged error by the compiler saying that the label print is not known because labels have only function scope. If you have to jump unconditionally between the functions, you have to use setjmp/longjmp functions.

3.6.4 Block Scope

Declarations that are active up to the end of the block (where a block is defined as statements within { }). All the declarations inside the function have only block scope.

```
int fun(int a, int b)
{
int c;
{
     int d;
}
// a, b, c, d all have block scope
}
```

As I have said, function scope applies only to labels. So should not be confused with block scope. The function arguments are treated as if they were declared at the beginning of the block with other variables (remember that the function body is also considered as a block within { }). So the function arguments have block scope (not function scope).

Local scope is general usage to refer the scope that is either function or block scope.

3.6.5 Prototype Scope

They are having the scope only inside the prototype declaration. This scope is interesting because the variable names are valid only in the declaration of prototype and

does not conflict with other variable names. It exists for very little time and of less use and so goes unnoticed.

e. g `int add(int a, float b);`

Here the variables a and b are said to have prototype scope.

```
int big(int p, int q);
// p, q have prototype scope. big has global scope.
int x=10, y=20;
// x, y have file scope

int big(int a, int b)
{
int temp;
// temp, a, b have block scope
if(a < b)
     goto ReturnB;
return a;
ReturnB:
// ReturnB has function scope
     return b;
}
```

3.6.6 Selecting Minimal Scope

When a name has to be resolved, that name is searched in the minimal scope, and if that is not available, it is searched at higher levels of scope. So, if a variable has to be declared, you have to select a minimal scope possible. If you can limit your scope, that increases the efficiency, readability and maintainability of your program. If you want a variable which is not useful outside a block, declare it inside the block and not in the outer ones. Similarly, if you want a variable whose value will be accessed only within the function but has to retain the value between the function calls, opt for static variable to a global one.

Exercise 3.1

What is lexical scope and is it related with the scoping discussed here?

3.7 Life Time

It is the period of time the object allocated space 'lives'. i.e. it is the time in which the variable is allocated space (also called as extent).

```
int h = 10;
//  global. static lifetime
int foo()
{
 int i;
 // automatic lifetime
 int * j = malloc (20);
```

```
 free(j);
 // dynamic lifetime
   {
     int k = 10;
     // automatic lifetime
     j = &k;
   }
 printf("%d", *j);
 // prints 10. Lifetime of k is of that of function foo
}
```

3.7.1 Static Lifetime

The object lives (i.e. storage remains) up to termination of the program as in global variables.

```
int foo(int i)
{
static int j = foo(i);
return j+1;
}
```

when happens when called with foo(1)? Returns value 2 to the calling function. (It does not go into infinite loop!)

For the first time the function foo is called with argument as 1, the control sees there is a static variable j that is to be initialized. For that, it calls foo again with i value

1. Now for the second time the control enters the function foo since static variable is called already for initialization, it executes return statement. All static variables by default are initialized to 0 so it returns 0+1(=1) as the value of j. The function return backs to initialize the j to 1. Finally, foo returns to the calling function with value 2.

3.7.2 Automatic Lifetime

The object lives (i.e. storage remains) up to termination of the function in which it is defined as in the case of local variables.

3.7.3 Dynamic Lifetime

The life is created (storage allocated) and exited (storage deleted) by the whims of the user as in the dynamically allocated memory. Although dynamic lifetime/dynamic memory allocation is treated here, facilities/functions (here, say, malloc function) provided in standard header files are not considered as part of the language. Even if dynamic lifetime makes sense that is why some authors do not consider dynamic lifetime as a type of lifetime in C.

3.8 Visibility

Concept of visibility comes only because of overloading of names (refer to namespaces). An object is said to be visible at a point if it is accessible through program code. An object association may be 'hidden' from other object associations due to name spaces.

e.g. {

```
    float i;
    // i is associated with float type
      {
         int i;
         // i is associated with int type
      }
  }
```

Here name i with associated with float object is hidden inside the block due to overloading of name i with int object association.

3.8.1 Visibility Vs Scope

Visibility must not be confused with scope. Although both concepts are related, they are distinct:

```
    {
    float i;
    // scope of float i begins,with block scope;
      {
      int i;
      // i is associated with int type and the
      // visibility of float i is lost, block still remains
     }
  }
// scope of float i ends.
```

To put it simply visibility is the accessibility of the variable, while scope is the availability if the variable. In the above example float i is still available inside the inner block but it cannot be accessed.

Point to Ponder:

Visibility cannot exceed scope, but scope can exceed visibility.

3.8.2 Scope Vs Lifetime

Similarly, scope must not be confused with lifetime:

```
int foo()
{
 int * iPtr;
 {
     int i = 10;
     iPtr = &i;
   }
 printf("%d", *iPtr);
// prints
// 10
}
```

C is a block-structured language. When the function returns to the calling function, the stack-frame gets erased and so all the local data for that function is

destroyed. However, it has to be remembered that the space is not erased as the block is exited although the scope of that variable is over (hence block scope).

In this example, ‘i’ has block-scope. Lifetime of i does not end with the exit of block. In other words, the space for ‘i’ remains allocated even after the ‘}’ of the block is reached. So this example serves as to differentiate between scope and lifetime.

Note: *Although this code works without any problem, the following one may not:*

```
int foo()
{
 int * iPtr;
 {
     int i = 10;
     iPtr = &i;
 }
 {
     int k = 20;
 }
printf(“%d”, *iPtr);
// may print
// 10 or 20
// in my system it printed 20
}
```

This is because the some optimizing compilers may reallocate the memory of variable ‘i’ to ‘k’ because the block is exited. Due to such reallocations while optimizations, this code may not work as expected.

Point to Ponder:

A declaration introduces a variable to a scope (not visibility or lifetime).

3.9 Name Spaces

If variable names are available afresh to use and independent of other associations they are called as name spaces. To term in other words, name-space is the scope within which an identifier must be unique.

In C namespaces are not sophisticated. New namespaces are created in following cases,

1. Tag names create a namespace.

```
struct name;
union name;
enum name { int a;};
```

creates three separate namespaces so is valid.

2. Members of structures / unions create a name space.

```
int name;
struct name{ int name};
```

3. Labels create separate namespace.

```
name :
    int name;
    int foo(int foo)
    {
    foo:
         printf("%d",foo);
    }
```

is a legal code that shows how the namespaces can be and still make sense.

Since compiler can keep track of the list of different variables in separate lists, there exists no ambiguity in differentiating between the usage of the variables. This is called as overloading of namespaces.

Overloading of names is allowed when name spaces are different.

```
struct name {int name ;} name;
```

is valid because name is overloaded in three different name spaces. We say name is overloaded and the meaning is resolved according to the context.

```
struct name d1;
d1.name = 10;
name.name = 10;
```

in all three cases the usage makes clear distinction for the compiler of which name is which and hence said that they lie in different namespaces. In other words, it should be clearly resolvable to the compiler by context without ambiguity. If any ambiguity arises, an error is flagged.

```
typedef struct dummy {int dummy ;} dummy;
dummy dummy;
// error. ambiguity. which dummy is which?
```

error is flagged here because the tagnames created by typedefs share same namespace as variable names.

Point to Ponder:

Duplicate names are legal for different namespaces irrespective of scope.

3.10 Linkage

It is creating the corresponding link between the actual declaration and the corresponding object, and the linker does this process.

3.10.1 No Linkage

No linkage indicates that the name is resolved at context itself and no linkage is necessary to be done by the compiler. The linker has no job working on this so said to have no linkage. An example for is local variables.

```
{
      // i is inside the block and is resolved inside the
      // block itself and so no linkage is required
int i;
for(i=0; i<10; i++)
      a[i]=10;
```

```
}
```

3.10.2 Internal Linkage

It specifies that the name that should be linked is within the current translation unit (say current file). So it simplifies the job because it knows that the definition is available in that file itself and so does not have to look in other files.

```
int i;
main()
{
printf("%d",i);
}
```

here the variable name i is of file scope, so is available only in the current translation unit. So linker searches and links with the name only within the translation unit.

3.10.3 External Linkage

This indicates that the definition available for the corresponding declaration(s) can be available anywhere outside the current translation unit. It as the keyword extern indicates that it is external to the compiler and the current translation unit, and so is beyond the control of compiler.

```
#include "somefile.h"
extern int out( );
int main()
{
```

```
out();
}
```

The linker has to resolve the name ‘out’ that is not defined in the current translation unit. So it is said to have external linkage.

***Exercise 3.2*:**

What type of linkage do you think function arguments have?

***Exercise 3.3*:**

What are static, global and dynamic linkages and early and late binding in programming languages point of view?

3.11 Storage Class

Variables can be put into some combination of scope, lifetime and visibility by using some keywords and placing them appropriately and this is called as storage class of the variable.

Broadly speaking there are only two storage classes. Auto and static. Other storage classes are extern and register.

3.11.1 Static

Depending on the context where the static declaration is used it can give two different meanings.

1. If declared at global scope it means that the variable has file scope.

E.g.1

```
static myprintf( );   // only file scope
```

declares that myprintf function is having file scope and is not accessible to other files.

E.g. 2.

```
int a;                // global scope
static int b;              // file scope
```

2. If declared within a block it means that the variable is having static lifetime but with local scope.

As a whole, the static keyword helps in limiting the variables to file scope such that collision of names at time of linkage is avoided. In other words, static limits variables to internal linkage.

A static variable (with local or block scope) can be initialized with the address of any external or static item, but not with the address of another auto item. Because the address of an auto item is not a constant since auto items are created in stacks and erased as the function exits.

3.11.2 Extern

'extern' is used for two main purposes; it can be used for:

- for specifying global linkage and
- forward declarations,

The first use of extern is to explicitly specify that the variable is a candidate for external linkage.

extern int foo();

specifies that the function foo is defined somewhere else and that will be known only at link-time.

For declaring a forward declaration for a structure, declare only the name, say, struct some; and later you can give the body. But this is not possible if you want to forward declare a variable. For this extern comes handy:

```
extern int i;
      // declare i is defined somewhere else
...
int i = 10;
      // i resolves to this variable i.
      // This is resolved by the linker and the programmer may have
      // used it for forward reference.
```

3.11.3 Auto

Auto keyword specifies that the variable is of automatic lifetime. It is not instructive to compiler but for you to just remember that it is of automatic type. Practically it serves no purpose (of course you know that the variables declared inside functions are automatic and telling explicitly that the variable is auto doesn't help you anyway).

3.11.4 Register

You can specify a local variable as register if you think that variable may be accessed frequently. The compiler will try to put that variable in a microprocessor register if one is available. Else this keyword is ignored and treated as if it is declared as

auto. Declaring such frequently used variables to be placed in registers may only lead to small performance improvement. Modern compilers will easily find out the variables that will be frequently accessed and will place them accordingly. This is a deprecated feature (so will be omitted in future versions of C) and so is not recommended to use.

It is illegal to apply '&' operator to a register variable. Why because, both the pointer that stores the address of the variable and the value in that address have to be in the memory for applying & and * operators. Then only these operations remain meaningful and valid. However, register variables may not be in active memory when the & operator applied and so C prohibits applying this operator on register variables.

Register variables can be used to improve the program efficiency. Looping variables are good candidates for such register optimizations,

```
register int i;
for( i=0 ; i < INT_MAX; i++)
     arrayAccess[i];
```

In such cases, the efficiency of execution certainly increases. If a memory register is not available for placing it in memory it may at-least place the variable in cache memory that is still an improvement in execution speed.

Anyway, using register variables is found to speed up many programs by a factor of two.

Exercise 3.4:

After reading this paragraph, can you reason it out why the register variables cannot be global/static?

Exercise 3.5:

Can local, register variables be declared extern?

3.11.5 Omission of Storage Classes

If variables of local scope are not given, a storage auto class is assumed. For variables outside functions, extern is assumed (not static). This applies to undeclared functions also. Default storage class for array of char is static.

It is not possible to have more than one storage class specifier in a declaration.

```
typedef static int i ;
//flags error. No two storage classes can be combined.
```

3.11.6 Is typedef a storage class?

Initially this question may not make any sense at all to you because all of us know that typedef is concerned with types and it has nothing to do with storage classes. First lets see the answer,

Yes. typedef is also considered as a storage class specifier. As the grammar for C specifies,

```
storage-class-specifier : one of
     auto register static extern typedef
```

typedef is a storage class.

As [Kernighan and Ritchie 1988] says, this classification of typedef under storage class specifier is purely for syntactic convenience. Here syntactic convenience means,

- to list typedef with storage class specifiers makes grammar simple; or else, another production is needed to be added to the grammar.
- It makes easy to check the rules like, *no two storage class specifiers can occur in a declaration*.

```
static extern int statExtern;
// error. static and extern cannot occur together in a declaration
```

the same rule applies to typedef,

```
typedef static int statInt;
// doesn't make any sense. how can a type name cannot have storage
// class specifier
```

Moreover, discussion of this question helps in understanding that grammar is the final authority for answering such questions because it is the one that describes the language itself. There are few instances where only going through grammar of language give you final answer and that cannot fail. Another such example is complex declarations. If you have any argument in finding the meaning of a complex declaration, refer grammar. It always gives the correct and final answer.

3.12 Complex Declarations

C is infamous for its complicated declaration syntax where the beginner will certainly stumble. Complex declarations are hard to decode and understand. However, it

is an essential area to master because you may require it in real programming applications.

There is a simple traditional technique available called as ‘clockwise’ rule [Binsock 1987].*“Take any declaration, start with the innermost parenthesis (in the absence of parenthesis, start with the declared item’s name), and work clockwise through the declaration going to the right first“*

Consider,

```
void (*x[10]) ( );
```

As the rule says start from the innermost parenthesis, take only (*x[10]) now. In that part start with the variable name, here it is x. Work clockwise and read it. x is an array[10] of pointer. Then continue reading remaining parts. ... to function returning type void. Reading it fully, it comes out that; x is an array[10] of pointer to function returning type void.

This rule works for all declarations of `[Kernighan and Ritchie 1978]` in the absence of qualifiers like const and volatile.

However, when these qualifiers (const & volatile) are involved there is no ‘shortcut’ for reading the declaration. Even though short cuts can help for some extent, its better to rely on grammar to help (which is the formal way and can never fail). The productions associated with this problem of decoding the complex declarations are,

```
declarator :
     *’s (opt) direct-declarator
direct-declarator :
     identifier
```

```
(declarator)
direct-declarator [ constant-expression(opt) ]
direct-declarator (identifier-list(opt))
```

Listed here are two productions for non-terminals declarator and direct-declarator. Non-terminals mean that they do not make part of the result but they are tools for getting the result. In the productions, the other non-terminals are constant-expression and parameter-type-list (which mean that they in turn will have other productions and they will be in the LHS). (opt) specifies that the particular part is optional.

To parse any complex declarations, just start with the non-terminal declarator and replace that non-terminal with the RHS of the production. In a production, if there are more than one alternative to choose from RHS, (as in the case of direct-declarator) select the best match that fits your input. Go on replacing the non-terminals until all the non-terminals are replaced with terminals, and the process of replacing is the required result for us.

3.13 Initialization

Initialization is setting the values in the object at the time when the objects are allocated space.

Initialization may be explicit or implicit.

3.13.1 Global Variables

Global variables are initialized implicitly. They are initialized to zero (since the allocation is made at the code segment)

```
// global scope
float f;
// f = 0.0
int i;
// i=0
struct  {int a, float b} b;
// b.a=0, b.b=0.0
union  {int a, float b} b;
// b.b=0.0 i.e. all the spaces are zero
double * i;
// i = NULL
int a[3];
// a[0] = 0, a[1] = 0, a[2] = 0
```

The automatic initialization is assured by the standard and so initializing them again like,

```
int i=0;
```

is redundant and I don't recommend to use like this.

The same rules apply for static variables as for the global variables (for example, the initialization cannot involve expressions).

3.13.2 Local Variables

Local variables on the other hand are not initialized automatically since they are allocated space in stack frames. Therefore, care should be taken to initialize them manually. Unlike global variables, they can be initialized with expressions.

```
int i = 10 , j = i;
```

this will issue an error if this declaration is in global scope but no problem if this is in block scope. This is possible because the creation and so initialization of the local variables are done at runtime in stack frames.

3.13.3 Arrays

```
for (i = 0 ; i < 100 ; i++)
     arr[i] = 0;
```

This method of traversing the array and initializing it is not efficient.

```
int arr[100] = {0};
```

gives the curt way to initialize all elements to zero. This technique for initialization is possible at the point of declaration only. In those cases memset() comes handy.

```
void *memset(void *str, int ch, size_t n);
```

It sets n bytes of str to character ch. The difference is that the initialization with {0} is done at compile time and memory set by memset is at runtime. So it is efficient and reliable to use the former method whenever possible.

Arrays can be initialized with values given within curly braces. If the initialization list contains members less than the available members do, the remaining members are initialized to zero.

It is an error to initialize an array with more than the possible elements. One exception for this rule is character arrays (because the objective may be to have not a string, but a sequence of characters),

```
char vowels[5]="aeiou";
// is valid. No space for '\0', so not NULL terminated
```

The initialization list may contain constant expressions.

Interestingly enough the [Kernighan and Ritchie 1988] did not have option for array initialization inside the functions. The reason was that the stack frames are allocated at runtime and if the arrays have to be initialized, the code has to be generated for that.

```
int globalArr[3] = {1,2,3};
// no code is generated and gets initialized in
// initialized data segment
int fun()
{
int localArr[3] = {1,2,3};
// o.k. (ANSI C); code is generated for initializing 'arr'
// and gets initialized when stack frame is created.
Static int staticArr[3] = {1,2,3};
// similarly, no code is generated
}
```

The code generated is bulky; that's why the initial [Kernighan and Ritchie 1988] did not have that feature. Later [ANSI C 1988] added this feature although it results in increased code size.

3.13.4 Difference Between Initialization and Assignment

It is good to remember that in case of local variable declarations, the compiler may postpone allocating memory until the variable is referenced once. This gives rise to the concept of early and late initialization. Practically in C late initializations can be treated as assignments.

```
int i = 10, j ;
// early initialization
...
j = 10;
// late initialization (here same as assignment)
```

Here when defining itself i is initialized to 10. So it is said to be have early initialization. However, in the case of j, we initialize it after sometime, and we call it as late initialization.

Depending on the situation, you can follow either early or late initialization. However, it is easy to declare a variable and forget to initialize it before using it. So it is recommended to use early initialization whenever possible.

When a value is stored into a memory location as soon as its is created, it is initialization. Assignment is storing a value after the memory location is created. This is the primary difference. Another difference is that, the target of initialization is always an un-initialized location whereas for assignment it can be either initialized or un-initialized.

In case of static and global variables, the difference is that the initialization values are stored in the areas like initialized data segment (as you have seen in the structure of a C program in memory). The compiler does this work, whereas code is generated for the

assignment statements. This makes little efficiency difference between initialization and assignment.

```
int j;
// global variable and allocated in BSS
{
static int i = 10;
// i is allocated space and stored initialized data segment
...
j = 10;
// code is generated to store value 10 in j
}
```

Except these minute differences there is no big difference between initialization and assignment in most of the cases and can be used interchangeably. An example for this is the negligible difference between storing the value 10 into i and j we saw now.

Lets see few cases where only initialization can be done and not the assignment.

1. In the case of initializing the constant variables,

```
const int i = 10;
// and
const int i;
i = 10;
// error. So both are not equivalent
```

Since the read only variables can only be initialized and the assignment happens to violate the read only nature, the difference is evident.

2. In the case of initialization of string constants to character arrays,

```
char str[] = "string";
// and
char str[];
str =  "string";
// error. So both are not equivalent
```

3. The case when the structure/union contains a const member. Consider the code:

```
struct temp{
const int i;
};
struct temp s = {10},t;
t = s;
// error: l-value specifies a constant object
t.i = 21;
// error: l-value specifies a constant object
```

3.13.5 Function declaration Vs Function calls

The compiler can easily find the difference between the function declaration and a call if the statement contains any of the keywords like int, static etc.

```
static int someFun(char );
```

is clearly a function declaration because the function call can only contain the variable names/ constants and cannot involve with such keywords. Similarly,

```
clrscr();
int main()
```

```
{
// something here
}
```

in this case the clrscr() is the declaration of the function that has implicit assumption of the int type as return type and that the function can take any number of arguments.

When it is not clear if the function declaration or a function call, the compiler gives the benefit of doubt to the function call and thus resolves to function call as in:

```
int main()
{
clrscr();
}
```

In this case, as you can see, 1) this can be a function declaration, 2) it is a function call treating it as a statement. Therefore, the compiler resolves it to a function call.

4 OPERATORS, EXPRESSIONS and CONTROL FLOW

Of all the things you wear,

your expression is the most important

- Janet Lane

Expressions are the fundamental means of manipulating the value of a data. An expression is a set of data objects and operators that can be evaluated. Any expression in C can be made into a statement by appending a semicolon with it.

4.1 Lvalues and Rvalues

An object is a region of memory that can be examined and stored into. A *lvalue* is an expression that refers to an object in such a way that the object may be altered as well as examined. Traditionally lvalues are variables that appear on the LHS of = operator. They can appear both as LHS or RHS part of an expression.

Example: names of variables declared to be of arithmetic, pointer and enumeration types.

Some operations on non-lvalues can produce lvalues.

However, not every lvalue may be used for assignment. They are *non-modifiable* lvalues. A function name is an lvalue, however you cannot modify it, so is a non-

modifiable lvalue. Another example is an array name, which is a pointer constant and cannot be modified.

An expression that permits examination but not alteration of an object/value is called the *rvalue*. It can be used on the RHS of an expression. Sometimes *rvalue* is the value stored in address of data object.

Example: named constants, names of arrays and enumeration constants.

Try the expression like ++a++, and you will get a compiler error. This is because, it is treated as ++(a++) (since the postfix ++ takes precedence to the prefix ++). a++ is a rvalue and prefix ++ tries to operate on it resulting in an error.

Note: This example assumes that 'a' is just a lvalue and it is not of any pointer type. Consider the code:

```
char *p= "C always have someway around",*p1;
p1=p;
while(*p!='\0')
      ++*p++;
```

Here the expression ++*p++ is perfectly valid. (*p++) yields an lvalue and prefix ++ operates on that lvalue so is valid.

Point to Ponder:

All lvalues are rvalues but the converse doesn't hold.

Exercise 4.1:

Comment on the following code:

```
struct constStruct{
     int i;
     const int j;
     // note const here
   };


int main()
{
 struct temp temp2;
 temp1.i = 10;
 temp2.j = 10;
}
```

In general, do you agree with the statements *"for structures and unions to be modifiable l-values, they must not have any members with the const attribute*" and "*operations on the structures a non-lvalue may yield an l-value*"?

Exercise 4.2:

Both the non-modifiable lvalues and rvalues don't allow assignment to be done on them; then what is the difference between them?

4.2 Difference between operators and punctuators

Unlike operators, punctuators do not specify an operation to be performed, they have special meanings in C source text and helps the compiler to understand the program. For example : (colon) is a punctuator (and not an operator) that comes after labels and case statements in switch, that helps the compiler in understanding the intended meaning of the program. Certain operators also work as punctuators depending on the context. An example is () that helps to group the expressions. Here it acts as punctuator. It acts as operator in function call. Another example is , (comma) operator that is used in expressions and when used in for loops or function declarations work as punctuator (sometimes called as separator)

punctuator : one of

[ ] ( ) { } * , : = ; ... #

4.3 'Arity' of Operators

The number of operands an operator takes is known as 'arity' of operators. +,-, * and & are the operators that have be used both as unary and binary operators. ?: is the only operator that takes three operands (thats why it is 'ternary' operator).

Exercise 4.3:

What is the 'arity' of , (comma) operator?

4.4 Unary Expressions

It is of operators that take only one operand.

- Names and Literals

The value of name depends on its data type, which is determined by the declaration of that name. The name itself is considered as an expression and is an lvalue. A literal is a numeric constant, when executed as an expression yields that constant as its value.

- Parenthesized and Subscript Expressions

(), [], . (dot) , -> operators come in the top of the operator precedence hierarchy.

A parenthesized expression consists of an open parenthesis, an expression followed by a closing parenthesis. The value of this is the value of the enclosed expression, and will be an lvalue if and only if the enclosed expression is an lvalue. The presence of parenthesis does not affect whether the expression is an lvalue or not.

- Component Selection Expressions

Component selection operators are used to access fields of the structure and union types. They are . (dot) and -> operators.

```
struct student{
    int rollNo;
    char name[10];
    }*studentPtr, student;
 student . rollNo = 10;
 // dot operator  to access the member
```

```
(&student) -> rollNo =10 ;
      // both are equivalent
```

Since . (dot) operator is having higher precedence over * (indirection) operator a parenthesis becomes necessary as in (*something).member. But this requires clumsy syntax of using parenthesis every time to dereference the member. To avoid this, -> operator was introduced. something->member is the shortcut for (*something).member. In any case, (*). and -> both are equivalent.

```
studentPtr -> rollNo = 10;
// using arrow operator  to access the member
(*studentPtr) . rollNo = 10;
// both are equivalent; parenthesis is necessary here because * is
// having lesser priority than . (dot)
```

- Function Calls

A function call consists of a postfix expression (the function expression).

- Post Increment/ Decrement Expressions

The postfix operator ++/-- performs these operations respectively, a side effect producing operation. The operand must be an lvalue and may be of any of integral type. The constant 1 is added/subtracted to the operand, modifying it. The result is the *old* value of the operand before it was incremented/decremented. This value is not an lvalue.

- Type Casting

A cast expression causes the operand value to be converted to the type named within the parenthesis. The result is not an lvalue. Some implementations in C ignore certain casts whose only effect is to make a value narrower than normal.

- Unary Minus/Plus

The unary operator - computes the arithmetic negation of its operand. The operand may be any of the arithmetic type. The usual unary conversions are performed on the operand. The result in not an lvalue.

I.e. `-x = 1 – x`

```
// error. Lvalue required
```

Unary plus operator is introduced in ANSI C for symmetry with unary minus. It is the only dummy operator in C (dummy operator means neither it will have effect in the program nor the compiler will generate any code). This is due to the multiplicative property of signed numbers in mathematics.

```
int negative = -1 , positive = 1;
printf("%d..%d", -negative, -positive);
// negates. prints 1..-1
printf("%d..%d", +negative, +positive);
// no effect. prints -1..1
```

There is an interesting question to ask about constants. Is –10000 the negative integer constant or a constant expression (an expression containing – (unary minus) applied to 10000)?

For finding the answer lets experiment with some sample values.

```
// in my system sizeof(int) == 2
// note: range of 2 byte int is –32768 to +32767
```

printf(“%d”, sizeof(32767));

```
// prints 2. Since 32767 is in positive range of int,
// it can be stored in an int.
```

printf(“%d”, sizeof(32768));

```
// prints 4. Since 32768 is beyond the positive range of int,
// it can’t be stored in an int. It requires long so it prints 4.
```

The similar logic can be applied to the negative values also. We don’t know if negative values are treated as integer constants or constant expressions:

printf(“%d”, sizeof(-32767));

```
// prints 2. Both -32767 and +32767 are in valid range of int,
// so we cannot find out if it is a negative integer constant or
// a constant expression
```

printf(“%d”, sizeof(-32768));

```
// prints 4. If –32768 were treated as negative integer constant it
// should have printed 2, because it is in valid integer range.
// +32768 cannot be represented so promoted to integer and – operator
// is applied on it. So the only possibility is that this is a
// constant expression.
```

So the result of from this experiment is that the – (minus) sign in the constants makes it a constant expression.

If you want to take advantage that the value -32768 be represented in integer itself, because it is in valid integer range, use explicit casting:

printf(“%d”, sizeof((int)-32768));

```
// prints 2. No loss of information since this value can be
// represented in an 2 byte integer.
```

- Negation

The unary operator ! computes the logical negation of its operand. The operand may be any of the scalar type. The result of the operator is of type int.

The unary operator ~ computes the bitwise negation of its operand. The operand may be any of the integral type. Every bit in the binary representation result is the inverse of what it was in the operand. The result is not an lvalue.

The following example shows the relationship between the – (unary minus) and ~ (bit-wise negation) operators. These relations are due to the nature of two's complement arithmetic.

```
int i =10;
if( (-i == ~i+1)  && (-i == ~(i-1) )
     printf("This is always get printed.”);
// This shows the equivalence between these operators
// because of the simple reason that ~ produces ones’ complement
// in twos’ complement machines
```

Another relation is between the ~ and ^ (ex-or) operators:

```
~a = -1 ^ a
```

(the reason being that -1 is represented as all 1’s and ex-or-ing it with the variable is same as ones compliment)

- Addressof

The unary operator & returns a pointer to its operand, which must be a lvalue. None of the usual conversions are relevant to the & operator and its result is never an lvalue.

- Indirection

The unary operator performs indirection operation through a pointer. The * operator is the inverse of & and vice-versa. The (converted) operand must be a pointer and the result is an lvalue referring to the object which the operand points.

```
int i;
j = *&*&i;
// assigns i to j
```

Here & operator creates a reference and * dereferences it. This sequence can be applied any number of times and can be assigned until the types of LHS and RHS agree.

The run-time effect of applying indirection operator to a null pointer is undefined.

- Pre Increment/Decrement

The unary operator ++/-- performs these operations respectively, a side effect producing operation. The operand must be an lvalue and may be of any of integral type (That is why expressions such as ++2 are not valid). The constant 1 is added/subtracted to the operand and the result stored back in the lvalue. The result is the *new* value of the operand but is not an lvalue.

Any number of unary operators may operate on an operand provided it is legal.

```
int i = 0;            // say int takes 2 bytes

i = ! ~ - sizeof(i) ;

printf("%d", i);      // prints 0
```

The unary operators evaluate from right to left. In this case, sizeof(i) => 2. Then applying - to it makes -2. ~(-2) is 1. ! 1 is 0. Therefore, 0 gets printed.

4.4.1 Sizeof

The sizeof operator is used to obtain the size of a type or data object. Sizeof operator is a special operator in C, because,

C language is the language in which the size of a data type may differ according to the underlying machine and to tackle this problem in a portable way this operator is required.

- it is the only operator evaluated entirely at compile time.
- it also is a keyword

It takes two forms:

- sizeof followed by parenthesized type name and
- sizeof followed by an operand expression.

The type name should not be,

- function type
- name of array type of no explicit length (incomplete type)
- type void

```
int i = sizeof(int);
```

```
// o.k. paranthesized type name
int j = sizeof int;
// error : typenames used in sizeof should be paranthesized
```

Similarly,

```
int fun()
{
return 0;
}
int main()
{
printf("%d", sizeof(foo));
// error.
printf("%d", sizeof(foo());
// o.k. because it becomes an expression.
// when function calls are involved, its type is its return type.
}
```

The result of applying sizeof operator to an expression can always be deduced at compile time by examining the type of the objects in the operand, thus the expression itself is not compiled into executable code. In other words the expression within the sizeof operator are not evaluated but parsed for finding its type. Hence, any side effects that might be produced by the execution will not take place. For e.g.

```
k = sizeof( j++);
```

will assign some value to k but will not increment j.

The operands that cannot be used in this operator are function type, incomplete type and bit-fields.

When array name is applied to sizeof operator then the result is the size of the array. The return type of sizeof is size_t that is defined in <stddef.h>.

When the string constant is given as parameter, it returns the number of characters in that string.

```
sizeof("abcd");
// prints 5 not 4 because '\0' character is included
```

For pointer to an array, it is size of the pointer type.

A macro to find the dimension of an array can be defined as follows,

```
#define DIM( array, type) sizeof(array)/sizeof(type)
int arr[10];
printf("The dimension of the array is %d", DIM(arr, int));
// prints 10
```

sizeof can take part in constant expressions. This is because it is evaluated and replaced at compile time.

Exercise 4.4:

Comment on the output of the following code:

Output was: size of arr[] = 2 and it has 1 element(s).

```
int size(int arr[]){
```

```
printf("size of arr[]  = %d and ", sizeof(arr);
     return(sizeof( arr ) / sizeof( int ));
}
main(){
 int arr[10];
 printf("it has %d element(s)", size( arr ));
}
```

Exercise 4.5:

What is the output for the following code segment?

```
void fun();
printf("%d", sizeof(foo());
```

4.5 Binary Expressions

These are expression involving two operands.

- Multiplicative

The multiplicative operators are *, /, %. The operands are of arithmetic type for * and / while integral type for %. / computes quotient, whereas % computes remainder.

The basic rule to follow while predicting the result is

Condition 1:

```
a = (a/b) * b + (a%b)
```

Condition 2:

```
abs( a % b) < abs(b)
```

No confusion arises if both the operands are positive. If any of the operands are negative, it is assured that condition 2 only holds.

C follows 'Eulerian' arithmetic. To avoid ambiguity, always use unsigned or positive integers as operands.

The following is the relation between % and & operators.

```
a % b == a & (b-1)
```

Interestingly enough, % may also yield divide by zero error!

- Additive

Additive operators are + and -. Operands of + should either both be arithmetic or one is arithmetic and another is pointer. No other operands are allowed. Both the operands may be of type pointers or arithmetic type for subtraction. The following is a famous example for swapping two values without using any temporary variables:

```
void swap(int *i, int *j)
{
*i = *i + *j;
*j = *i – *j;
*i = *i – *j;
}
```

However, such examples of swapping the two variables serve only for aesthetic purpose only not practically.

```
#define SIZE 5
```

```
int i;
int arr[SIZE] = { 1,2,3,4,5};
for(i=0; i<SIZE; i++)
     swap(&arr[i], &arr[SIZE-i-1]);
for(i=0; i<SIZE; i++)
     printf("%d ",arr[i]);
// prints 1,2,0,4,5
```

The middle element arr[2] contains 0 now. What happened?

The problem is not with the array; it is with the swapping function. When elements to be swapped happens to refer to the same location, the swapping fails. You can verify this fact by the following example,

```
int i = 1;
swap(&i, &i);
printf("%d", i);
// prints 0 !!
```

Note that this problem would have gone unnoticed with testing done only on arrays containing even number of elements as in:

```
int arr[SIZE] = { 1,2,3,4,5,6};
```

The moral is that, proper testing should be made and most of the possible cases (if possible, all) have to be considered before resorting to any solution.

- Shift

The binary operator << indicates a left shift and >> indicates a right shift. They are left associative. The operands for these must be of integral type. The usual conversions are performed separately on each operand and the type of the result is that of the converted left operand.

The value of exp1 << exp2 is exp1 left-shifted exp2 bits; in absence overflow the equivalent is multiplied by exp2. In case of exp1 >> exp2 the value is exp1 right-shifted exp2 bits or is equivalent to dividing by exp2 if exp1 is unsigned or if it has a non-negative value. If exp1 is a signed number, the result depends if the implementation follows arithmetic or logical shift depending on compiler/hardware.

```
 int i = -1;
// say sizeof (int) == 2 so, -1 is represented as 1111 1111 1111 1111
 i >>= 1;
// shift i once by right. i becomes  ?111 1111 1111 1111
```

If the bit indicated by ? is filled with 1, the value of the sign bit, we say it is arithmetic shift. If, by default, it is filled with 0s in the vacated space, we call it as logical shift.

- Arithmetic Shift

The MSB is filled with the copy of sign bit, preserving the sign of the value.

- Logical Shift

The MSB is filled with zero thus modifying the sign of the value.

So, it is always safer to use unsigned short or int for right shift.

In >> the right operand should not be larger than the size of the object.

E.g. 1 >> 17

```
// unpredictable result if sizeof (int)==2


// Code segment to demonstrate the property of arithmetic
// and logical fill while using right shift operator
// and the property that characters may be signed or unsigned
{
char ch1=128;
unsigned char ch2=128;
printf("ch1 = %d, ch2 = %d\n",ch1,ch2);
ch1 >>= 1;
ch2 >>= 1;
if(ch1 == ch2)
     printf("Default   char   type   in   my   system   is
     unsigned");
else
     printf("Default char type in my system is signed
")
}
// in our system it printed
// ch1 = -128, ch2 = 128
// Default char type in my system is signed.
```

- Relational

The operators < <= > >= indicate comparison. Either the operands may be of arithmetic type or both may be of the same pointer type. The result is either 0 or 1. For pointer operands, the result depends on the relative location within the address space of the two objects pointed to; the result is portable only if the object pointed to lie within the same array.

Comparisons like i<j<k is valid in traditional mathematics. It also logically makes sense (the aim is to see if i is smaller than j and is in-turn smaller than k).

This expression is also a valid C expression but yields an unlikely result. For (i<j) result is either 1 or 0. This resulting value is compared with k which yields wrong result (except if k is not 0 or 1).

- Equality

The operators == and != indicate equality comparison. Either the operands may be both of arithmetic type or both may be of the same pointer type, or one of the operand may be a pointer and the other a constant integer expression with value 0. The result is always of the type of integer (either 0 or 1).

- Bitwise

These operators perform bitwise manipulation. The operands used must be of integral type. The result is that of the converted operands. The operators are &, ^ and |.

Consider the problem of clearing a particular bit in an integer.

```
int x=1;
```

```
x &= 0xFFFE;
// aims to reset the first bit
```

However, this suffers a portability problem. This setting assumes that your machine has the integer size of 2 bytes (hence it clears all the bits remaining)

So the solution is to rewrite code like this,

```
x &= ~0x1;
```

This does not assume anything about the internal int size and ports well.

- Logical

The operators && and | | are used for binary logical operations. They accept operands of any scalar type. There is no constraint between the types of the two operators. The type of result is always integer (either 0 or 1).

The Boolean expression involving these operators are executed only as far as it is required to determine the truth-value of the expression. It derives from the logical fact that there is no point in continuing evaluating the truth-value of the Boolean expression once it is determined and cannot change after that. So the remaining expression is abandoned. This also to improves efficiency because unnecessary computations are avoided. For example:

```
int i = 0;
if(++i==1 || i++==1)
      printf("i=%d",i);
      // prints i=1
i = 0;
```

```
if(++i==0 && i++==1)
      ;
else
      printf("i=%d",i);
      // prints i=1
```

Take the first case: ++i==1 is true, and the following operator is ||. (true || anything) is true. So the remaining expression will not be evaluated. So i++==1 is not executed and the result is evident by the output from printf statement. Similarly, i++ is false and the following operator is &&. Its truth-value is determined (false && anything) is false. The remaining expression is not evaluated and this is evident by the value of i in printf statement.

4.6 Ternary Operator

The conditional operator ? : is known as the ternary operator as it takes three operands and is the only operator in C that takes three operands.

If its operands are of different types, type-conversion takes place.

Conditional operator has right to left associativity. This can be shown with an example of a simple function to find biggest of given three numbers:

```
int biggestOfThree(int a, int b, int c)
{
return ( (a>b) ? (a>c) ? a : c : (b>c) ? b : c );
}
```

```
((a>b) ? ((a>c) ? a : c) : ((b>c) ? b : c))
-------------------------1-----------------
```

The numberings and parenthesis show which '?' is linked to which ':'.

Conditional operator can be placed on the left-hand side of the assignment operator, if the resulting variables are lvalues. So, expressions like,

```
((a>b) ? a : b) = 1;
```

is valid.

It is the shortcut for if-then-else statement with a single statement. As an example let us see how efficiently can it be used in implementing 'assert' macro.

The 'assert' macro is used to verify certain conditions in the program and report if the condition fails and so acts as a debugging tool. The advantage of using the assert is that defining NDEBUG will remove all the assert statements from the source code. Even after debugging is over the assert statements become a documentation for the conditions to be ascertained and becomes a helpful tool for maintenance

One possible implementation using conditional operator is as follows,

```
#ifndef  NDEBUG
     #define assert(cond) ((cond)?(0): (fprintf
     (stderr, "assertion failed: %s, file %s, line %d
     \n",#cond, __FILE__,__LINE__), abort()))
#else
     #define assert(cond)
#endif
```

Using conditional operator in the place of if has the advantage of being very curt and does not have the problem of 'if matching nearest else' is avoided. To elaborate, consider it were implemented using if then there is a chance that it will cause problems when assert macro is used in if-else statements.

```
#define assert(cond) if(!(cond)) \
     (fprintf(stderr, "assertion failed: %s, file %s,\
     line %d \n",#cond, __FILE__, __LINE__),abort())
```

and called with,

```
int i = 0;
if(i==0)
     assert(i < 100);
     // bug here in implementing assert
else
     printf("This part becomes else of if in assert");
// this printed
// "This part becomes else of if in assert"
```

The bug is in the if condition of assert macro. The else part in which the prinf statement is available becomes the else of the if condition of assert macro. When the condition of assert becomes true the else part is executed, leading to bug in the code.

Placing the if statement of assert macro inside a block is the usual method to avoid the 'nesting to nearest else' problem.

```
#define assert(cond) {   \
     if(!(cond))         \
```

```
(fprintf(stderr, "assertion failed: %s, file %s,\
line %d\n",#cond, __FILE__, __LINE__),abort()); \
}
```

But the caller calls the assert macro ending with an semi-colon. In macro expansion the combination }; becomes available leading to the error “illegal else without matching if”. So this problem of ‘nesting to nearest else’ cannot be solved by enclosing if statement inside a block. The best solution happens to be using the conditional statement as we saw in the first case.

Exercise 4.6:

The ternary operator(? :) is roughly an equivalent for if-then-else. Can you think of any reason why there is no operator like a single question mark(?) as a binary operator equivalent to if-then statement?

Exercise 4.7:

Explain how the expression is evaluated as:

```
int a = 50, b = 100;
int c = (a,b)?(b,a)?a:b:a;
```

4.7 Assignment expression

Assignment expression consists of two expressions separated by an assignment operator. Every assignment requires an lvalue as its left operand and modifies it by

storing new value into it. The result of this expression is never an lvalue. Assignment between all types is permitted except arrays.

- Simple

Given by =, which replaces the value of the expression by the object referred by the lvalue.

- Compound

The compound statements are of the form var1 op = exp where op is a valid binary operator. This is nothing but the shortcut for var = var op exp.

This may seem to be a trivial advantage. However, it serves two purposes,

1) It becomes handy when the exp becomes increasingly complex.

2) Efficiency. Most of the machines have machine instructions with immediate operands. These instructions increase directly specify the required information so help in faster execution. 'C' makes efficient use of this feature by providing compound statements for which translation can be done directly to its corresponding machine instruction. For example:

```
a=a+10;
```

may be converted to,

```
MOV AX,_a
ADD 10
MOV _a, AX
```

Whereas a+=10; may be converted directly to,

```
INC _a, 10
```

in some machine.

4.8 Sequential (Comma) Operator

Comma operator has the lowest precedence of all operators. Comma operator is one of the few operators to which the order of evaluation is specified. Both comma and conditional operators evaluate from left to right. The values of left expressions are discarded. The type and value of the entire expression is the type and value of the rightmost expression.

```
int i = (1, 2);
      // assigns 2 to i. ‘,’ operates from left to right.
      // value 1 is discarded.
int i = 1,2;
      // assigns 1 to i, since ‘=’ is having higher precedence than
      // ‘,’. 2 is evaluated but not used. It is as if
      // given as i = 1;   2;
```

The amazing power of comma operator can be realised by the fact that the semicolons at the end of statements by comma operator.

```
void interChange(int *a, int *b)
{
a^=b; b^=a; a^=b;
// will do the job!
// or
a^=b,  b^=a, a^=b;
// does it in a single statement!!
// or even
(a^=b),(b^=a),(a^=b);
```

```
// to show explicity how the expression is evaluated
// but not this
a ^= b ^= a ^= b;
// because side effects is involved due to the attempts to modify
// the variable twice between sequence points. Refer about sequence
// points and side-effects.
}
```

Some symbols can act as both a separator and an operator. Comma operator is one such operator. In the following code segment,

```
int i, j;
for( i=0, j=0 ; i < 10 ; i++)
     add(1,2);
```

in all these three cases comma acts as a separator.

```
int j = (0,1);
for( ; i, j ; j--)
     add( ( 1,2) , 3);
```

in all these 3 cases comma acts as an operator. (1,2) evaluates to 2 and passed as a first argument and 3 as the second argument.

4.9 Significance of Equality of Operators

Equivalence of operators is helpful in reducing the strength of the operations. Reducing the strength of the operations means replacing the operators with equivalent, but more efficient ones. For example:

```
iexp * 8
```

can be replaced by,

```
iexp << 3
```

Because multiplication by 2 is equivalent to left-shift by 2 (8 is multiple of 2 and so is equal to left-shift by 3). Left-shift requires less microprocessor resources than multiplication, increasing the efficiency.

These optimizations are done by the compiler optimizations without your knowledge. However, you can make this explicit if execution efficiency a top concern for you.

4.10 Operator Precedence

When two or more operators are mixed with each other, precedence determines which operands are evaluated first. It means that an operators' precedence is meaningful only if other operators with higher or lower precedence are present. The precedence of conventional operators is easy to understand because they follow traditional mathematics. C is rich in operators, so the precedence may not be evident in some cases.

Consider the expression,

```
something = a << 4 + b;
```

The programmer intended to calculate (a << 4) and add b to the result. But it is interpreted as:

```
something = a << (4 + b);
```

Another such instance is the precedence of operators &&, || vs. ==:

```
(a && b == c && d)
```

== is having higher precedence than &&. So it is interpreted as:

```
( (a && (b == c) && d)
```

This is counter-intuitive. History gives the reason for that.

There where no separate operators for the bit-wise and logical operators (&,&&,| and ||) when C was originally developed. The & and | operators served a dual role and had the precedence level as it is now. It used the traditional notion of finding the meaning based on context and usage ("truth value context").

To separate the concepts of bit-wise and logical operations, these two new operators (&& and || operators) were added. But the problems persisted with precedence levels. For example the conditions like the following one were still a problem:

```
if (a==b & c==d)
```

"In retrospect it would have been better to go ahead and change the precedence of `&` to higher than `==`, but it seemed safer just to split `&` and `&&` without moving `&` past an existing operator"`[Ritchie1982]`.

( ) [] -> .
++ -- ~ ! + - * & (type) sizeof
* / %
+ -
>> <<
> >= < <=
= = !=

&
^
|
&&
||
?:
= op=
,

The operator precedence table

Note : Here operators are listed in descending order of precedence. Several operators appearing on the same line or in a group have equal precedence.

Another example for confusion with operator precedence is given here:

```
a > 0 ? a++ : a += 2
```

Here the programmer intends to increment the value of ‘a’ if it is greater than zero else increment ‘a’ by 2. Since ?: operator is having higher precedence than += operator, it is treated as :

```
( (a > 0) ? (a++) : a) += 2
```

which is makes compiler to issue an error. So if you are not clear of the operator precedence *use explicit paranthesis to group the expressions explicitly* as in:

```
( a > 0 ) ? ( a++ ) : ( a += 2 )
```

4.11 Associativity

Associativity determines how the operators group if they are at the same precedence level. Operators can be left-associative, right-associative or sometimes have no associativity at all[†]. For example:

```
a = b = c = d;
```

is equivalent to,

```
a = (b = (c = d))
```

because the = operator is right associative. All the assignment operators (like *=), conditional operator and most of the unary operators are right associative. All other operators are left associative.

4.12 Order of Evaluation

Order of evaluation is the sequence in which the operands are evaluated in an expression given that the operands are of same precedence (remember that order of evaluation is different from associativity and precedence).

```
int a = b + c + d;
```

it may be evaluated as (b + c) + d or as b + (c + d).

The reason for allowing compiler is free to evaluate expressions in its desired order is to improve the efficiency.

[†] For example, such case of no-associativity arises in case of FORTRAN where the relational operators cannot be combined together.

```
int i = i + j + i;
// can be replaced with
int i = 2 * i + j;
```

In general, the compiler is free to evaluate expressions in any order. It may even rearrange the expression as long as the rearrangement does not affect the final result, but with few constraints:

1) Rearrangement of arguments of a function call or two operands of a binary operator in some particular order except left-to-right.

2) The binary operators, * + & ^ |, are assumed to be completely commutative are associative and the compiler is free to exploit this assumption.

```
int a = a() + b() + c() ;
// this cannot be changed to
int a =  b() + a()+ c();
```

Similarly the following optimization cannot be done,

```
int i = i() + j() + i();
// cannot be replaced with
int i = 2 * i() + j();
```

because the function call i() would be done only once instead of two.

Note: *Although the operator precedence, associativity and order-of-evaluation are all closely related, they are distinct.*

4.13 Side Effects

As the name indicates, side effects are changes because of evaluation of a main computation.

An expression becomes a statement if it is followed by a semicolon and executed only for its side effect and the value returned is ignored.

```
int  timesCalled;
void someFunction(int x, int y)
{
printf("Sum = %d",x+y);
++timesCalled ;
}
```

Here timesCalled is a global variable and whenever you call someFunction, timesCalled will be incremented as a side result. The better way to achieve the same result by the following code,

```
int someFunction(int x, int y)
{
static int  timesCalled;
printf("Sum = %d", x+y);
return ++timesCalled;
}
```

In this function call the information that timesCalled is made explicit and that variable is also contained within the function, so it is better than the previous version, and also makes use of the valuable information of return value.

So, side effects need not only be caused by variables, it may be even done by a function participating in an expression. Side effects hidden within the function shows the design is not good and such functions make debugging complex and reduce readability.

Side effects is the idea to be understood well since it can affect the portability of your code and reduces the quality of your code if used without care.

Most of the inexperienced C programmers find happiness in posing questions to their fellow programmers like:

```
printf("%d %d %d %d", ++i, i++, ++i, i++);

i = ++i + i++ + ++ i;
```

that can lead nowhere. The point is that such expressions involve side effects and the result value of evaluating the expression will not be the same across various systems.

Once my friend asked me about the result of executing the following piece of code:

```
signed char ch =  5;

while(ch = ch--)

     printf("%d",ch);
```

His aim was to print the value of ch and decrement it as the loop executes that would eventually terminate when the value of ch becomes 0. So for this code he expected the output to be 43210. But the program went to infinite loop, printing 555555….! Such are the vagaries of the side effects.

Actually, if you think you can reason it out why this code went to infinite loop. However, remember that the same code may run perfectly as expected in another system,

giving expected results. So *beware of side-effects*. Understanding about side effects is so important.

Normally increment, decrement and assignment operators cause side effects (called as side effect operators). A problem in side effect operators is that it is not always possible to predict the order in which the side effects occur. For example:

```
int a, i = 2;
int a = i++ + ++i;
```

can give different results depending on the implementation. If a side effect operator is used in an expression then the effected variable should not be used anywhere else in the expression. On the other hand, the ambiguity can be removed by breaking it into two assignment statements like this:

```
int a = i++;
a += ++i;
```

Side effects will be completed in case a semicolon or a function call is encountered.

A program should not depend heavily on side effects. It is not a desirable property if it needs to be portable. For example:

```
a[i] = ++i;
```

will always give the same result when executed in your system but may yield some other result it other systems.

By using side effects, debugging becomes harder. For example: A function participating in an expression that alters the value of a global variable.

On the other hand, side effects are not always bad. Sometimes it increases the readability of the code or it smoothens the flow of control and can be handy if used carefully.

For example, I preferred using the following code in one of my programs.

```
while ((ch = getch()) != '\n')
        /* skip  */  ;
```

this has better readability than the equivalent code and is short and so I preferred it.

4.13.1 Sequence points

To determine the effect of side effects in an expression, ANSI C defines sequence points. "Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored". If the statement involves change of value within a sequence point we can say that side effect is involved and that the behavior is undefined.

Most operators impose no sequencing requirements, but a few operators impose *sequence points* upon their evaluation [Rationale ANSI C 1999]. The operators are:

- comma,
- logical-AND,
- logical-OR, and
- conditional.

The expressions involving these operators mostly do not suffer from the problem of side effects.

4.14 Discarded Values

There are some contexts in which the expression appears, but its value is not used:

- An expression statement,

```
1. 10; "good"; 10.0;
2. void ;
// should be given where declaration is possible
3. (void) i * j ;
//intentionally discarding the value of expression
```

- The first operand of the comma operator,

```
int i = (10,20);
// the value 10 is ignored
```

- The initialization and increment expressions in a *for* statement, because they are executed for their side-effects,

```
int i = 0;
for(++i; i<10; i++)
      // do something
```

- Return value from function call of not fetched in a variable. E. g.

```
int add(int a,int b) { return a+b ;}
int main( ){
 add(1, 2);
}
```

```
// no variable to assign the return value
```

This discarding of return value can be made explicit by casting it to void,

```
(void)add(1, 2)
```

In all these cases, the expression's value is discarded. Value of an expression without side effects is discarded, the compiler may issue an error warning message, this may also occur if main operator of a discarded expression has no side effect. Side-effect producing operations include assignment and function calls.

4.15 Constant Expressions

These expressions must evaluate to a constant at compile time (or sometimes at link-time). In all cases, evaluating a constant expression is identical to the result of evaluating the same at run-time.

- You can use a constant expression anywhere that a constant is legal.
- All constant expressions may contain integer constants, char constants, sizeof operator and enumeration constants.
- Constant expressions are required in following situations,
  - after case in switch statement,
  - to specify size of an array,
  - for assigning value to enumeration constant,
  - as initial values for external and static variables (for this unary & operator can be used),
  - to specify bit-field size in structure definition,

- as an expression following the #if (here sizeof and enumeration constants are not permitted).

- Constant expressions cannot contain any of the following operators(unless the operators are contained within the operand of a sizeof operator)
  - Assignment,
  - Comma,
  - Decrement,
  - Increment,

  and function call.

It is worth noting that ?: operator can be used in constant expressions. The values defined using const qualifier cannot be used in constant expression.

```
const int arraySize=10;
int array[arraySize];
// this will issue an error.
```

4.16 Control flow

Control flow in the program is altered by if-then-else statements, loops, break, continue and goto statements. Direct way of altering the program flow in C is through goto and setjump / longjmp library routines. The use of goto is considered to harm the structured design of the program [Dijkstra 1968] although most of the languages support goto statements. Dijkstra notes, “The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one’s program”. That paper stirred lot of arguments in programming community leading to wider approval that goto statements

hurt structured programming. So the use of goto statements is strongly discouraged and gotos can be altogether eliminated from use in the programs by using the other control transfer statements.

One control transfer statement is switch and in that case statements integral constants are expected. Floating types are not possible to use with 'switch' statements because the designers have thought that since checking of exact equality in floating point is not portable [Rationale ANSI C 1999].

4.16.1 Duff's Device

One of the sources of nasty bugs in C is that the case statements in switch statement are fall-through. This property is exploited in a technique called as Duff's device [Duff 1984]. The objective is to get 'count' number of data pointed by 'from' to 'to'. The code for this was,

```
send(to, from, count)
register short *to, *from;
register count;
 {
     do
      *to = *from++;
      while(--count>0);
  }  // this program fails if count is equal to zero.
```

and this version compiled in VAX C compiler ran very slow for some reason. So Tom Duff proposed another version,

```
send(to, from, count)
register short *to, *from;
register count;
{
 register n=(count+7)/8;
 // get number of times to execute while loop
 switch(count%8){
     // go to the remaining mod value
     case 0: do{ *to = *from++;
     case 7: *to = *from++;
     case 6: *to = *from++;
     case 5: *to = *from++;
     case 4: *to = *from++;
     case 3: *to = *from++;
     case 2: *to = *from++;
     case 1: *to = *from++;
           }while(--n>0);
           // this loop is executed n times
   }
  }
  // this program fails if count is equal to zero.
```

The idea is to find out the number of times the loop to be executed in ‘n’ and call switch to copy for modulus value. The do-while loop just ignores the case statements as labels. This technique exploits the fact that case statements do not break automatically. It

is not clear whether this technique is for or against the fall-through property of switch, but is presented here just to show how such mechanisms can be exploited.

4.17 The Structure Theorem

Is it enough to have only few control flow statements like if, while, for etc. Does this limit the expressive power of a programming language?

The structure theorem [Bohm and Jacobini 1966] shows that *any program construct can be converted into one using only while and if statements*! To give one example, the equivalence between the for and while loops is a well known one:

```
for(init;check;incr)
{
// body
}
// is equivalent to,
init;
while(check)
{
// body
incr;
}
```

The theorem proves that all the programs can be programmed using only standard structures (for your astonishment, only ‘while’ is enough and even ‘if’ is not necessary! A little analysis will help you understand that).

Point to Ponder:

Any algorithm can be written in terms of sequence of statements(sequence), alternation (selection) and repetition (and with some boolean variables).

5 POINTERS, ARRAYS and STRINGS

Addresses are given to us to conceal our whereabouts!

'Saki'(H.H. Munro)

Pointers are forte of C and C employs pointers more explicitly and often than in any other programming language. It is the most difficult area to master and error prone. C's power is sometimes attributed to pointers.

In C the design, decision is made such that the usage of pointers is explicit. C pointers can point to virtually any object/anywhere. "...its integration of pointers, arrays, and address arithmetic is one of the major strengths of the language" [Kernighan and Ritchie 1988].

Point to Ponder:

A pointer is an address! [Allison 1998]

5.1 Void pointer

void * acts as a generic pointer only because conversions are applied automatically when other pointer types are assigned to and from void *'s. 's (but note that the ANSI standard made void * pointers a generic type for pointers to objects, it exempted function pointers from this universality). In memory, it is aligned in such a way that it can be used with pointer to any other types and for compound types also.

A pointer must be declared as pointing to some particular type. This holds true even for void *. void * is means it is capable of pointing to anything as opposed to the popular belief that it is a pointer to nothing.

5.2 Multiple level of pointers

Declaring multiple level of pointers is possible. There is no limit for indirection as far as the standard is considered, but for practical programming more than three levels of indirection is rarely required. For each level of indirection qualifiers may be introduced,

```
unsigned int *const *volatile multiLevel;
```

Consider the following code that makes compiler issue an error:

```
int arr2D[10][10];

int **ptr2D = arr2D;
```

This is because both are not of the compatible type. Since the compiler issues an error, and both are of different types, you always resort to explicit casting,

```
int **ptr2D = (int **)arr2D;
```

Alas, the compiler again issues an error stating that the array type cannot be casted to char **. But you are intelligent and know that you can always cast any pointer type to void *, so try this technique:

```
void * temp = (void *) arr2D;
int **ptr2D = (int **) temp;
```

To your frustration, the compiler complains that it cannot convert array type to void * type and you give-up trying and end-up deciding to get a new C compiler.

I illustrated through the steps because this is normally the way we attack problems. When an error message occurs, we try to do something that will shut-up the compiler.

Actually the problem associated with is different. Had you closely followed the error message, you would have known that arr2D is not of two dimensional pointer type, it is of type (int *)[10]

Exercise 5.1:

void * is generic pointer type for any pointer type(like char *,int * etc.) . Find out what is the generic pointer types for pointer-pointer, pointer-pointer-pointer (like char **…) etc? Is it void **… or is such generic pointer types necessary at all?

5.3 Size of a Pointer Variable

```
printf("sizeof (void *) = %d ", sizeof( void *));
printf("sizeof (int *)    = %d", sizeof(int *));
```

```
printf("sizeof (double *)  = %d", sizeof(double *));
printf("sizeof(struct unknown *) = %d", sizeof(struct unknown *));
```

All will print the same value for a particular machine. Similarly the whatever may be the dimension of the pointer be, the size remains the same.

```
printf("sizeof (void *) = %d ", sizeof( void *));
printf("sizeof (void **) = %d ", sizeof( void **));
printf("sizeof (void ***) = %d ", sizeof( void ***));
```

All will print the same value for a particular machine. Consider the following two statements:

```
int **arrPtr = (int **) malloc(sizeof(int **)*10);
int **arrPtr = (int **) malloc(sizeof(int *)*10);
```

both of the statements are equivalent even though the first is more clear and the purpose is evident.

The point is that pointer is also a variable, but unlike other variables, these variables store addresses (of other memory locations). *Therefore, the size of any pointer variable irrespective of the type or dimensions is same.* Now, coming to the previously discussed point on forward references, I said declarations like,

```
struct a { struct b * next;};
struct b { struct a *next;};
```

are valid. This is because the compiler knows the size of any pointer variable. So it has no problem in determining the size of the structure which contains the forward reference. Still the declarations like,

```
typedef struct a{
   b *next;
}b;
```

that involve typedefs will flag an error. Typedefs cannot be forward referenced because the type itself that is being used is not known at that point. So the compiler cannot understand what the identifier ‘b’ means, resulting in issuing an error message.

5.4 Pointers to Enumeration Types

```
enum bool{true, false} boolVar,*boolPtr;
boolPtr = &boolVar;
*boolPtr = true;
printf("%d",*boolPtr);
// prints
// 1
```

Pointers for enumeration types can be created and this shows the close relationship between the integers and enums because enums are internally of integral types. Although such pointer types are rarely used its nonetheless possible.

5.5 Pointers to Structure Members

C assures that the order of the structure members is assured as given by the programmer. Consider the implication of this in the following code:

```
struct someStruct{
     int i;
```

```
    float j;
}aStruct;
void *ps1 = (void *)&aStruct.i;
void *ps2 = (void *)&aStruct.j;

if(ps1<ps2)
    printf("This always get printed");
// it prints
// This always gets printed
```

The order of the members is maintained in the memory. This concept is easily understood in the case of structures. What about unions? Look at the following code:

```
union someUnion{
    int i;
    float f;
}aUnion;
void *pu1 = (void *)&aUnion.i;
void *pu2 = (void *)&aUnion.f;

if( pu1==pu2)
    printf("This too is always get printed");
// it prints
// This too is always gets printed
```

This shows that the pointers to the members of the same structure are always equal, stating that they start at the same location, a shared location.

5.6 Pointer Arithmetic

Arithmetic on pointers differs from ordinary arithmetic, so it is called as pointer arithmetic. Arithmetic on pointers is limited to addition, subtraction and comparison. Addition operation has restricted usage. Two pointers cannot be added. Any other mathematical operation is meaningless and not allowed. But even in the three allowed operations, the arithmetic is assured to be meaningful and defined only if the address to which it points to is limited to the range of the (same) array.

```
char cArr [10][10];
printf("%p..%p", cArr, cArr + 1);
// first is the base address and the next is 10 locations
// after that and sizeof char is 1 byte

int    iArr [10][10];
printf("%p..%p", iArr, iArr + 1);
// first is the base address and the second is
// 10*sizeof(int) locations from the base address

int iArr [10][10] ;
int **ptr = (int**)iArr;
printf("%p..%p", ptr, ptr + 1);
// first is the base address of iArr and the second location
```

```
// after sizeof(int) locations from ptr. ptr is an ordinary
// integer pointer and knows nothing about the type or dimension
// of the array it points to
```

Arithmetical operations on object pointers of type "pointer to type" automatically take into account the size of type (i.e. the number of bytes needed to store a type object).

```
intPtr++;
//increments pointer to the location for sizeof(int)
doublePtr++;
//increments pointer to the location for sizeof(double)
someStructPtr++;
//increments pointer to the location for sizeof(someStruct)
```

Subtracting two pointers to elements of the same array object gives an integral value of type ptrdiff_t (defined in <stddef.h>)

As I have said, addition operation has restricted usage. Restricted usage in the sense two pointers cannot be added but other addition operations like increment and the addition with constant values/ variables are allowed.

```
int *iarr = {1,2,3,4,5};
int *jarr = {1,2,3};
int i = 1;
iarr += i;
// allowed
iarr += 2;
// allowed
iarr++;
```

```
// allowed
iarr + jarr;
// not allowed
```

Let us see another example to fix it in mind,

```
int iarr[10];
int *i = &iarr[5] , *j = &iarr[8];
int *k= i + j;
// Not possible. Two pointers cannot be added. Here itself you
// can note that the k will point beyond the limits of iarr.
int diff = j – i;
// int diff = &iarr[8] - &iarr[5]; is also the same.
printf(“%d”,diff);
// prints 3 irrespective of the size of integer in that system.
// In other words, the subtraction operation gives the difference
// of positions.
```

As I have said, two pointers cannot be added. However, practically, requirements may arise for such pointer additions as in the following one (this example is in [Kernighan and Ritchie 1988] page138):

The requirement is to find the middle element of the array in binary search algorithm. ‘low’ and ‘high’ are the two pointers pointing to the beginning of the array.

```
struct key *low = &tab[0];
struct key *high = &tab[n];
struct key *mid = (low + high)/2;
 // not possible, cannot add to pointers
```

In such cases the alternate way can be used for finding the middle value,

```
struct key *mid = low + (high - low)/2;
```

Since subtracting two pointers is perfectly acceptable, this turns out to be a fine solution.

5.7 The NULL Pointer

NULL is a pointer constant and its definition may vary with implementations. It is the universal initializer for all pointer types. It is normally defined as follows,

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

Instead of this you can use 0 or 0L explicitly if you prefer.

Why NULL is used as end of the string?

Because the NULL is the only character that is neither printable nor a control character.

In general, when we program with pointers, it is necessary to guard against possibilities of accessing null pointers. The following code tells one such way to do that,

```
int foo(int * p){
if(p){
      // before using p make sure that it is not a null pointer
      // before processing on it.
}
}
```

Or still more intelligently in case of strings as follows,

```
if(p != 0 && strlen(p) > 10)
// no problem. if p == 0 then strlen(p) will not be evaluated, this is
```

```
// a good way to make sure that access to null pointer is not made
```

5.8 Illegal Accesses

If the pointer variable is not initialized and if access is made to that variable, the behavior is undefined. The pointer arithmetic holds only if the range is within the array. So it is always good to check the validity of the pointer before accessing it. The following code makes sure that ptr is not NULL before accessing it.

```
if(ptr && *ptr)
   printf("%d", *ptr);
```

This uses the property that boolean operator will be evaluated until the truth-value is determined. So if the value of ptr is zero, the condition will fail and will come out.

5.8.1 Dangling Pointers

Dangling pointers and memory leaks are the nastiest problems that may arise in a program. Dangling pointers arises when you use the address of an object after its lifetime is over. This may occur in the situations like,

- returning addresses of the automatic variables from a function

```
int *fun()
{
int x = 10;
return &x;
}
….
```

```
int *k = fun();
// dangling pointer. never return auto variables from functions
```

- using the address of the memory block after it is freed

```
int *block = (int*) malloc(sizeof(int));
if(a>b){
   free(block);
   }
….
*block = 10;
// dangling pointer. accessing the memory area after it is freed
```

Dangling pointers may harm the execution of your program to any extent. So it should be strictly avoided.

5.8.2 Memory(storage) Leaks:

Memory/storage leaks occur when you fail to free the storage when it is no longer used.

```
int *ptr = malloc(100);
ptr = ptr1;
// now there is no pointer to point at the allocated
// memory block and that too is not freed.
```

The programmers frequently forget that the lifetime of a dynamically allocated block is that of the program.

Most of the systems support a function known as ‘alloca’ that allocates memory dynamically in stack. The advantage is that the memory is automatically recollected

when the function exits, and is local to a function (so no possibility of memory leaks). The problem also is the same property itself! We cannot return the dynamically allocated chunk back from the function and other functions cannot manipulate it. In addition, standard library does not support it.

The effect of memory leaks is not evident until the system runs out of memory. Consider the case where multiple programs are executed at a time, which share same heap. If the memory leak is much the system will run out of memory and the only way to recover is to reset the system. Therefore, it is the responsibility of the programmer to promptly free the resources.

5.9 Wild Pointers

- Dangling references,
- Uninitialized pointers and
- Corrupted pointers

are sometimes referred to as wild pointers.

A global pointer should never reference a local auto variable, because when that local variable goes out of scope its memory is released to be reused by the stack for something else. Now this global pointer becomes a dangling pointer. If the global pointer is later used it may be erroneously referencing something else.

To avoid the problems with uninitialized pointers (that is discussed above), they should be initialized if known with the intended variable addresses or to NULL.

Misusing pointer arithmetic and making them pointing to illegal addresses creates corrupted pointers. They are also created if made pointed to other word boundaries than its intended type.

Since wild pointers can pose serious threats to the correct execution, validity of the programs and notoriously hard to debug. The programs should be free from these wild pointers.

The checking validity of a pointer object i.e. if the pointer is pointing to a valid location or not is a very big problem and in some cases, the validation cannot be done at-all. An example is function arguments taking pointers. For example,

```
size_t strlen(const char *);
```

It is taken for granted that the address passed is a valid one and processing is made accordingly (and there is no way to verify either).

Exercise 5.2:

Find out the subtle error in the following code segment:

```
void fun(int n, int arr[]){
int *p=0;
int i=0;
while(i++<n)
     p = &arr[i];
*p = 0;
}
```

5.10 Pointers and Const Qualifier

When the pointer declaration involves const qualifier, it can be understood easily by reading from right to left.

```
const int *ptr = &var;

int const *ptr = &var;

// both are pointer to const.
```

are equivalent. It can be read, as ptr is a pointer to a variable of type integer that is a constant. It says the value of the variable pointed by ptr may not be modified.

```
*ptr = 10;

// invalid,  but

ptr = &var1;

// is valid
```

This can also easily be remembered by the familiar prototype of strcmp.

```
char *strcmp ( const char *str1, const char *str2);
```

where the const in the arguments guarantee that the strings pointed by the arguments will not be changed.

```
int *const ptr = & var;

// constant pointer
```

which can be read as ptr is a constant pointer(don' get confused with pointer constant. Pointer constant is used in case of array names) to variables of type int. Now,

```
ptr = &var1;

// invalid, but

*ptr = 10;

// is valid
```

Combining the both,

```
const int * const ptr = & var;
```

which can be read as ptr is a constant pointer to a variable of type int which is constant. So, both the pointer and the variable pointed by it cannot be modified.

```
ptr = &var1;
// invalid, also
*ptr = 10;
// is invalid
```

5.10.1 Difference Between Pointer Constant and Constant Pointer

Pointer constant is distinct from constant pointer and should not be confused with each other. An array name for example is a pointer constant. Function names are also pointer constants because they address they point to cannot be changed. NULL as we have just seen is also a pointer constant. It is used in the sense as the word ‘constant’ as used in character constant, integer constant etc. Where the implicit meaning is that the object itself is a constant and so no chance of modifying. In other words it can never be an lvalue and the ‘const’ness is implicit and is a must.

But in the case of constant pointers the word const is applied as an adjective qualifying the pointer as a constant. So it means that it also may not appear as an lvalue and the address it ‘contains’ may not be modified. Unlike pointer constants the constant pointer it can be modified indirectly and the ‘const’ness is imposed artificially by the compiler.

```
int array[10], someArray[10];
```

```
// here 'array', 'someArray' are pointer constants

int *const ptr = array;
// ptr is a constant pointer and it is initialized
// with the rvalue 'array'.

int *temp;

ptr = someArray;
// error. address it stores cannot be modified.

ptr[0] = 10;          // o.k.

temp = &ptr;
*temp = someArray;
// o.k. Now ptr points to someArray;

temp = &array;
*temp = someArray;
// no chance. This is a pointer constant
```

Point to Ponder:

Pointer to a constant means that the value is not changed through that pointer, not that it is unchanged through any pointer.

5.10.2 Near, Far and Huge Pointers

Pointer is defined as the type that can hold the addressable range of the system. In addition, the size of the pointer is normally the size of int (for efficiency). This two may not necessarily are be compatible with each other, so leads to problems in few implementations.

In Intel's x86 systems, the size of int is 2 bytes. So pointers may be implemented to hold two bytes. This can address up to 2^{16}-1 memory locations (size of one segment in these machines). This is enough for small programs that manipulate fewer amounts of data. Nevertheless this is far less than the addressable portion of the memory. To overcome this difficulty they have an alternative patchwork where they have far pointers. Therefore, a pointer variable when declared as far is of 4 bytes that can address up to 2^{32}-1. This too cannot determine a memory location exactly if used in comparisons like ptr1 = = ptr2. Because it is stored in the format segment : offset. To overcome this, when you declare a pointer as huge, it also takes 4 bytes of memory, but which stores the addresses as absolute addresses. Again to make these kinds of changes to pointers as default type they have memory models etc. which makes the problem worse.

Such implementations (compiler vendors like Microsoft, Borland etc. support this one) are specific to some subset only and the reader is recommended to follow the standard strictly if he wants his software to be portable.

5.11 Arrays

Arrays are list of objects of same type. Array name is an lvalue. However, a non-modifiable lvalue (a pointer constant). Since an array name points to the beginning of a memory location, if it is modified just like a pointer reference, that memory location will be lost. So it is called as pointer constant signifying that you may refer its value but may not modify it. As we said previously, assignment between arrays is not allowed, whereas all other assignment types are allowed. The reason being that arrays are pointer constants indicating they are addresses, whereas a structure assignment is allowed because a structure name signifies a value.

5.11.1 Low Level Nature of Arrays

The C arrays are low-level in nature. They reflect the storage of the array elements in the physical hardware. To support the argument that C-arrays are sufficiently low-level, consider the following points,

- The array name refers to the starting address of where the actual storage of the array members begin, and this helps in assigning the array to a pointer of the same element type
- The array elements are stored in a contiguous storage of bytes,
- No padding is done,
- The arrays are not self-describing.

When you declare an array, it is assured that it is allocated contiguously and with no padding between the array elements done. Were padding possible, it would be costly

in case of large and multi-dimensional arrays. Arrays doesn't have any information store in itself on the type of information they store or its size etc. Consider the following example:

```
void foo(void *vPtr)
{
  // ??? is vPtr
}
int iArr[] = {1,2,3,4};
foo((void *)iArr);
```

now, using vPtr there is no way to determine the type of the array or size of the array or other related information.

Only through all these properties, pointer - array relationship is possible and pointer arithmetic becomes possible with arrays.

5.11.2 Array Bounds

```
int i, a[10];
for(i=0;i<=10;i++)
     a[i]=0;
```

This seems to be a harmless code. But the for loop accesses a[10] which is not there. This is called as 'one past error' (also referred to as 'fence-post' error) that even experienced programmers commit. In C array index begins from 0 and there is no way to change this default base. Pointer arithmetic should be limited within the available block. So, references like a[-1] and a[15] are illegal since the reference is outside the block and so

the behavior is undefined. ANSI C loosens this rule by allowing access one element past the memory block. However, it cannot point below the base address.

```
int array[10];
array[10]=100;
// O.K. ANSI C relaxes the rule.
array[11]=100;
// illegal
array[-1]=100;
// illegal. Cannot point below base address.
```

To reason out this behavior consider the following code in a x86 machine.

```
int array[10];
int far *ptr = array;
// say the array is located in 1234 : 0000
ptr --;
// will now point to 1234 : FFFF
if( array < ptr)
     printf("The condition becomes false and this statement will not be executed");
```

Due to such problems as in the previous code, it is illegal to access the elements below the base address. But what about ‘one past error’. The same problem occurs if the array ends at the end of a segment and tries to move past it. In this case, an ANSI compiler makes sure that the array bound is at-least one below the segment limit. All this is due to this ubiquitous one past error. So by accidentally accessing a[10] as in that

example will not lead to undefined behavior (but it by no means say that a[10] is correct. It just avoids undefined behavior).

5.12 Arrays and pointers

Arrays and pointers are closely related. Pointers are just l-value to the object they point to. array[index] is exactly same as *(array + index) (or in turn *(index + array) in any case). Since most of the times there is a direct hardware support, the pointer implementation is very efficient.

In most of the languages, arrays are implemented as pointers only. In C, the relation is direct and explicit. So we can refer array[index] as index[array] that means the same. Exactly one of (a, b) must be a pointer and one of (a, b) must be an integer expression. So, array[1] is equal to 1[array] since array is a pointer and 1 is an integer expression. Even casting may be applied and is valid till the condition is satisfied. For example:

```
a[b]==(char*)a[(int)b].
```

Consider this example,

```
int a[10][10];

int k = 2[a][2];

// O.K. Treated as *(*(a +2)+2)

int m = 2[2][a];

// Error. Treated as *(*(2+2)+a) which is illegal.
```

Showing the relationship between arrays and pointers is very simple as in the following; assume that ‘arr’ is declared as:

```
typedef char Type;
Type arr[10];
```

Lets start with assuming the relation,

```
&arr[0] == arr
```

The base address of the array is &arr[0] and it is equal to just saying arr. In other words the array starts at the 0'th position and applying the & operator to that position yields nothing but the base address of that array. The same can be expanded as,

```
&arr[0] == (char *)arr + 0 * sizeof(Type)
```

Note that the casting 'arr' to (char *) is essential to make sure that the addresses are added in a scale of 1.

So whatever may be the type, for arr[1], the following relation holds true:

```
&arr[1] == (char *)arr + 1 * sizeof(Type)
```

And you can also generalize it for,

```
&arr[i] == (char *)arr + i * sizeof(Type))
```

This shows the relationship between single dimensional arrays and pointers.

Why did I use a typedef for 'Type'? That is to extend this relationship to multi-dimensional arrays. Consider,

```
typedef char Type[10];
typedef char Type[10][20];
// or more
```

and still the relation holds good, for the simple reason that multi-dimensional arrays are made-up of single-dimensional arrays.

5.12.1 Flattening of Arrays

Arrays are implemented as a contiguous memory block. This information can be used to manipulate the values stored in the array and allows rapid access to a particular array location.

```
int arr[10][10];

int *ptr = (int *)arr ;

ptr[11] = 10;

// this is equivalent to arr[1][0] = 10; assign a 2D array

// and manipulate now as a single dimensional array.
```

The technique of exploiting the contiguous nature of arrays is known as ‘flattening of arrays’.

5.12.2 Ragged Arrays

```
char **list;

list[0] = "United States of America";

list[1] = "India";

list[2] = "United Kingdom";

for(int i=0; i< 3 ;i++)

   printf(" %d ",strlen(list[i]));

// prints 24 5 14

// list[0] -> "United States of America"

// list[1] -> "India"

// list[2] -> "United Kingdom"
```

This type of implementation is known as ragged array, and is useful in places where the strings of variable size are used. Popular method is to have dynamic memory allocation to be done on the every dimension. The command line arguments for example are passed only as ragged arrays.

5.12.3 Comparing flattened and ragged arrays

After knowing what is flattening of arrays and ragged arrays it is the time to compare the two. Let us have an example,

```
int flattened[30][20][10];
int ***ragged;
int i,j,numElements=0,numPointers=1;
ragged = (int ***) malloc(sizeof(int **) * 30);
numPointers+=30;
for( i=0; i<30; i++)
        {
        ragged[i] = (int **)malloc(sizeof(int*) * 20);
        numPointers+=20;
        for(j=0; j<20; j++)
            {
            ragged[i][j]=(int*)malloc(sizeof(int)*10);
            numElements +=10;
            }
        }
```

```
printf("Number of elements = %d",numElements);
printf("Number of pointers = %d",numPointers);
// it prints
// Number of elements = 6000
// Number of pointers = 631
```

As you can see the ragged arrays require 631 pointers, in other words, 631 * sizeof (int *) extra memory locations for pointing 6000 integers. Whereas, the flattened array requires only one base pointer: the name of the array enough to point to the contiguous 6000 memory locations.

On the other hand, the ragged arrays are flexible. In cases where the exact number of memory locations required is not known you cannot have the luxury of allocating the memory for worst possible case. Again, in some cases the exact number of memory space required is known only at runtime. In such situations ragged arrays become handy.

To illustrate, consider the example of a text editor. The size of the text the user is going to type cannot be predicted. If worst case of 256 columns and 1024 lines is assumed and the space is allocated statically, it will require 256 * 1024 bytes, or 256 kilo-bytes. Even if we allocate such big chunk of memory, our text-editor will have the limitation that the user can type only up-to 256 characters per line and 1024 lines at the maximum. On the other hand, declare a 2D pointer for storing the information that the user types. Memory can be allocated dynamically to fit the need. If the user types nothing, no space will be allocated. If he types a lot with varied number of characters in each line, the space can be allocated exactly with additional space for storing the pointers for each line. There is no limit on the number of lines, since the pointers to the line is also

allocated dynamically and can vary. So the only limitation happens is to be the size of the available memory. As you can see this ragged array approach conserves lot of space in this case and is very flexible.

So the selection depends on our requirement, and each approach have their own advantages and disadvantages.

5.12.4 Row-major of Arrays

Unlike the languages Pascal and FORTRAN that follows column-major order, C follows row-major ordering for multi-dimensional arrays. 'Flattening of arrays' can be viewed as an effect due this aspect in C.

What is the significance of row-major order of C? It fits to the natural way in which most of the accessing is made in the programming. Lets look at an example for traversing a N * M 2D matrix,

```
for(i=0; i < N; i++)
     for(j =0; j < M; j++)
          printf(" %d ", matrix[i][j]);
```

Each row in the matrix is accessed one by one, by varying the column rapidly. The C array is arranged in memory in this natural way. Consider the following one,

```
for(i=0; i < M; i++)
     for(j =0; j < N; j++)
          printf(" %d ", matrix[j][i]);
```

This changes the column index most frequently than the row index. Is there any difference in efficiency between the two codes?

Yes. The first one is more efficient than the second one! Because the first one accesses the array in the natural order (row-major order) of C, hence it is faster, whereas the second one takes more time to jump (If you want to verify the fact that the first one is faster than the second one, use clock() function in <time.h>, run in your machine and see the time difference to execute them. In our machine, the second code took twice as much time than the first one to execute).

The difference may be small in case of small arrays. However, as the number of dimensions and the size of element increases the performance difference would be significant.

5.12.5 Static Allocation

In C static allocation is made for arrays. So all the information required for allocation of memory is needed. In other words incomplete information will not suffice and will lead to compile time error. This is except for first dimension.

```
int a[] = { 1,2,3,4 };
// valid. It is left for compiler to calculate the dimension.
int a[][]={ {1,2}, {3,4} };
// not valid. Only one dimension may be left free
// to be calculated by the compiler
```

A constant expression is allowed to specify the size of the array.

```
int a[]={1,2,3,4};  // o.k.
int *a ={1,2,3,4};  // error.
```

*[] and * usage are not equivalent and so cannot be used interchangeably.* int * a cannot be used in place of int a[] in the above declaration and they in no way are equivalent. The confusion between the two of the usage arises from the fact that they have the same meaning when used as function arguments.

I.e. foo(int *arr) and foo(int arr[]) are equivalent. Similarly foo(int**arr) and foo(int *arr[]) are all equivalent. But they are *not* equivalent elsewhere.

Consider,

```
char *pstr = "string";
char astr[] = "string";
printf("pstr = %d, astr = "%d",sizeof(pstr)
,sizeof(astr));
// if size of pointer type is assumed to be 4 bytes it prints
// psrt = 4, astr = 7
```

The difference is evident in case of multi-dimensional arrays:

```
int *i;
int j[20];
i = j;
// no problem. i and j both are of type int *.
int **i;
int j[10][20];
i = j;
// error/warning : cannot convert from int (*)[20] to int **
```

These two are not equivalent. i and j are of different types and hence i cannot be assigned to j. An explicit casting is required to do the same.

```
i = (int **)j;
// o.k. explicit casting
```

However, the same j can be passed without casting as in:

```
extern int foo(int *arr[]);
int j[10][20];
foo(j);
// o.k. foo((int**)j); is not required.
```

This is because only pointers not arrays can be passed to functions and only here the [] and * can be used interchangeably.

Usage of negative indices is not an error.

```
int iarr[] = {1,2,3,4};
int *iptr = &iarr[2];
printf("%d", iptr[-1]);
// prints 2
```

The negative indices may even be useful in some cases. If 'iptr' is pointing somewhere middle in the integer array then iptr[-1], iptr[0], iptr[1] will give the previous, current and next integers respectively.

```
char *listOfTokens[] = { "char", "short", "integer", "long", "float", "double"};
char **currTok =  listOfTokens+3;
printf("prevTok = %s\n currTok = %s \n nextTok = %s",\
currTok[-1],currTok[0],currTok[1]);
// prints
```

```
prevTok = integer

currTok = long

nextTok = float
```

Using negative indices is not recommended because it may confuse the reader who reads the program. The array access should be within the limits of the array and using negetive indices may possibly violate it.

```
int iarr[] = {1,2,3,4};

printf("%d", iarr[-1]);

// undefined behavior
```

5.12.6 Array Names are Pointer Constants

```
char *ptr = "string";

char arr[] = "string";

ptr++;

// perfectly o.k.

arr++;

// compiler error
```

The reason for the behavior being that arr is a name of an array, and if expressions such as a++ were allowed it may leave the memory area allocated to be stranded and you may miss the link later. To avoid these kind of pitfalls C restricts the arithmetic on array names a calls it as pointer constant. This means you can examine the contents using the name but may not modify it. In case of ptr in the example it is declared as a pointer so is having all the rights to be arithmetic performed on it.

5.13 Strings

Strings are implemented as const char pointers. Since they are pointers the string manipulation as pointers is very efficient (e.g. the implementation of most of functions in <string.h>).

5.13.1 String constants

A sequence of characters enclosed within double quotes is known as string constant. It includes printable characters and escape characters. Two continuous strings separated by white space characters (that include new-line character) are concatenated by the compiler and treated as a same string (‘stringization’ operation):

```
char str[] = “stringi”
             ”zation”;
puts(str);
// prints stringization
```

Consider the following example,

```
char name[] = {"ravikumar",
               "hari",
               “ranjitha”
               “prakash”,
               0,
                };
```

```
while(name[i])
      puts(name[i++]);
```

The programmer expected that four names be printed but he got only three. What went wrong? Clue. It printed ravikumar, hari and ranjithaprakash. The programmer missed a comma while typing that led to stringization of strings, causing an unexpected problem.

Constant strings may get placed either in code or data area and that depends on the implementation. That means, the string constants are available to use even after functions are exited. Let us see an example:

```
char *try1()
{
char *temp = "string constant";
return temp;
}
char *try2()
{
char temp[] = "character string";
return temp;
}
char *try3()
{
char temp[] = {'c','h','a','r','s','\0'};
return temp;
```

```
}
int main()
{
puts(try1());
// prints "string constant"
puts(try2());
// undefined behavior!
puts(try3());
// undefined behavior!
}
```

In the try1() 'temp' is initialized to a character const. Since the character constants are stored in code/data area and not allocated in stack, this doesn't lead to dangling pointers. But this is not the case of try2() and try3(). These two functions suffer from the problem of dangling pointers. In try2() temp is a character array and so the space for it is allocated in heap and is initialized with the character string "character string". This is created dynamically as the function is called, so is also deleted dynamically on exiting the function so the string data is not available in the calling function main(). So the puts reads some unknown position leading to undefined behavior. The function try3() also suffers from the same problem and the problem is more easily identifiable because the initialization is done character by character explicitly.

5.13.2 Shared Strings

```
char *ptr = "string";
char arr[] = "string";
```

```
ptr[3] = 'a';
puts(arr);
// may print 'strang' !!!
```

You know that constants are also allocated space while compilation of the program. In this example, "string" is stored somewhere in the memory and its address is stored in ptr. Compiler may store both the strings in a single location and assign the same string to both arr and ptr. Any modifications through ptr may also affect arr. This concept is known as shared strings.

It is to be noted that the shared strings does not have to occur in full strings, part of the strings can also overlap.

```
char *str1 = "some string";
char *str2 = "string";
// beware that the part "string" may be common in both the strings
// and so may overlap.
```

Checking for if your compiler uses shared strings or not is easy,

```
char *ptr = "string";
char arr[] = "string";
if(ptr == arr)
     printf("Yes. shared strings");
// or check it directly with the string constants
if("string" == "string")
     printf("Yes. shared strings");
```

These checking works on the common sense that no two different string constants can be stored in the same location.

```
if("someString" == "differentString")
     printf("This will never get printed");
```

This sharing of string optimization by the compiler is to save space. This may result in a significant gain in storage space, if tens or hundreds of similar character constants are used in the source program. For example, you may have "some string" as the string used 100 times in the text. If shared strings are used it will avoid storing "some string" 100 times in the code generated and store only one string instead.

Since shared strings can economize space, how will you force sharing of strings to be enforced by the compiler? One obvious way is to switch on the shared string flag option in your compiler. Another indirect way to do it in programs is to have a character pointer initialized with that character constant like the following,

```
const char *something = "some string";
```

and use that identifier 'something' instead of that string,

```
puts(something);
// instead of puts("some string")
```

in places where the string "some string" is required.

5.13.3 Shallow Vs Deep Copy

```
char * t;
char s[] ="something";
t = s;
//now t also contains the address of the string constant.
```

No copying of contents takes place. Only the address is copied. Still changes made through t affects the object it is pointing.

```
t[0] = 'm';
// now the string becomes "momething"
```

This is called as shallow copy. By default in C shallow copy is done for pointer assignment.

If you want the coping to be done in the space for the t then you should do the same explicitly.

```
char * t = malloc(strlen(s) + 1);
strcpy(t,s);
// deep copy is affected now.
```

Since it is pointing to the different location from the source, (even though the contents of the both are same after the copying) , change by one variable doesn't affect the another.

```
t[0] = 'm';
// now the string pointed by t becomes "momething"
// string pointed by s remains the same "something"
```

In other words, shallow copy is just to copy the pointers whereas deep copy is to provide address for the new object where the copy of the source content is stored.

5.14 Arrays and Strings

Arrays and strings are closely related, as string is just an array of characters. Consider an example,

```
char string[10];
```

```
strncpy(string ,"Tom cruise",3)[3] = '\0';
      // strncpy returns char *
```

Copies "Tom" into string and NULL terminates it. Similarly to traverse a character array this perfectly works,

```
for( i = 0 ; i < 5 ; i++)
      putchar("aeiou"[i]);
```

Although these examples will not help in real-life programs, they show how the arrays, pointers and strings are closely associated with each other.

Look at the following code to see that the rules that apply for multidimentional arrays also apply to character arrays,

```
char *array[10] = {"good", "bad", "better"};
// Allocates three pointers each of size 10 bytes
char *array[] = {"good", "bad", "better"};
// Allocates three pointers of size 5,4,7 bytes respectively
char **array[10] = { {"good", "bad"} ,{"better"} };
// Error. Required information is missing
```

5.14.1 char *s and char Arrays

What is the difference between the declarations,

```
char arrayInfo[]="hello";
char *ptrInfo ="hai";
```

What happens if assignment like this occur?

```
ptrInfo[2]='a';
```

The first one defines an ordinary character array. The second one points to a character string. The compiler is free to store the string constant “hai” anywhere it wishes. Consider that it happens to be stored in a read only memory (ROM). If we want to change this string at runtime, it may produce unexpected result. So if you want to modify the string later, it is advisable to store it in an array as in the first one.

5.15 Arrays and Structures

An array cannot be passed or returned from the function, whereas structures can be. Similarly, structures can be assigned to each other whereas arrays cannot be although both are aggregate data-types. This is a particular place where C suffers from the problem called as ‘lack of orthogonality’ in programming language’s terminology. This is because the array bounds need not be perfectly known. Hence the size of the whole array may not be predicted exactly and due to the close relationship between the pointers and arrays (an array name is a pointer constant and doesn’t correspond to the whole array). But this is not in the case of structures. So it is possible to embed the array in a structure in an array and return it.

6 STRUCTURES AND UNIONS

Raw datatypes, when combined together to have a logical relationship with them, it becomes very powerful. Structures do that job of containtership; unions and bit-fields are slight variations of structures.

6.1 Initializing Structures

Structures may be initialized with the expressions of the same type. The structure members can be initialized with a list of values enclosed within the braces according to the order of members. If any members are left uninitialized then they are initialized to zero.

Structures and unions can also be initialized just like arrays as follows,

```
struct structType aStruct  = {0};
```

initializes all members of sStruct to zero including the padding bits (if any). This applies to unions also.

```
union unionType aUnion  = {0};
```

initializes the space occupied by the union to 0 including the padding bits (if any);

6.2 Name Equivalence of Structures

In C structures can be assigned only if they have the same name (refer to “type equivalence”). This is called as name equivalence of structures[†] in C.

[†] Few other languages provide structure assignment that follow structural equivalence.

```
struct structure1 {int i;} sv1={1}, sv2;
struct structure2 {int i;}s2;
sv2 = sv1;
// OK. Since both variables are of structures of same name.
s2 = s1;
// Compiler error : incompatible types
// Both are of different structure types
```

This kind of assignments cannot be done in C because C follows name equivalence of structures. It serves an important purpose. It provides a means to assure that two different structures that are meant for different purposes can't be mixed together accidentally or purposefully. This leads to safe and clean model for structure assignment [Stroustrup 1994].

So in C, by following name equivalence is that 'power' lost? No.

Consider the example:

```
struct structure1 {int i;} s1={1}, *sp1=&s1;
struct structure2 {int i;}*sp2;
sp2 = (struct structure2 *) sp1;
// now OK. explicit cast
printf("\n%d %d",sp1->i, sp2->i);
// it prints
// 1 1
```

This idea is for assigning to structure pointers of different names. Similarly this idea can be extened to assign one struct variable another with different name.

```
struct structure1 {int i;}s1={1};
```

```
struct structure2 {int i;}s2={2};
s1 = *((struct structure1 *) &s2);
//now OK. explicit cast
printf("\n%d %d",s1.i, s2.i);
// it prints
// 2 2
```

Here an explicit cast does the job to override the name equivalence. When making explicit cast, the programmer is aware that he is explicitly changing the type, so acceptable. When a situation arises that you must override the name-equivalence and require structural equivalence, the simple technique of explicit casting can be used.

6.3 Offsetof Macro

ANSI provides a way to determine the byte offset of any non-bitfield structure member from the base by offsetof macro.

```
size_t = offsetof(type,mem);
// syntax
size_t position = offsetof(student,rollno);
// e.g. of usage.
```

One of the possible implementation for this macro is,

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

or simply as,

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

This information can be very useful in determining how the compiler aligns the members (and about the endian scheme followed).

This macro can be applied to unions also.

6.4 Nested Structures

Nested structures are allowed in C.

```
struct temp{
      char *name;
      struct dateOfBirth{
            int day, month, year;
            }DOB;
      }student;
```

The above example uses a nested structure. Since a new name space is created inside a structure, the names may be overloaded. The nested structures need to be defined (it is not possible to just declare structure because a structure member is needed) inside the structure.

It possible to create a inner structure variable outside the enclosing structure. For example:

```
struct dateOfBirth myDOB;
// legal
```

of the previous nested structure example.

6.5 Structures and Functions

You can create on-the-fly structure definitions in the function arguments and return types. Such structures are available from that point of definition.

```
struct  some { int i; } foo ( some s );
```

defines a structure called some and passes a variable of the same type to the function foo. This style reduces the readability of the code much, so, should be avoided.

Passing a structure by value to a function requires a copy of the structure be sent to the function. Since passing is done through stack, passing structures by value is costlier. So if you really want to send a structure by value make sure that it is very small or alter the program such that it is sent through a pointer to that structure.

6.6 Unions

Unions can be considered as special case of structures. The syntax for both is mostly the same and only the semantics differ. The memory is allocated such that it can accommodate the biggest member. There is no in-built mechanism to know which union member is currently used to store the value. Consider the following code:

```
union  value{
int iVal; double dVal; void (*fp)();
};
union value foo();
union value v = foo();
printf("%d",v.iVal);
```

```
// be cautious. How do you know that foo() has only set
// the int member of the union?
```

In other words, with the union itself it is not possible to know the member that it is currently used. So there is more possibility that you access a wrong member type and get into problem.

The solution may seem simple by introducing another variable of type integer. Now, with addition to the original code you have:

```
#define INT          0
#define DOUBLE       1
#define FUNC         2
int type;
union value v = foo();
// the foo() should set the global variable 'type' to correct value
switch(type) {
case INT        : printf("%d", v.iVal); break;
case DOUBLE     : printf("%lf", v.dVal); break;
default         : (v.fp)();
}
```

Its better to use enums to ints because it specifies a closed set of values as in this case:

```
enum  type { INT, DOUBLE, FUNC };
```

However, there is no logical relationship between the int/enum and the union concerned. To provide that logical relationship, use a structure. The full code may now become:

```
enum type{INT=1, DOUBLE, FUNCP};
union  value{
int iVal; double dVal; void (*fp)();
};
struct dummy{
enum type t;
union  value val;
};
 struct dummy foo(){
     struct dummy v;
     v.t =  INT;
     v.val.iVal = 10;
     return v;
}
int main() {
struct dummy v= foo();
switch(v.t) {
case INT  : printf("%d", v.val.iVal); break;
case DOUBLE : printf("%lf", v.val.dVal); break;
default        : (v.val.fp)();
```

```
}
}
```

Yes, the code becomes longer, but this is a robust solution for using the unions.

Similarly, in case of enums, there is no way to know the number of enumerative constants available using the enumeration itself. It is a good design to have the count in the end of the enumeration. Similarly add a enumeration constant specifying the illegal value. For example:

```
enum holidays{ ILLEGAL = 0, SATURDAY, SUNDAY, NUMOFHOLIDAYS = 2};
// 'NUMOFHOLIDAYS' refers to number of valid holidays
```

6.7 Bitfields

The members of structures that have bitfields cannot be applied with the following operators,

- (indirection),
- & (addressof),
- [] (subscripting) and
- sizeof operator

and all other operators can be applied on them as with structs.

Pointers store addresses and that means the memory locations that are directly addressable can only be used for pointers assigning to pointers. In case of bitfields, direct addressing of bits is not possible. That is why pointer operations (indirection and addressof) cannot be done on bit-fields.

Why [] cannot be applied on bitfields? Simply because of * (indirection) cannot be applied on bit-fields (remember that a[i] is treated as *(a+i)).

Sizeof returns the number of bytes it occupies and not bits. If sizeof were to allowed on bitfields, confusion may arise if it tells the number of bits or bytes the member occupies.

In the case when bit-fields and other data fields come together, the bitfields should come first.

e.g
```
struct student {
        unsigned int rollno : 5;
        unsigned int sex    : 1;
        unsigned int : 3;
        // unnamed member-used as padding bits
        char * name ;
        // note here that the bitfields come first.
        };
```

All the members in a bitfield need not be named ones. Unnamed members cannot be accessed and used from the code and serve the purpose of padding.

Consider the following problem,

```
struct bitField {
        int day   : 3;
        int sex   : 1;
        }sample;
enum workingDay{monday=2,tuesday,wednesday,thursday,friday };
```

```
      #define MALE   0
      #define FEMALE 1
      sample.day = friday;
      sample.sex = FEMALE;
      printf("\n day = %d, sex = %d", sample.day ,
sample.sex);
      // it printed
      // day = -2, sex = -1
```

The poor programmer expected day = 6, sex = 1 to be printed but what happened? Before that lets see something about the range of types. For signed chars, the range is from –128 including 0 to +127. In other words it is from -2^7 including 0 to 2^7-1. Generalizing this idea, for signed integral quantities of bitsize n the range is from $-2^{(n-1)}$ including 0 to $2^{(n-1)}-1$.

Coming back to the problem notice that the bit-fields are ints and the signed is the default for ints. For the member 'day' it takes 3 bits. Finding the range by using our formula it is –4 to 3. 'friday' is 6 that cannot be represented in the range and thus rotates to –2. The same applies to 'sex' field and +1 cannot be represented with single signed bit and so the only bit available is treated as the sign-bit and the value printed is –1.

The cost of using bitfields is losing of portability. How the bit field definition and declaration will pack the bits in the given space depends mostly on the underlying hardware.

The following example is a highly unportable way to get the parts of a floating point variable. It is assumed that the machine is small-endian and it follows IEEE floating point arithmetic.

```
union{
     float flt;
     struct{
          unsigned int mantissa3 :8;
          unsigned int mantissa2 :8;
          unsigned int mantissa1 :7;
          unsigned int exponent :8;
          unsigned int sign:1;
        }bitField;
     }floatVar={-1};
printf("The floating point number:%f\n",floatVar.flt);
printf("Its sign bit:%d\n",floatVar.bitField.sign);
printf("Its exponent :%d\n",
floatVar.bitField.exponent);
// prints
// The floating point number: -1.000000
// Its sign bit: 1
// Its exponent: 127
```

Since portability is the cost, bitfields should be used in places where space is very limited and that functionality is demanding. The space vs. time complexity is involved in

fetching bitfields. In assigning storage for integers, bitfields should not be used instead to save space.

```
struct birthdate {
     unsigned int day    :5;
     unsigned int month  :4;
     unsigned int year   :7;
     }myDOB;
```

is not recommended because the space(one byte) saved is comparatively less preferable to the complex shifting of bits involved (time complexity is involved so, using three integers instead of bitfields is more efficient and is recommended). Separate integers may be used for representing true and false values rather than using individual bits for representing true/false for the same reason.

However bitfields,

- are invaluable in accessing the particular system resources,
- are to avoid unnecessary manipulations by the bit manipulation operators,
- once defined can be accessed in the same way as we access the ordinary variables.

6.8 Self-referential structures

Lets see an example to illusrate the use of self-referential structures,

```
// this code fragment prints the list of names in sorted order
typedef struct nameList{
     char name[10];
```

```
        struct nameList *next;
        }nameList;
    nameList list[] = { {"carl clemens",list+1 },
                        {"prabhakar",0},
                        {"abilash",list+3},
                        {"arvind",list},
                    };
    nameList *temp = list+2;
    while(temp)
    {
        printf("\n%s",temp->name);
        temp = temp->next;
    }
    // it printed
    // abilash
    // arvind
    // carl clemens
    // prabhakar
```

This program works on the rule we have already seen that ‘a declarator is available to be referred in the program code and used from the point where the declaration/definition is made’. Here the identifier list is available and used in its initialization list itself and this is based on the pointer arithmetic. The compiler knows the size of the structure so it can calculate the addresses like list + 3 (which is nothing but list + (sizeof(nameList) * 3).

Such self-referential structures are very useful in implementing data structures like linked-lists, trees etc. that uses pointers to the nodes of same type.

6.9 Padding in Structures

Padding (also referred to as buffering) is an important issue associated with the structures. There may be some alignment requirements required by the environment. For example, ints may required to be aligned at even numbered addresses and longs at address numbers divisible by 4. This leads to internal and trailing padding of bytes in the structure.

Due to this padding, the value returned by the sizeof may not be equal to the simple addition of the sizeof the structure members. Because, sizeof when applied to structures, returns number of bytes required to represent the structure including that padding bytes. The following code illustrates this:

```
struct someStruct {
     char cc;
     float ff;
     double dd;
     void *vp;
};
int size1 = sizeof(struct someStruct);
int size2 = sizeof(char) + sizeof(float) +
     sizeof(double) + sizeof(void *);
if(size1 == size2)
```

```
        printf("No padding is done");
    else
        printf("%d bytes used for padding",size1- size2);
// in my system it printed,
// 7 bytes used for padding
```

6.10 Portability Issues

The alignment and buffering of structure members in a structure are machine dependent. Therefore, writing the structure information to a file in one machine and reading that file in another machine may not work even if you use the same structure. So, transferring binary files is not portable. However, these problems are not there in transferring the text files and so it is better to store and transfer the information as a text file although some extra overhead may be there.

You can assign one structure to another structure.

Since you cannot use == operator to check the equality between the two structures, you have to write a separate function that will check for equality of each structure members. memcmp should not be used because there may be padding between the members (whereas memcpy can be used for copying the structure).

```
memcmp(struct1, struct2, sizeof(struct1));
        // is not correct and shouldn't be used.
memcpy(struct1, struct2, sizeof(struct1));
// ok. Because all the fields including the padding part are copied.
// But you will not require this because you can assign as
// struct1 = struct2;
```

ANSI C specifies that there should atleast be one data member in a structure.

Unions do not assure the way in which the fields are arranged and are compiler dependent. So, the program shouldn't depend on the internal arrangement of the fields. If you want to do some type conversion between two types, use casting to do the same. Don't use the unions to convert from one type to another as in the following example.

```
union value {
     int intValue;
     long longValue;
     }val;
val.intValue = 10;
long someLong = val.longValue;
   // never assign like this.
```

Here there are two types, int and long that are members of union value. To convert from int to float type use casting in assignment.

```
long someLong = (long) val.intValue;
```

is the correct way to perform the conversion. The reason is that in the previous case, you are depending on the way in which the fields of union are aligned.

6.11 The 'struct hack' Technique

In C the sizeof the structures should be known at compile-time itself and its size cannot be expanded as required. For example, to have a structure for storing computer hardware details, following details are required:

```
#define MAX_POSSIBLE     10
struct system{
     char type[10];
     char manufacturer[20];
     int  numOfPeripherals;
     int  perpheralID[MAX_POSSIBLE];
     // worst-case assumption of number of peripherals
};
```

The number of peripherals shall vary with systems and worst case size should be allocated for accommodate that. Still if more peripherals than the 'MAX_POSSIBLE' are required to be accommodated that cannot be done. So the solution as a whole is unsatisfactory.

There is a common idiom known as the "struct hack" in C community to work as a short cut for this problem of creating a structure containing a variable-size array:

```
struct system{
     char type[10];
     char manufacturer[20];
     int numOfPeripherals;
     // this field has the number of items that are pointed
     // by the following 'nameOfPeripherals' member.
     int  perpheralID[1];
};
```

And for allocation of space:

```
struct system *structPtr;
```

```
int num = 5, i;

structPtr = malloc( sizeof(struct system) + (num - 1) *
sizeof(int));
// note how space is allocated for the structure

if(structPtr == NULL)
     errorHandle("Error: cannot allocate space");
structPtr->numOfPeripherals = num;

for (i = 0; i < structPtr->numOfPeripherals; i++)
     structPtr->peripheralID = getID();
```

Surprisingly this technique is known to work for almost all systems where C is implemented (because of contiguous space allocation).

7 FUNCTIONS

Take care of the means and

the end will take care of itself

- Mahatma Gandhi

Functions increase modularity, make program more readable and reduce redundant code and help in reusability. For example, the functions provided in standard header files.

7.1 General Information About Functions

Declaration of the function means introducing the function with return type and arguments.

Storage class for functions may be either static or extern. Extern is the default for functions and for function declarations it is the only storage class allowed. So,

```
void fun(int);
extern void fun(int);
// both declarations are no-different from each other.
```

Static is the only storage class specifier allowed before a function definition. This limits the function to file scope.

Nesting of functions is not allowed in C (but only extern function declarations are allowed inside function definitions).

The C language only has functions since all units return a value as opposed to the idea of procedures as in other languages (like Pascal). The return type void means that some value is returned from the function but that return value is purposefully ignored.

In K&R C the both of the functions are equivalent:

```
int fun();      // is equivalent to
int fun(...);
```

ellipsis specify that any no. of arguments(or none) will be accepted without type checking or conversions. fun() specifies that nothing is said about the function arguments. This is an example for incomplete-type. It can be later defined to take any number of arguments.

ANSI C introduces void, and so, if you want to specify that the function takes no arguments, explicitly gives it as:

```
int fun(void)
// specifies that the function takes no arguments
```

7.2 Functions as Lvalues

```
int i = 10;
int * foo()
{
puts("foo( ) is in lhs and is called");
return &i;
}
int main()
```

```
{
*foo() = 100;
// set i to 100; prints "foo() is in lhs and is called"
printf("%d", i);
// prints 100;
}
```

the function name in LHS of = operator is nothing but the variable i in disguise. This code employs the pointer concept to show that the pointers are addresses and demonstrate how powerful the manipulation on it is.

What about the following function as l-value?

```
int * foo()
{
int j;
return &j;
}
```

No compiler errors are raised. The behavior of this code is undefined. Just remember that the functions and the auto variables are allocated space in stack and are automatically removed from memory after the function returns. In this case, &j (taking addressof 'j') will give some address which is not allocated for this purpose and assignment like,

```
*foo() = 100;
```

means assigning to that unknown address. This leads to the undefined behavior of the code.

Now consider the following code segment.

```
int * foo()
{
static int j;
return &j;
}
```

This code is acceptable. The static data is allocated in the space that exists up to the end of program execution. So assignment like:

```
*foo() = 100;
 // is actually,
 j = 100;
 // where j is local to foo() and accessed outside foo()
```

is reasonable. I have seen some expert programmers using the functions as l-values like this using this idea. Remembering the previously set value is used in the case of strtok() standard library function.

To fix the idea in the mind, let us see another example:

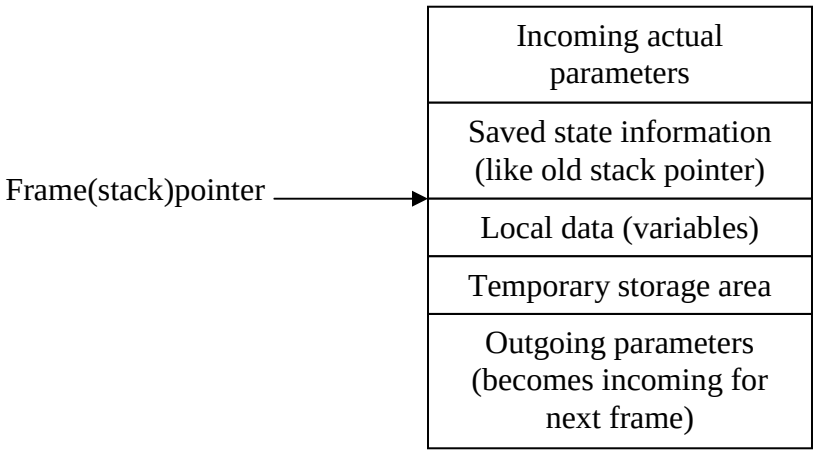
```
char *wish()
{
static char s[20]="Wake up”;
printf("%s\n",s);
return s;
}
int main()
```

```
{
strcpy(wish(),"Good morning");
wish();
}
// it prints
// Wake up
// Good morning
```

On the first-call it issues the message *Wake up* and after that it always issues the previously set message or allows you to set it to a new message.

7.3 Organization of a C activation record

The word ‘activation record’ is used in the same meaning as ‘stack frame’ here. Every function that is called is allocated memory in the stack and that is called as stack-frame.



Format of a generalized activation record

This format of generalized C activation record is adapted from [Johnson and Ritchie1981]. This format is generally followed in many implementations but may differ from implementations.

Return values are actually passed by register. The size of all the contents can be predetermined by the compiler in the compile time itself. If the arguments are variable length arguments then the size may vary only for that part of the frame. So except for the variable length arguments, the size of the activation record remains fixed.

It has to be noted that, the incoming parameters are saved by the calling function rather than in the called function's activation record. The stack frame of the called function overlaps the stack frame of the calling function to get the parameters. Similarly when the called function calls yet another function, it stores the parameter data to be passed to the next frame in its activation record and so on.

The saved information includes the control link to the calling function. Only for the activation records of same function, the size remains the same. For different functions the size may vary. So control links are required to keep track of how to return to the function that called it.

7.4 Function Parameters

When a function is called the calling position of the calling function is remembered by pushing the address in the stack. Function instance or stack frame is created on each call to a function. Local variables are allocated space in the stack area

itself. The parameters are treated as local variables. Therefore, it is legal to apply addressof operator to the parameters. But never return the address of the local variables since space is returned back for further usage and is immediately destroyed.

The function parameters are treated as local variables declared in the first level of scope inside a function (so it forces the compiler to enter a new scope before the { is encountered).

```
int foo( int i )
{
//the parameter i is treated as if it were declared with j
int j;
// it is as if:   int i, j;
}
```

The only legal storage class in function arguments is register storage class.

7.4.1 Pass By Value

The C functions support only pass by value. The copy of the data is sent to the called functions. Pass by reference is actually passing addresses by value.

```
swap(&i, &j);
// pass the addresses say 1000 and 2000.
swap(int *a, int *b)
// passes the 'addresses' by value
{
int temp = *a;
// change made to the memory locations pointed
```

```
*a = *b;
// by a and b i.e the values in the memory locations
*b =temp;
 // pointed by 1000 and 2000 are interchanged.
}
```

Since any changes made to the memory location pointed by that address, the change made is not local to that particular for that function. Thus, it successfully imitates pass by reference.

No type checking is made in passing actual arguments to functions. If the types do not agree then type conversion takes place according to the conversion rules. If the number of arguments do not agree, then last significant arguments are taken into account.

The function call,

```
add( &(i + j) , i);
```

will lead to compile time error because in pass by reference, the function expects actual addresses to be passed. To calculate (i + j) a temporary variable is created and the addresses of that variable is tried to be passed resulting in an error.

7.5 Endian problem

If your machine follows ‘small endian’ order it will be stored in the machine as 0010 in the lower byte and 0001 in the higher byte. I.e. the lower order bytes are stored in the higher addresses and vice-versa. For example, this ordering method is followed in Intel based machines. In the ‘big endian’ machines, the higher order bytes are kept in the

higher addresses itself. Examples are systems with the processors SUN's SPARC and Motorola PowerPC.

Take a two-byte integer.

```
int anInteger = 0x00010010;
//takes two bytes.
```

The figures listed shows how the bytes are organized in the memory.

Unions can be used to find the endian-ness of the machine.

```
union findEndian{
     int i;
     char c[sizeof(int)];
     }myEndian;
myEndian.i = 1;
if(myEndian.c[0] == 1)
     printf("your machine follows small endian scheme");
else
     printf("your machine follows big endian scheme");
```

By using pointers also, you can determine the byte order of your machine:

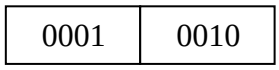
```
int x = 1;
if( (*(char *) & x) == 1)
     printf(" little-endian \n");
else
      printf(" big-endian \n");
```

Spend some time to think how it works to find the endianess of the machine.

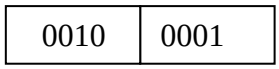
Problems may occur if the information of both the types is used in a mixed manner. Such problems that occur due to the mix between the two types of endian schemes are known in general as ‘the endian problems’.

The ‘endian problem’ may crop up in your program if you intend to make your program work between machines (say networks). For example, if you are taking the address of the parameter,

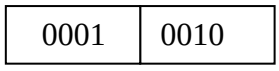
```
int transfer(int data, char *addr)
{ // don't pass addresses.
int *ptr = &data;
// This may suffer 'endian' problem.
}
```



anInteger



anInteger stored in small endian order



anInteger stored in big endian order

The solution is to declare a local variable and use the address of that local variable.

```
int transfer(int data)
{
```

```
int value = data;
int *ptr = &value;
// you can access the information like this with no problems
}
```

Exercise 7.1:

What would be the behavior of the following code segment?

```
int i = 0;
// sizeof(int) == 2
scanf("%c", &i);
// input 0 here
printf("%d", i);
```

7.6 Function Declaration/ Prototypes

Since functions by default have external linkage, if the function is not already defined, the match between the formal and actual parameters go unchecked. There is lot of scope that the programmer makes mistake by calling the function with wrong argument type. This can be costly since the function executing with wrong arguments may even lead crashing the system.

To avoid this, functions can be declared by using the prototypes. This prototype feature is added in ANSI C following the idea from C++. Prototypes are just an indication to the compiler about the name of a function, its argument and its return type. This helps the compiler to make strict type checking when the function calls are made.

To support prototypes, the compiler requires only some little more effort and no code is generated. So there is no overhead involved in writing prototypes. Do write prototypes whenever necessary and possible.

Function declaration with no parameters is used to specify the return type.

```
double foo();
// (in C) declares foo to take any number of arguments
// with return type double.
```

There is another important use of prototype: since it is a declaration, it is useful for initializing the pointers to functions before those functions are defined.

7.7 Return Values

The return type also is by value. When the function encounters return statement it returns to the calling position of the calling function. The default return type is int (if return type is not specified). The value is normally returned via registers.

Consider the example,

```
 // file-1
double foo( )
{ return 1.0 };


 // file-2
int i= (int) foo( );
// may not assign i=1; since foo is not declared in file-2 it is
// assumed to be of return type int. So even if it is type casted it
// doesn't work as expected.
```

So care so should be taken to see that functions are properly declared and return types are matched.

Return statement without expression,

i.e. `return ;`

is used in the functions to return the control to the calling function. If the return type for a function is void, then it is more meaningful to have such construct. For functions with other return types, the function may return unpredictable values.

So it is always good to make sure that the actual parameters and return types are matched with formal parameters and return types.

7.8 The C argument passing mechanism

The standard specifies nothing about the mechanism of argument passing in C. Normally the arguments are pushed into the stack before the function is called. The effect is that the arguments are passed from right to left. For example:

```
int add(int op1,int op2,int op3)
{
 return (opt1+10*opt2+20*opt3);
}
```

when this function add is called as:

```
add(10,20,30);
```

the arguments 10,20,30 are pushed into the stack one by one. Then the arguments are popped up from the stack getting 30,20,10.

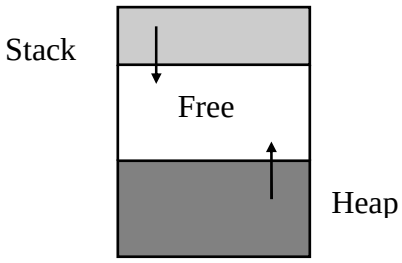
If you want to examine the parameters by yourself and understand that they are really passed through stack, try this program:

```
add(int a, int b, int c)
{
int *ptr = &a;
int *i;
int numOfArguments=3;
printf("The arguments are ");
for (i = ptr; i < ptr + numOfArguments; i++)
     printf("%d ",*i);
}
```

If you want to see the call stack (the stack in which the return addresses of the previously called functions are stored) decrement i in the above program. To verify that they really are the previously called functions try printing the locations which will contain the addresses of functions, together with the registers, pc values etc. by your own.

C is one of the few languages that require a runtime stack and a heap. Stacks normally grow from top to bottom and heaps from bottom to top as shown in the diagram (which is a generalized one). (ANSI C requires a downward-growing stack) Both the stack and heaps share the common space and are uninitialized and that's why it is asserted that it is the duty of the programmer to properly initialize it. It is a common design to have the design of heap and stack growing towards each other. This is because in most of the cases, either of them will be used much and so this type of organization to

helps to efficiently use the memory. The figure also shows how the code where the actual program code resides and the data area where all the static data are allocated space.




```
void foo()
{
int i, j, k, l;
printf("%p, %p, %p, %p ", &i, &j, &k, &l);
}// in our system it printed 0FFE, OFFC, OFFA, OFF8
```

Since local variables are allocated in stack frames the program shows that the addresses of the local variables are in descending order.

ANSI C does not assure anything about the order in which the arguments are evaluated.

```
someFunction("%d %d", &index, &array[index]);
```

This function call depends on the index value to be scanned first and that scanned value is used as an index in next argument (technically, this statement involves side effects). There is no assurance that 'index' will be scanned first. In this example, even if we assume that &index is evaluated first there is no assurance that value of index will be successfully scanned. To see how it practically applies, consider again the example of swapping two values.

```
int a = 10;
```

```
int b = 20;
b = func(a=b, a);
```

where func is defined as,

```
int func(int x, int y)
{
return y;
}
```

The idea is to use the order of evaluation of the function arguments. Function arguments are evaluated and pushed from right to left. So, func() sends value of ‘a’ and remembered inside the function and goes on being assigned by ‘b’. The func() returns the remembered value of ‘a’ which is in turn assigned to ‘b’, in effect interchanging the values of ‘a’ and ‘b’

This solution suffers from the problem that the finding of values depend on the order of evaluation of the function arguments.

7.8.1 Passing arguments using registers

Interestingly enough passing the arguments by specifying it as register variable allows it to be passed through the machine registers rather than the stack,

```
int process(register int i, register int j);
// the arguments are passed using registers than using stack
```

Since the arguments are register variables, this helps to improve the efficiency also if the variables iand j are accessed very frequently.

It should also be noted that, argument passing for functions is similar to initialization. For example, usual arithmetic conversions are performed in both the cases.

7.8.2 C and PASCAL argument passing mechanisms

Pascal calling convention is known as standard calling convention. It is given as extensions to the language like PASCAL keyword in Borland compilers and as __stdcall in Microsoft compilers. The C calling convention is given as cdecl in Borland compilers and as __stdcall in Microsoft compilers.

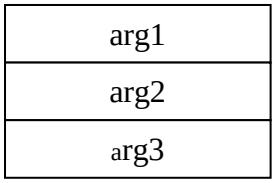
These keywords may appear only before functions. ‘cdecl’ forces the arguments be accepted in conventional C style. Pascal keyword specifies that the arguments are passed the other way. For example:

```
add(1,2);
```

here ‘cdecl’ is assumed and the number of operands does not match. Since the values are popped in the reverse direction, the return value is 50. If it is Pascal type function be called the return value will be 21.

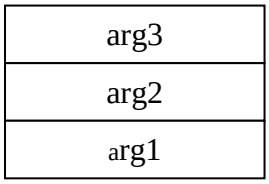
Understanding this difference can help you understand many interesting nuances about the argument passing mechanism.

```
E.g. pascal fun1(int arg1,arg2,arg3);
```



stack frame model for fun1

E.g. cdecl fun2(int arg1,arg2,arg3);



stack frame model for fun2

C	PASCAL
Pushes the arguments in the reverse order than it is listed in the function	Pushes the arguments in the stack as listed in the function declaration
The first argument in the function declaration is at the top of the stack	The last argument is at the top of the stack
It requires extra code generated by the compiler in the execution to facilitate this convention	No extra code is needed to be generated by the compiler in this convention
The stack is cleared by the called function	The stack is cleared by the calling function
It is possible to have variable number of arguments due to this mechanism	It is possible to have variable number of arguments

When the function follows Pascal convention, it is the responsibility of the functions to remove the arguments from the stack, before they return from the caller. Whereas, in the ‘cdecl’, the calling function is responsible for cleaning up the stack. This is one of the main differences between the Pascal calling and C calling convention.

Most of the languages follow the standard (Pascal) calling convention (like Visual Basic and APIs such as Win32). This is because it reduces the size of the code generated. On the other hand, the cdecl allows the variable argument lists to be implemented.

The differences between the two calling conventions are summarized in the table given.

7.9 Recursion

Functions can be called in C recursively, either direct or indirect. For each call a new stack frame is created. This makes them independent of automatic variables of previous set.

The following is a simple program to check for well formed parenthesis that applies recursion and introduces the use of functions like advance() and match() that are normally used in lexical analysers that is discussed later.

```
static char currentToken;
char *inputString= "(()())";
char advance()
{
```

```
return (currentToken = *inputString++);
}
void match(char token)
{
if(currentToken != token)
        printf("\nError : missing matching ')'");
else
        advance();
}
void parens()
{
 while(currentToken=='(')
 {
        advance();
        parens();
        match(')');
 }
}
int main()
{
 advance();
 parens();
 if(currentToken == ')')
```

```
            printf("\nError : ')' without matching '('");
    }
```

Recursion is very powerful in modeling the real-world problems. The following example is about solving the n-queens problem using recursion (this technique is called back-tracking algorithm).

```
    // the power of recursion
    #include<math.h>
    #include<stdio.h>
    #include<stdlib.h>


    int x[9];
    int num=0;
    int xyz=0;
    int place(int k, int i)
    {
     for(int j=1; j<=k-1; j++)
         {
         if( (x[j]==i) || (abs(x[j]-i) == abs(j-k)) )
                    return 0;
         }
     return 1;
    }
    int nQueens(int k,int n)
```

```
{
 for(int i=1; i<=n; i = i + 1)
     {
     if(place(k,i))
          {
          x[k] = i;
          if(k==n)
               {
               printf("%6d",num);
               for(int j=1; j<=n; j++)
                    printf("%5d",x[j]);
               printf("\n");
               num++;
          }
       else
            nQueens(k+1,n);
          }
     }
 }
int main()
{
 nQueens(1,8);
 // solve for 8 queens
```

```
 printf("\n %d combinations",num);
 // it prints 92
}
```

Point to Ponder:

For all recursive programs, there exists an equivalent iterative program.

Exercise 7.2:

Write a recursive function to reverse a string. The solution may seem simple and direct. However, finding a very good solution to this problem can be challenging (and interesting!)

7.10 Function Pointers

Functions, just like data types, are declared and defined. The defined function is available in the memory. So we can take address of it and store it in a function pointer.

In other words, a pointer to a function is best thought of as an address, usually in a code segment, where that function's executable code is stored; that is, the address to which control is transferred when that function is called.

```
int (*fooPtr)( );
```

Here fooPtr is a function pointer. It can be assigned with any function having no arguments with return type int.

```
int foo( );
```

Let be a function. It can be assigned to fooPtr as,

```
fooPtr = foo;
// or
fooPtr = &foo;
// both are equivalent
```

The pointer can be used later to call that corresponding function in two ways.

```
fooPtr( );
// or
(*fooPtr)( );
```

Both the ways are equivalent (ANSI standard). I.e. (*) is optional.

For example the * is to indirect the function pointer and the surrounding parenthesis is to force that indirection.

So it comes out that,

```
fooPtr( );      // or
(fooPtr)();     // or
(*fooPtr)( );   // or
(***fooPtr)(); // or more... are all equivalent!!!
// more than one indirection cannot change the address
// where the function is available, so valid.
```

Similarly due to this the syntax of function pointers, the following is valid:

```
int fun();
// declare fun() as function returning int.
(&fun)();
// call it using the & operator to function name
```

because &fun is nothing but the address of the function and so (&fun)()is same as fun() or (fun)().

In other words it can be viewed as if functions are always called via pointers, and that "real" function names always decay implicitly into pointers (The same reasoning that applies to the equivalence of a[2], 2[a], *(2+a) and *(a+2). "... a reference to an array is converted by the compiler to a pointer to the beginning of the array" [Kernighan and Ritchie 1988]) and so unambiguous and valid.

This equivalence is true except inside sizeof.

```
int foo();
sizeof(foo);
// error, because foo is a function name here inside a sizeof
// and you cannot apply sizeof to find the sizeof a function
// it should be, sizeof(&foo); to find the sizeof the
// function pointer
```

The point to remember is that the return type and arguments must be identical for the function pointers and the function assigned.

```
int (*fp)(float );
// fp is a pointer to functions that take float as
// argument and return the value of type int.
int foo1(float f1) { return 0;}
fp = foo1;
// O.K. the argument and return types matches.
extern foo2();
fp = foo2;
```

```
// Erroneous. There is no way by which the compiler can verify the
// types. If the return type or argument types differ for foo2 then
// undefined behavior.
```

This is a very crucial point to remember because most of the times the compilers cannot flag error by noting the mismatch. This is because many of the assignments are done at runtime and the type information may not be available to the compiler.

7.11 Polymorphic Routines

Function pointer is a powerful mechanism that it can provide polymorphic routines. *bsearch* and *qsort* (defined in <stdlib.h>) are examples for such functions.

```
void qsort(void *base, size_t n, size_t size, int
(*cmp)(const void *,const void *))
```

This sorts the array base having n elements of each size in ascending order. This provides sorting function for any type of objects including structures. The comparison function of each type should be passed and the appropriate size should be given. The same applies to bsearch.

The following example is to show the real world example of using function pointers. Say you want to write a menu program. The aim is to write a program that will call corresponding function is selected in the menu at runtime. Therefore, the requirement is to write declaration a function pointer with int as common return type. It may be declared as:

```
int (*fnPtr)( );
switch(select) {
```

```
case NEW  : fnPtr = & new( ); break;
case OPEN : fnPtr = & new( ); break;
…..
 // assign the address of the corresponding function to fnPtr.
}
fnPtr( );
```

This is an easy example, but it shows they apply in real programming and it is essential to be able to understand and compose such declarations. Calling of functions using these function pointers whose value is determined at runtime is known as 'call back'.

7.12 Library Functions

In writing code for library functions, frequently called functions or in critical applications, care should be taken and problems may crop up even in unexpected places.

Consider the example of comparing two integers (analogous to strcmp). It may be written as,

```
int intCmp(int x, int y){
     return (x-y);
}
```

This solution may suffer a problem. If x is a big positive integer and y a big negative one then x-y may lead to overflow and so will give incorrect output (e. g take 32000 and -32000, assume sizeof(int) ==2).

So individual comparison be made and rewritten accordingly.

```
int intCmp(int x, int y){
     if ( x < y )  return -1;
     else if ( x == y ) return 0;
     else return 1;
} // no chance of overflow in this case.
```

7.13 Design of interface

Return type and arguments of a function is called as the interface of the function. It is called as an interface because these are the actual parts that form the window to the use that function. Functions abstract the implementation of the function as the user is interested only in using it. Since functions you write may be frequently used, the interface design should be taken care. It is a good methodology to make interfaces straightforward and there should be no ‘surprises’ by using the functions.

It is also a not good design to modify the values passed to the functions without explicitly evident that the change is made. An unusual example of how a function should not be designed is from the standard library itself.

Consider the function strtok(). It is used to separate the tokens from the passed string. If the first argument is not NULL the pointer value is *remembered* inside the strtok function to be used in the subsequent calls. In following calls strtok if first argument as NULL, the tokens from previously remembered string are returned. strtok() returns NULL when it reaches end of the string. Also when strtok() is called it replaces the end of token in the passed string with a NULL character.

```
char statement[] = “This is about interface design”;
```

```
char seperator[] = " \n";
     // blank space and newline are separators
char *aToken;
aToken = strtok(statement, separator);
while(aToken = strtok(NULL, separator)
     puts(aToken);
```

In this example, at first the strtok gets the address of ‘statement’ in the first call. In the subsequent calls, it is called with first argument as NULL, so it returns tokens from the ’statement’ separated by any of the ‘separator’. At end the original string looks like this,

```
"This\0 is\0 about\0 interface\0 design"
```

in which the original string is modified as a side effect of the, main effect of returning a token from the string (so that the subsequent calls will return the consecutive calls). So it is necessary for the users to know about the effect of strtok and so he must pass the copy of the original string rather than the original string itself, which is an undesirable property.

7.14 Variable Arguments

Variable arguments in C are implemented using the three *macros* defined in <stdarg.h>, va_start, va_arg and va_end. va_list is used to declare variable to access the list. Examples for variable argument functions are the ubiquitous scanf and printf.

Variable for variable argument list should be declared to access the arguments of type va_list. Then va_start initializes that list and arguments are accessed one by one by calling va_arg repeatedly.

```
type va_arg(va_list, type)
      // where type indicates data type.
```

Before getting the next argument, it is necessary to know the argument type (in printf/scanf this is achieved by information in format string).

The accessing ends with calling va_end.

The limitations of variable argument mechanism are,

- it is implemented as macros itself is a disadvantage (for e.g. macros does not know the argument type. So default conversions will be made on the arguments passed).
- it can be accessed only sequentially (it means that you cannot access the argument in the middle directly).
- type has to be predicted before accessing the next argument.
- it should have at-least one named argument

```
#include<stdarg.h>
#include<stdio.h>
      // a small version of printf
      // supports %s and %d only and escape sequence \t
int myPrint(char *format,...){
 va_list printList;
 va_start(printList,format);
```

```
// variable argument lists depend on the assumption that
// the arguments are passed in a contiguous memory to the function
// called. va_start gets the initial position and va_list increments
// to the next argument. This is done by incrementing the pointer to
// the next location.
for(char *p = format; *p != '\0'; p++){
 if(*p=='%')
     switch(*++p){
          case 'd': {
                     int iVal = va_arg(printList,int);
                     printf("%d",iVal);
                    } break;
          case 's':  {
                   char *sVal=va_arg(printList,char *);
                     printf("%s",sVal);
                    } break;
     default   :  printf("not supported yet");
     }
 else if(*p=='\\') {
     // escape sequence
     switch(*++p) {
     case 't':   putchar('\t'); break;
     default   :  printf("not supported yet");
     }
```

```
 else
     putchar(*p);
 }
 va_end(printList);
}
int main(){
  int i=10;
  myPrint("format %d %s",109,"sriram");
}
```

7.15 Wrapper Functions

Procedural programming languages are vulnerable to the changes in the interface. Consider the example of a student structure,

```
int addStudent(char *name, int typeOfEntry);
```

If the second argument typeOfEntry is not necessary then the declaration becomes,

```
int addStudent(char *name);
```

This means that all the code that is calling the function addStudent needs to be modified to take single argument (this problem of changing the interface can be easily solved in object oriented languages by function overloading, default arguments etc.). Here the wrapper functions may be used.

```
int addStudent(char *name, int typeOfEntry){
 // ignore typeOfEntry that is useless.
```

```
 int addStudentWrapped(name);
}
```

The idea is to wrap the new function into the old one such that the interface remains the same. The old code remains unaffected. Similarly, let us consider that the new structure

```
typedef struct {
     char *name;
     int typeOfEntry;
}student;
```

is defined. Now the addStudent function is changed to,

```
int addStudent(student *);
```

In this case also wrapper function can be used without affecting the legacy code,

```
wrapperAddStudent(student *stud){
 addStudent(stud->name, stud->typeOfEntry);
}
```

Thus the idea of wrapper functions can be precious in reuse of code and while maintenance is done.

8 DYNAMIC MEMORY ALLOCATION

In C functions, dynamic memory allocation functions are declared in the header file <stdlib.h> (also <alloc.h> in some implementations). If at the compilation time itself we do not know the amount of memory required we do dynamic allocation. Always allocate right amount of memory since dynamic memory allocation is not the part of the language itself. If it were part of the language there will be some support to allocate the right amount of memory but since this is not allocating right amount of memory should be taken care by the programmer.

8.1 Memory Allocator

The functions like malloc(), realloc() are implemented minimally dependent on operating system or underlying hardware. There is a memory allocator that will manage the services required by allocation functions, by acquiring a very large chunk of memory from the operating system. Then it fragments the memory, keeps track of allocated parts and deallocates as required. If the allocated block of memory is exhausted, the memory manager requests for another block from the OS and continues functioning. Interested readers are recommended to read one such implementation of a dynamic memory allocation manager program for UNIX operating system that is described in [Kernighan and Ritchie 1988]. If either memory allocator fails to allocate

memory (due to excess fragmentation or any other reason) or if OS fails to provide memory it silently returns NULL. So, always check if memory is properly allocated.

The following simple program may be used to determine your heap size.

```
int main(){
const int SIZE = 1024;
int kbs=0;
while(malloc(SIZE))
     kbs++;
printf("  %d  kilobytes  of  memory  can  be  allocated dynamically in your system", kbs);
}
```

Never assume about the size of the data type when allocating memory for various data types. For example say you want to allocate memory for 10 integers. It is portable to use malloc (sizeof(int)*10), to malloc(2*10) where you assume that an integer is of size 2 bytes.

8.1.1 Memory Size Allocated by malloc()

It is a false notion to believe that malloc allocates exact amount of memory you request. In practice it may allocate more (or even nothing, if no memory is available by returning NULL). Say our requirement is to have a linked list of individual characters, you may declare a structure like this,

```
struct node{
     char data;
```

```
        struct node * next;
    }*node;

node=(struct node *)malloc(sizeof(struct node));
// doesn't allocate exactly 3 bytes, but more.
// in our system it took 8 bytes
```

and so may think that it takes approximately three bytes (assuming two bytes for pointer). But the memory allocator in C is implemented such that it allocates memory only in multiples of blocks of predetermined size (say 8 bytes or 16 bytes). This is to avoid minute fragmentation and to improve the access speed.

So it comes out that dynamic allocation is not suitable for applications where very small chunks of memory are needed. The solution to this problem may be,

(1) Predict the approximate amount of memory that will be required and allocate at compile time itself.

(2) Use your own memory allocator tailored to your need (this may work out very efficiently if you have to allocate memory blocks of same size).

8.2 Initializing Memory Block

For allocating large blocks, the available allocation mechanism suites very well. It is also the duty of the programmer to properly initialize the space allocated dynamically. calloc() function returns the memory (all initialized to zero) so may be handy to you if you want to make sure that the memory is properly initialized. calloc is internally malloc with facility to initialize the memory block allocated to zero.

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

Here the memset() function is employed to initialize the allocated block. This function is very useful and handy to initialize large block of memory without the need to traverse the whole array. A point to note here is that the second argument (the value to initialize) is a character. So for initializing floats or doubles the same old technique of traversing and initializing should be employed.

Similarly, the blocks of memory can be copied using memcpy() and memmove() functions (memset and these two functions are declared in <string.h>). The only difference between these two functions is that memmove() can be used with overlapping memory area, whereas memcpy() for non-overlapping memory areas (of course with some loss of efficiency).

8.3 void * and char *

K&R C didn't have the generic pointer(void *) so it had char * as the return type for malloc. Why a char * ? Why not an int * or float * ? The reason being that characters require no padding (since chars are assured to be one byte in length nothing is required to pad and *padding is one of the main reasons why casting should be done between pointer types*). So it is as if you are accessing individual bytes and so served the purpose well.

Since pointers of type void can be assigned to any type without casting you can use malloc without any casting.

```
int *array = malloc(sizeof(int)*10);
```

since ANSI C says that malloc returns void *. But the old function (char *) malloc() needed to be casted before being assigned to other pointer types. It is the matter of taste to select between the two methods to cast explicitly or not

Exercise 8.1 :

You know that in some machines certain types have to meet the alignment requirements, for example, ints and floats should start at the even addresses. This will be taken care by the compiler when allocating memory for such types declared in the programs. How will you take care of the problem of aligning the types to required boundaries when you allocate memory explicitly by using dynamic memory allocation (say malloc)?

8.4 realloc() (a complete memory manager)

```
void *realloc(void *ptr, size_t size );
```

- if ptr is NULL then it is same as malloc(size) is called.
- changes the area of memory pointed by ptr to size bytes.
- if the space can be continuously allocated after ptr, the memory size is extended and the same ptr is returned. If the space is not available continuously after ptr it is checked if the size of memory can be allocated somewhere else. If so, the data pointed by ptr is copied byte by byte to the

new location and the new address is returned. If size bytes are not available NULL is returned.

- it always succeeds in shrinking a block.
- if size is 0 and ptr is not NULL then it acts like free(ptr) (and always returns NULL).

```
char *ptr=NULL;
ptr = realloc(ptr,100);
/* ptr is NULL. It now acts as if malloc(100); is given */
if(ptr == NULL)      // may or may not succeed
     printf("Error: cannot allocate memory");

ptr = realloc(ptr, 250);
/* extends the block (of size 100) currently pointed by ptr to size of 200 bytes if possible. Else it allocates 250 bytes somewhere else, preserves old block contents upto 100 bytes and returns new pointer */
if(ptr == NULL)      // may or may not succeed
     printf("Error: cannot allocate memory");

ptr = realloc(ptr,100);
/* shrink the memory block by 150 bytes keeping only first 100 bytes. Always succeeds. So not necessary to verify if it is succeeded or not */

ptr = realloc(ptr,0);
```

```
/* ptr != NULL and size ==0. So it acts as it free(ptr) is called. */
```

Due to its dynamic behavior as free, malloc, and realloc depending on the arguments passed it is called as ‘a complete memory manager’. It is wonderful to see that it behaves like this depending on the arguments passed. It is meaningful and gives the power. However, do not do this in your programs. This approach is suitable for library functions. You cannot expect users of your users be aware of such behavior by your functions depending on the arguments passed. If you have to write such function provide four functions, each for one behavior (like shrinkMem, extendMem etc.). This will make your code more readable and allow your users to select the function depending on the functionality required.

Consider a problem like a string that may require growing string at runtime by concatenating it. realloc() comes here handy for implementing this growing arrays. You pass the pointer to the already available string(p1) and the string to be concatenated(p2).

```
const int SIZE = 100;
char *addStr(char *p1, char *p2){
 if(p1 == NULL)
      {
      p1 = malloc (strlen (p2) + 1);
      if(p1 == NULL)   // if malloc() fails
           return NULL;
      }
 else  // if previously allocated some memory strcat it
```

```
        {
        p1 = realloc(p1, strlen(p1) + strlen(p2) + 1);
        if(p1 == NULL)       // if realloc() fails
             return NULL;
        strcat(p1, p2);
        }
    return p1;
   }
```

A subtle problem is there in the realloc it to source. If realloc fails then source will be set to NULL so the pointer to previously allocated string will be lost. So the modified version is,

```
    …
   {
   char *temp;
   temp = realloc(source, strlen(source)+
strlen(target)+1);
   if(temp == NULL)      // if realloc() fails
         return -1;
   source = temp;
   strcat(source, target);
   }
    …
```

By introducing a new temporary variable temp, the original array is preserved even if realloc fails.

Since there is close relationship between pointers and arrays, the memory allocated from dynamic memory can be used as if it is a statically allocated array.

```
for(i=0;i<10;i++)
      array[i]=0;
// no difference between the static and dynamically allocated
// array access
```

8.5 free()

Not freeing the dynamically allocated memory after use may lead to serious problems. Read about memory leaks in this chapter.

free() assumes that the argument given is a pointer to the memory that is to be freed and performs no check to verify that memory has already been allocated. Freeing the unallocated memory will lead to undefined behavior. Similarly if free() is called with invalid argument that may collapse the memory management mechanism. So always make sure that free() is called with only valid argument.

Consider the example of freeing a linked list [Kernighan and Ritchie 1988],

```
for( ptr = head ; ptr != NULL ; ptr = ptr -> next )
      free (ptr);
```

Here, in the expression ptr = ptr -> next, ptr is accessed after ptr is released using function free (This also serves as example for dangling pointers where the pointer is used even after freeing the block). So the behavior of this code segment is undefined.

For this problem [Kernighan and Ritchie 1988] suggests a simple solution: introduce a temporary variable to hold the address to be pointed next. The code can now be written as:

```
for( ptr = head ; ptr != NULL ; ptr = temp)
{
temp=ptr->next;
free (ptr);
}
```

Another frequently made mistake is to free a same block twice. This may occur accidentally if two pointers point an object and you call free by using both the pointers. A good point is to remember is that the pointer is not set to NULL after it is freed, and so it is our duty to make sure that this problem does not occur. Most of the compilers do not pose any problem in freeing a pointer whose value is NULL. So it is a good idea to set the pointer to NULL after freeing it and if the same pointer used in free will not cause problems.

```
int  * ptr;
{ // a block begins here
     ptr = malloc(10);
     ….
     free(ptr);
```

```
        // freeing doesn't set ptr to NULL
        ptr = NULL;
        //good practice, no problem if accidentally used in free again
    }
    free(ptr);
    // now no problem.
```

Dynamic memory allocation is like a knife. No doubt it is very a powerful tool, but that has to be used very carefully. C leaves this dynamic memory management to be controlled by the programmer and so it is the programmer's duty to use it correctly.

9 PREPROCESSOR

C is a free formatted language. Except for preprocessor. Every preprocessor statement begins with a # symbol in a separate line and so is not a free formatted one. Beginning every preprocessor directive with beginning # symbol has an advantage. It allows preprocessor to just check only the first character of every line and determine if it belongs to preprocessor or not. This helps in increasing the speed of preprocessing.

Preprocessor in C is considered as a separate logical pass. The output from preprocessor is sent to the C compiler for compilation. It should be noted that preprocessor is not a part of C language itself and is considered as a traditional utility. Preprocessing is a powerful tool that should be used with care because it can result in hard to find errors, because the code after preprocessing is not visible/transparent to the user. The code what the user sees is different from the preprocessed code that is sent to the compiler and so logically forms a layer between the user and the actual code, making it hard to debug.

Normally assemblers have preprocessing facility and C has a preprocessor due to its close association with low-level programming and assembly language programming. Its power is most often underutilized because macro-processors are more familiar to assembly language programmers than to the high-level language programmers.

The main responsibilities of the preprocessor are,

- conditional compilation,

- text replacement,
- file inclusion.

The other functions include,

- stripping comments from source code,
- processing escape sequences,
- processing of trigraph sequences,
- stringization operation,
- tackling line continuation character,
- separation of tokens etc.

9.1 Comments

Comments can be present in 'any' part of the C code where a white space is allowed. Comments are stripped from the source code and a white space is inserted in place of the comment.

[Kernighan and Ritchie 1988] specifies that the comments can not be nested. This means comment within comments are not allowed.

The attempts to comment out a line,

```
/* first one /* nested one */ continuing first one*/
```

fails, because the first `/* ends at the first */`. This leaves,

```
 continuing first one*/
```

which would generate a syntax error.

But it is a convenient for the programmers to use such comments and some compilers have the option to have nested comments. An interesting problem on nested

comments is discussed in [KOE-89]. The problem is to write a C code that would run in the compilers that support both the nested comments and normal comments and find out it is being run in such a compiler without error messages. Interesting problem indeed! I didn't feel the toughness of the problem till I tried. He gives a hint too: "a comment symbol /* inside a quoted string is just part of the string; a double quote " " inside a comment is part of the comment". The solution finally [KOE-89] give is complex but the way to arrive that solution is interesting. The solution is:

```
/* /* */ " */ " /* " /* */

// this expression is equivalent to " */ " if comments nest
// and " /* " if they don't.
```

[KOE-89] continues with the solution by Coug McIlroy, an astonishing solution:

```
/*/*0*/**/1

// This takes advantage of the "maximal munch" rule that is
// discussed on the chapter "Compiler Design in C"
```

Exercise 9.1 :

Write simple C code to strip the C style comments from the source code.

9.2 Line Continuation Character

Even tokens can be splitted across lines according to ANSI C.

E.g.
```
aContin\
    uousToken
```

This facility is also useful in defining lengthy #defines because #defines are terminated with newline characters.

```
#define errorMsg(str) printf("Compiler error at Line \
          %d File %s : %s", __LINE__,__FILE__,str);
```

This allows the macro replacement text to be typed in the next line and by the line continuation character, it is considered to be the same line.

9.3 String Substitution

String substitution can serve number of purposes where it really suits to the purpose and other alternatives (like functions, consts) are not suitable. It

- serves to improve readability,
- becomes easy for updating a program,
- can be used in constant expressions.

Even though other better alternatives to declare a constant exist like consts and enums; preprocessor statements such as,

```
#define MAX 100
```

are very common and are normally harmless to use.

```
const int MAX = 100;
```

It is better to declare constant variables like this instead of #defines because type-checking can be done by the compiler and type errors can be caught easily.

Still better version would be to declare constants using an enum,

```
enum  {MAX = 100 };
```

But enums suits better in case of closed set of constant values like the following one,

```
enum errorType{ warning, syntaxError, fatalError };
```

Hence, there are three ways for declaring constants,

- #defines,
- constant variables and
- enums

For me, selecting the way to declare a constant mostly makes no much difference and is a matter of taste.

9.4 Preprocessor Constants

There are some predefined macro constants for use in the programs.

__LINE__ : Has the current line number. If you want to print the information of in which line the program is, you can use this.

__DATE__: Replaces the string "date" where __DATE__ appears at the place of preprocessing.

__TIME__ : Replaces the text "time" where __TIME__ appears and is the time at that point of preprocessing.

__FILE__ : Defines the file name in which it appears.

__STDC__: Defined if the compiler conforms to ANSI/ISO C standard.

```
#if defined(__STDC__)
     printf("... yours is an ANSI/ISO compliant C
compiler");
     // for e. g in ANSI compliant compilers you declare
     // functions with parameters
```

```
        int demoFnDcl(int arg1, char *arg2);
    #elif
        printf("Alas ..your is not an ANSI compliant C
        compiler. Upgrade soon");
        // on other compilers(say K&R) you can declare functions
        // without parameters
        demoFnDcl();
    #endif

    // type this at the end of  the source code this is to
    // demonstrate the use of preprocessor constants.
   printf(" This file %s  has %s lines of source code ",
__FILE__, __LINE__);
    printf("  \n  Compiled  on  %s  ..  %s  ",  __DATE__,
__TIME__);
```

9.5 # (NULL Directive)

When the # symbol is given without any text serves no purpose but to increase readability.

9.6 #line

The compiler keeps track of the line numbers in the program code for indicating the compiler errors. For example, in your compiler you may get a compiler error like this:

“myprog.c”, line 100: syntax error - undefined symbol ‘z’

The preprocessor command #line helps to change the line number and the filename displayed in the error messages.

```
#line lineNumber
```

forces the __LINE__ constant to be changed to lineNumer.

```
#line lineNumber "fileName"
```

the optional "fileName" forces the __FILE__ constant to be changed to the given value.

```
#line 99 "newfile.c"
printf(stderr,"\n Error in %s %d  : Undefined symbol ",__FILE__,__LINE__);
// Output : Error in newfile.c 99 : Undefined symbol
```

forces the corresponding constants being changed to 99 an "newfile.c" respectively. It may be used in utilities like syntax analyser or a compiler to show the error messages with modified values.

Let us have another example. Assume that the following code is given in the same file.

```
#line 1 funOne       // line numbers
void funOne()              1
{                          2
int i;                     3
i = j + 10;                4
}                          5
```

```
#line 1 funTwo
void funTwo()              1
{                          2
char str[] = "something;   3
}                          4
// Here it prints the error messages as,
// "funOne", line 4,      syntax error – undefined symbol 'y'
// "funTwo", line 3, syntax error – ending " missing
```

This helps debugging better in the bigger programs. The line command forces the line number and filename maintained by the compiler to be changed.

9.7 #pragma

C allows its compilers to provide facilities that are compiler dependant, implementation dependent or machine specific. This is by the #pragma directive. For e.g Turbo C++ provides,

```
#pragma startup
```

to specify the starting function to begin execution with.

9.8 Header Files

Header files are included by using the #include directive. Purpose of using header files include,

- The definition of data structures and types,
- The declaration of functions, which use the data structures,

- The declaration of external data objects.

Following technique is popular to avoid re-including any header files in multiple file programs.

```
// this is "myheader.h"
#define  _MYHEADER_
      .... // the contents go here


// this is "actual.c"
#ifndef _MYHEADER_
     #include "myheader.h"
#endif
```

Header files can only be text files. It is better to include declarations of functions and other data structures in your own header files. It is better to keep coding part separate from header file.

One point to note about the file-inclusion:

```
#include "somefilename"
```

Here the "somefilename" is not a string literal. So the escape characters are not considered and similarly any \ inside "somefilename" need not be written as \\.

9.8.1 Nesting of #includes:

Often a declaration appearing in a header file depends on another declaration in a second header file. That is, in order for a declaration in a header to compile without error,

the compiler must have already included another header file. There are primarily two ways of satisfying this requirement: shallow and deep nesting of header files.

The ‘shallow nesting’ approach forces the programmer to explicitly #include the required header files where it is used. For example:

```
// contents of first.h
struct s1 { ... };


// contents of second.h
struct s2 { struct s1 someS1; };
// note here that this requires the first.h be for using this


// contents of  myprog.c
#include “first.h”
#include “second.h”
// this inclusion sequence is a must
struct s2 someS2;
// O.K. No problem
```

As you can see, the inclusion of the file “second.h” requires that “first.h” is already included. This is the evident disadvantage of this approach.

On the other hand, in “deep nesting” approach, automates much of the work of #including by automatically including “first.h” in any file that #includes “second.h”. Look at the code:

```
// contents of first.h
struct s1 { ... };
```

```
// contents of second.h
#include "first.h"
struct s2 { struct s1 someS1; };
```

This kind of nesting relieves the programmer from the burden of manual inclusion. This approach is preferable when an *entire* header file must be processed to enable the compilation of a second header. But they have a problem. Consider the code:

```
// contents of common.h
struct s1 { ... };

// contents of first.h
#include "common.h"

// contents of second.h
 #include "common.h"

// contents of  myprog.c
#include "first.h"
#include "second.h"
// problem of multiple definition at this point
```

So this may lead for possible redefinition errors to occur (of course, this problem can be avoided by having the conditional compilation to avoid multiple inclusion of files).

It is not always possible to follow the same approach and so, depending on the

situation, the nesting approach should be chosen. In general try to use forward declarations to avoid nesting of headers (but in the example discussed this is not possible)

9.8.2 Circular #includes

Another interesting problem arises when you try to include one header file in another when forward declarations are not possible. Consider the code:

```
// contents of  first.h
#include "second.h"
struct s1 {
     struct s2 someS2;
};
// contents of  second.h
#include "first.h"
struct s2 {
     struct s1 someS1;
};
// error: "#include nesting level is deep : possible infinite
// recursion"
```

It is illegal to directly or indirectly self-compose a structure; so such circular #includes are not possible.

9.8.3 Precompiled header files

In most of the programs, lot of standard header files and other header files are included. The compiler parses the contents of the header files and the information is included in the symbol tables, occupying space. In case of projects that are linked from various files, there is much chance that the header files are reincluded and thus processed again by the compiler again redoing the processing already done. Thus, most of the compiler's time is spent like this, care should be taken not to reinclude any header files (we just saw techniques for avoiding such reinclusion of header files).

Some compilers (for instance Borland compilers) provide an option for handling this problem in a different way automatically is through procompiled header files. Symbol tables corresponding to the parse of header files when they are compiled for the first time are entered and stored into the disk as files (possibly with .SYM extension). The next time the header file is included by any other files the information from the corresponding file already stored is loaded and used. This greatly improves the speed of the compilation of the re-included header files. Using this compiler facility will not affect portability in any way. See your compiler documentation for more information about precompiled header files.

9.9 Macro Processing

Consider the following example,

```
foo()
{
```

```
        return 1;
}
#define foo( ) return 1;
int main()
{
        foo();    // this is replaced by return 1;
        (foo)(); // this calls the foo() function;
}
```

Preprocessor does not have a separate namespace. It just operates on code before compiler operates on it. So the preprocessor tokens and the program text share the same name space and the above example points out the problem due to this fact. This is one of the reasons why using preprocessor may lead to hard to find errors.

A macro table is constructed internally for processing macros. The macro replacement text is stored without expanding it with arguments (the effect is that any errors in macro expansions are reported only if macros are called in the source text).

It is valid to #if check with or without macro arguments.

```
        #if defined(getchar)
        #if defined(getchar(ch))
```

are equivalent.

Macros within macros are allowed and are expanded accordingly.

```
#define macro1(str) macro2(str)
#define macro2(str)       puts(str)
```

is one such example of macro within a macro.

Beware that we can redefine the preprocessor identifiers. For example,

```
# define something "first"
int main(){
     # define something "second"
     puts(something);
}
```

It prints 'second'. However, it should be noted that the preprocessor is a naive tool. So scope rules don't apply to it.

9.10 Conditional Compilation

The use of preprocessor is not only limited to including files and macros. However, the inevitable use of preprocessor is conditional compilation.

Conditional compilation is useful making the code portable. For example:

```
# if defined (__STDC__)
     printf("This is complied by an ANSI C compiler");
#else
     printf("This is not compiled by an ANSI C
complier");
#endif
```

This also helps in eliminating the redundant declarative code. What will happen if a header file is included in the program like this? Will the code be included twice?

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

No. It will include the required information only once since conditional inclusion is made.

The header files normally have a guard like this,

```
// inside <stdio.h>
# if !defined(__STDIO_H)
      // this is how contents of <stdio.h> may start with
      #define __STDIO_H
      // code for <stdio.h> goes here
...
#endif
// code for <stdio.h> ends here.
```

When the #include <stdio.h> is encountered for the first time the macro constant __STDIO_H gets defined. For the second time the preprocessor encounters #include<stdio.h>, since __STDIO_H is already defined, no code in <stdio.h> is included again. The same idea can be used for the header files written by the users for guarding against re-including them.

Conditional compilation removes parts of source code logically. Conditional compilation can be used in places where we want to have dependent information to be included.

The following example demonstrates how the code can be used at the development phase to include debugging information.

```
// sample program to demonstrate the use of conditional compilation.
// put #define  TEST here to make it a testing program
#if defined (TEST)
```

```
        testPrint(str1,str2) printf(str1,str2)
    #else
        testPrint(str1,str2)
    #endif
    ......
    #if defined(TEST)
    int main( ){
    .....
    }
    #end if
```

Constant expressions used in preprocessor directives are known as macro expressions and are subject to additional restrictions (known as “restricted constant expressions”) than ordinary constant expressions. A restricted constant expression cannot contain sizeof expressions, enumeration constants, type casts to any type, or floating-type constants.

sizeof operator cannot be used in macro expressions because the preprocessor is a naïve one and it cannot recognize (parse) typenames. In addition to that, sizeof is a compile time operator and sophistication (one such that is available in the semantic analyzer) is needed to find out the size of the type or the size required for storing the result of the expression.

So, macro expressions are not ordinary expressions. Always be cautious while using macro expressions to avoid ‘surprises’. Look at the following example,

```
    #undef SOMETHING
```

```
#if SOMETHING != 0
     printf(" This is what I expected too ");
#else
     printf(" That's C ");
#endif
```

it always prints " That's C ".

Conventionally all the variables occurring in the preprocessor expressions are expected to be predefined. The C preprocessor assumes all the undefined preprocessor constants as equal to 0 (signifying that it is undefined).

Let us look at another example.

In practical programming, forgetting to include header files may result in subtle errors. For e.g. if you forget to include <limits.h>, then the following condition becomes always false and thus never executed.

```
#if   INT_MAX > 32767
     // this will never be encountered if<limits.h> is not included
     // since  INT_MAX is assumed to have value 0
#endif
```

The 'assert' macro (defined in <assert.h>) is a fine example of how you can use macros efficiently (assert macro is discussed elabotrately in the chapter 'operators, expressions and control flow').

While testing the programs where conditional compilation is done, all the alternative paths in the code has to be tested.

9.11 defined Operator

defined is a special operator in preprocessor which can be used after #if or #elif.

#ifdef (MYDEF) is equivalent to `#if defined(MYDEF)`. The defined operator returns either 1 or 0 based on if the identifier is defined or not.

With defined operator you can check like this

```
#if defined(VAR) && !defined(CONSTANT)
```

which makes code crisp and to the point and the above code cannot be given in a single #ifdef statement and will require multiple #ifdefs to achieve the same result.

9.12 Macros Vs Functions

The main differences between the macros and functions circle around the following four points:

- Type-checking,
- Speed versus size,
- Function evaluation, (a function evaluates to an address; a macro does not)
- Macro side effects.

Macro expansion is not just text replacement with arguments; it is C's version of pass by name functions. Since the preprocessor does it, macros do not know types and so indirectly gives 'type independent functions'. This is a facility to be used cautiously.

Macros also mimics 'inline functions' which is compile-time operation and also enables crisp code and faster operation since there is no overload of creating and destroying stack frames (i.e. no overhead of function calls is involved). This performance

gain was valuable at the time when C was designed and the machines were slow. Heavy use of macros in standard header files demonstrates this fact. But this notion is losing ground today because, modern machines are considerably efficient and fast and the extra overhead of calling function is minimal compared to the disadvantages of using macros.

Consider a very simple example of writing a macro for swapping two values.

```
#define swap(x,y) { int temp = x; x = y; y = temp; }
```

it works for calls like,

```
int a =10;
int b = 20;
swap(a,b);
```

but what about this,

```
int j = 1;
int a[] = {1,2,3};
swap(j,a[j]);
printf("j = %d , a[j] = %d",j,a[j]);
// it prints "j = 2 , a[j] = 1"
```

However, there is a subtle bug in the program. It seems to work correctly, but mysteriously the array now contains the values {1,2,1}! What happened in macro replacement?

The macro replacement text is,

```
{ int temp = j; j = a[j] ; a[j] = temp; }
```

j is assigned with the expected value. The problem is with the statement a[j] = temp; Here what happens is, the value of j is modified to 2 due to the assignment j = a[j]. Now this

new value of j is used to access the value a[j] (i.e. a[2]) which is assigned to the value the value temp (which is 1). In other words the assignment a[2] = temp has taken place leading the mysterious change (in the output you print values of the modified j, so output seems that it is as expected).

This is a frequently encountered problem with macros. It cannot be generalized and sometimes for unexpected inputs the macros fail.

Then you may ask me to give the correct macro. For your surprise, generalized macro for swapping two values cannot be written!

Similarly the ubiquitous factorial example used to show the use of recursion cannot be written using macros, because it is textual replacement:

```
#define fact(n)      (n>1)? n*fact(n-1): n
```

does not work correctly and you can verify that.

9.13 Function Equivalent for Macros in Standard Library

Since many of library 'functions' in header files are macros, is it true that you cannot take address of those library functions?

For example:

```
char (*fp)();

fp=getchar;
```

will this result in an error?

No. ANSI C standard guarantees that there is an ‘actual function’ for each standard library function. So, there exists both macro and function for certain ‘library functions’.

When a header file declares both a function and a macro version of a routine, the macro definition takes precedence, because it always appears after the function declaration. When you invoke a routine that is implemented as both a function and a macro, you can force the compiler to use the function version in two ways:

- Enclose the ‘function’ name in parentheses.

```
a = toupper(a);
//use macro version of toupper
a = (toupper)(a);
//force compiler to use function version of toupper
```

- “Undefine” the macro definition with the #undef directive:

```
#include <ctype.h>
#undef toupper
```

For comparison lets see how assert ‘macro’ can be implemented using functions.

```
void assertFun(int cond, char const *condExpr, char const
     *fileName, int lineNo)
     {
     if (!cond)
      {
      printf("Assertion failed : %s, file %s, line %d\n",
```

```
        condExpr, fileName, lineNo);
        abort();
        }
      }

#ifdef NDEBUG
     #define assert(cond) ((void) 0)
#else
     #define assert(cond) \ assertFun ((cond), #cond,
     __FILE__, __LINE__)
#endif
```

So the availability of both the macro and function equivalent makes sure that the address of that ‘function’ can be taken and passed as function pointer to other functions.

10 STANDARD HEADER FILES AND I/O

This chapter gives a quick look at what the standard header files in alphabetical order to show what they can offer. <stdio.h> is discussed in detail in this chapter because many of the functionality for standard I/O are available in this standerd header and so is important.

10.1 Standard Header Files

There are many header files supported by your compiler. For example <graphics.h>, <conio.h> are supported in Borland, Turbo C compilers which are non-standard and are platform specific for graphical and console input/output functions respectively. ANSI C gives a set of standard header files that have to be supported by every ANSI conforming compilers and using the functions is highly recommended and is assured to be portable.

This chapter deals with the details associated with these standard header files.

The header files may contain functions and function like macros. The word 'function' in this chapter may refer to both functions and macros defined in it.

10.2 Naming Convention

The names in standard header files follow naming convention as follows,

ctype all names start with is/to

errno	all names start with E
locale	all names start with LC_
signal	all names start with sa_ / SIG / SA_
string	all names start with str/mem/wcs
float	all names start with FLT_ / DBL_
limits	all names end with _MAX / _MAX

It may lead to mystifying bugs if you use any function or variable name that is same as the runtime function identifiers. Also avoid using variables starting with _ (underscore) since compiler may use them internally.

10.3 Library Function Names & Variables

The library function names share the same namespace of the ordinary variables declared and forgetting this may lead to subtle errors.

```
if(pow)
    printf("always get printed");
```

always evaluates to true. The programmer intended to use the variable name pow and since pow is the name of a library function and 'if' condition checks for its address that is always a non-zero value. In particular library function names should not be overlapped with user defined function names.

10.4 Error handling in Library Functions

Consider the following code,

```
int * p = (int *) malloc( sizeof(int) * 100);
if( p == 0)
     printf("Error : cannot allocate memory");
```

It becomes essential to check if the memory allocation has succeeded or not. malloc() returns 0 if it cannot allocate memory to indicate error. This type of error indication is better than to give runtime error and then terminate the program abnormally. Error indication is part of the return values in library functions and it is customary to use non-zero values for success and 0 for failure (as in the previous case). It is also the duty of the programmers to check (or to forget/overlook) for the possible error that may have occurred.

Returning non-zero to indicate success in is true only for C library functions. For UNIX it is the other way. There 0 indicates success (as in exit(0) for successful/normal exit)). This is because the return value is not for the use in the program. Rather it is for use by the OS possibly UNIX, so is one such exception. Another examples for indicating errors by return values is getchar(), getc() etc. that may return EOF.

getc() returns EOF when some error occurs while reading or if it cannot read from the specified file. If end-of-file is reached in the file it is reading also it returns EOF. It is left to the programmer for finding the cause of return of EOF. For example:

```
int ch = getc(someStream);
// note the type of ch is int, not char.
if(ch==EOF)
// find out if it is due to error or due to actual EOF file reached
{
```

```
        if(ferror(someStream))
        {
        fprintf(stderr,"Error reading the stream");
        clearrerr(someStream);
        // clear the error set the file pointer to read again
        }
        else
        fprintf(stdout,"End of file reached");
}
```

Indicating errors by return values poses no problem as far as the values for error and the valid values that should be returned by the function doesn't overlap.

In the example of malloc() we saw, it is easy to differentiate (no overlap) between the error value (i.e. 0) and valid return value because 0 is not a valid pointer value. But consider the following example,

```
char *str = "0";
int i = atoi(str);
// atoi returns 0 if it cannot convert the string argument to integer
if( i == 0)
        printf("Error : str cannot be converted to int");
```

Can you spot out the problem? atoi converts the string value "0" successfully to integer value 0 and returns the same. The checking condition checks for 0 and mistakes it to be an error in conversion and issues an error. In this case, overlapping of the valid

return values and the error condition is there, resulting in subtle bug/error. Another such example is getchar(), which may return EOF.

If you were to design the interface for such functions, an alternative approach can be used.

```
int myAtoi(int *iVal, char *str);
```

separating the return value which can either return true/false (1/0) and the value that is required as result is passed by reference.

The previous example has to be then modified as,

```
char *str = "0";
int i;
int flag = myAtoi(&i, str);
if( flag == 0)
     printf("Error : str cannot be converted to int");
```

this means extra effort to programmer, but works for most of the cases.

Similarly for getchar the modified form is,

```
if(myGetChar(&ch))
     // no problem
else
     // some problem in getting the input from stdin.
     // use ferror(stdin) to trace the error out
```

On the other hand the convenience of using old shortcuts like printf("%d", atoi(str)) is lost. Mostly an overhead of extra arguments is there.

The method of combining the return values and error codes stem from C tradition for efficient and compact code.

10.5 <assert.h>

This header file contains the assert macro that is for minimal support available in C language for debugging purpose. Enough has been discussed already about its use and implementations. Each of the implementation of assert macro have heir own merits and demerits. Finally lets see the following implementation of assert as function.

```
#ifndef NDEBUG
void __assertFun__(const char *cond, const char*
fileName, int lineNo);
  // this is the prototype

      #define assert(cond,fileName,lineNo)         \
      if(cond)                                     \
            {}                                     \
      else                                         \
            __assertFun__ (cond,fileName,lineNo)   \
#else
            #define assert(cond,fileName,lineNo)
#endif
```

```
// The function definition should not be included twice and so should not
// be part of the header file. Put the following function definition
// in some other convenient source file

     #ifndef NDEBUG
          void __assertFun__ (const char *cond, const char*
     fileName, int lineNo)
          {
          fflush(NULL);
          fprintf(stderr, "assertion failed: %s, file %s, \
               line %d \n",cond,fileName, lineNo);
          fflush(stderr);
          abort();
          }
     #endif
```

One point to note while using the assert statements is that there shouldn't be any side-effects involved in assert expressions. For example can you see what may go wrong with this statement?

```
assert(val++ != 0);
```

In this case the assert expression involves side-effects. So the behavior of the code becomes different in case of debug version and the release version thus leading to a subtle bug. *Don't use assert expressions that have side-effects.*

‘assert’ should not be used for error/exception handling. It should only be used for debugging and finding out bugs. Consider the following example,

```
int *foo(){
     int *s = (int *)malloc(sizeof(int) * 100);
     assert(s != NULL);
     return s;
     }
```

This is wrong. Because ‘assert’ would be disabled when code is released and so there is no way to handle the dynamic memory allocation failure at runtime. So a plain if statement checking the condition and the corresponding remedy statement has to be given.

10.6 <ctype.h>

This header file contains functions/macros for that are used for testing the characters. The functions available is highly recommended to be used instead of explicitly checking the values of the characters that are particular to a character set (say ASCII or EBCDIC) and hence usage of these functions is highly portable.

10.7 <errno.h>

The exceptions we saw in the form of signals are automatic. If an exception occurs then the corresponding handler will be called. Another approach by C is by letting

the user to check for any erroneous conditions. ‘errno’ is a global value available to the whole program. It is an integer value and is set to 0 at program start.

It may have been defined in <errno.h> as follows,

```
extern int errno;
```

Each possible error recognized by the system is assigned a unique value. The standard library functions may also set errno. The most recently occurred error would be available in the errno. It should be remembered that neither examining the errno does not reset the error condition nor any other library functions reset it to 0. So to detect an error first set errno to zero, call the library function and after that to verify that the library function executed without any problem, check its value again.

ANSI standard mandates the following three errors to be defined by the implementation.

- EDOM domain error
- ERANGE range error
- EILSEQ multibyte-sequence error

It is left for the compiler to define more errors.

For example lets see the specification for log function.

```
log:            natural logarithm function
prototype:      double log(double x);
Return Value:
On success, log returns the natural log of x
On error, if x = 0,these functions set errno to ERANGE
On error, if x is real and < 0, it sets errno to EDOM
```

```
// to demostrate how to use errno
#include<errno.h>
#include<math.h>
#include <stdio.h>

int main(void)
{
   double result;
   double x = -1;
   errno = 0;
   result = log(x);
   if(errno == EDOM)
     perror("domain error");
   else if(errno == ERANGE)
     perror("range error");
   else
     printf("The  natural  log  of  %lf  is  %lf\n",  x,
result);
        return 0;
}
// when executed in our system it printed
// domain error : math error
```

perror() is the standard library function that will send an error to the stderr.

10.8 <float.h> and <limits.h>

These two header files defines constants that specify various the numerical limits.

10.9 <locale.h>

The functions associated with locale issues.

10.10 <math.h>

Since library functions are included in thousands of source files, efficiency of the library was a driving force behind designing library functions. For example, consider the pow() library function in <math.h>. The exponentiation operation is not a part of the language (as in some other languages). The programmer has to explicitly invoke the function to perform the operation. This is because the language design decisions was made such that the features that may be implemented efficiently using runtime functions are left from language proper.

10.11 <setjmp.h>

This header file has macros setjmp and longjmp to support non-local jumps.

Setjmp saves the entire state of the computer's CPU in a buffer declared in the jmp_buf jbuf; statement and longjmp restores it exactly with the exception of the register which carries the return value from longjmp.

The stack and frame pointer registers are reset to the old values and all the local variables being used at the time of the longjmp call are going to be lost forever. Note that any pointer to memory allocated from the heap will also be lost, and you will be unable to access the data stored in the buffer. The solution is to keep a record of the buffers' addresses and free them before the call to longjmp.

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

jmp_buf jbuf;
// put it as a global variable to be called
// by longjump from any subroutine
int callSomeRoutine();
int main(){
     int val;
     val = setjmp(jbuf);
     if(val != 0){
     printf("\nError occured in one of subroutines");
     exit(1);
     }
     callSomeRoutine();
}
int callSomeRoutine(){
```

```
 printf("\nThis is from the subroutine");
 longjmp(jbuf,1);
} // it prints
  // This is from the subroutine
  // Error occured in one of subroutines
```

It is well known that, the usage of goto can be completely eliminated by using structured programming techniques, and setjmp/longjmp is a kind of goto.

10.12 <signal.h>

Signals are C's primitive solution for,

- Avoiding and handling the exceptional conditions (exception handling),
- Handling of asynchronous events (event management)

This standard header file provides the prototypes for *signal* and *raise* functions. signal() is used for installing the handler for the various signals,

```
typedef void (*fn)(int sigCode);
int signal(int sigName, fn fName);
```

where fName is,

- the installed function by the user
- SIG_DFL
- SIG_IGN

and the sigName can be any of the following,

- SIGABRT termination error (abort)
- SIGFPE floating-point error

- SIGILL bad (illegal) instruction
- SIGINT interrupt (user pressed the key combination ^C)
- SIGSEGV illegal memory access
- SIGTERM terminate program

These are the six standard signals defined by the ANSI standard. Addition to this the implementation may support more signals. The user cannot declare his/her own signals, only can have own handlers. For example:

```
void myHandler(int);
// declare signal handler function.
signal(SIGSEGV, myHandler);
// install the handler to the corresponding signal
      .....
raise(SIGSEGV)
// raises the SIGSEGV signal resulting in call to myHandler
```

Handlers are meant for recovering or resuming from exception occurred. There are predefined signal handlers available for each signal and for installing the default signal, the corresponding sigName has to be called in signal() with second argument as SIG_DFL To ignore the signal sigName install that sigName with fName value SIG_IGN.

```
signal(SIGFPE, SIG_DFL);
// installs the default handler for floating point errors.
Signal(SIGILL, SIG_IGN);
// ignore any signals due to bad instructions.
```

Note that the return type of the signal() is int. It returns pointer to the previously installed handler if it can successfully install the function, and in case of failure, it returns SIG_ERR.

```
if (signal(SIGSEGV, myHandler) == SIG_ERR)
     printf("Error : Cannot install handler to SIGSEGV");
```

Under event management, interrupts can be handled using signals. Examples for such events are hardware interrupts, and the interrupts like Ctrl C.

<signal.h> also defines the type `sig_atomic_t`. This is the type with which the exit handlers can communicate to other functions. The word 'atomic' indicates that any assignments that are made to the objects of this type are done atomically free of hindered by interrupts.

10.13 <stdarg.h>

This header file is for the support of variable argument lists that are described in the chapter on functions. It has the declaration of type va_list and the macros va_start, va_arg and va_end.

10.14 <stddef.h>

It contains declarations of types like NULL, ptrdiff_t, size_t and wchar_t.

10.15 <stdio.h>

This is the most important header file that almost all projects use because it constains functions for I/O, file management etc. This is a big header file that nearly one third of functions/macros in the standard library are from this header file.

10.15.1 I/O Functions

The C language provides no facility for I/O, leaving this job to library routines. [Ritchie et al.] illustrates one difficulty with this approach. In machines where int is not the same as long, to indicate the difference the format specifier may be writen as “%D” instead of “%d”.

“Thus, changing the type of x involves changing not only its declaration, but also other parts of the program. If I/O were built into the language, the association between the type of an expression and the format in which it is printed could be reconciled by the compiler.”

In other words, separating the I/O part from the language proper may have been a good design approach. This will make the language small and also leave it to the libraries to take care of. And this has few disadvantages. By not being part of the language, the I/O cannot take advantage of the type information available to the compiler.

```
int i;
printf(“%d”, i);
```

Here the format string is required to specify the type of the argument associated with i. If it were the part of the language this redundant information will not be required.

The problem of mismatch of number of items in format string and the arguments cannot be found out at compile time for the same reason. And also if a change is made in the type or number of arguments in the I/O routine, the change must be reflected in the format string also.

```
float fVar = 10.0;
printf("fVar = %d and sizeof(fVar) = %d ",fVar,sizeof(fVar));
// printed "fVar = 0 and sizeo(fVar) = 0" in our system
// clue: sizeof(float) = 4 and sizeof(int) = 2 in our system
```

This is an example of the problems that may arise due to the mismatch between the format string and the arguments. The first argument is a floating point variable and it takes 4 bytes. But the format string expects the first argument to be an integer (%d) and so reads two bytes from the stream (it doesn't do any typecast that you may expect). The effect is reflected in the argument read next (i.e. the sizeof(fVar)) leading to wrong output.

10.15.2 Design Considerations

The main aim behind designing the standard I/O library is the provision for efficient, extensive, portable file access facilities and easy formatting. The routines that makeup the library achieve efficiency by providing an automatic buffering mechanism, invisible to the user, which minimizes both the number of actual file accesses and the number of low-level system calls made.

FILE is a pointer to a structure used by the standard I/O to identify files. All data that are read or written pass through FILE's character buffer. For e.g. you send character by character to a file using a FILE *. You are actually writing to that FILE's character buffer and when that buffer gets filled the data is written in bulk (block) to the file intended to be written. Similarly while reading from file/any input device the data read is through the FILE's buffer and when it gets full, it will call the read primitive/system call.

These details are completely hidden from the users. This buffering makes limited use of system calls (like few calls to read/write system calls). It also avoids unnecessary manipulation directly by the user (the user need not maintain the buffer explicitly) and offers a high-level of abstraction that they can be implemented in any supported OS.

10.15.3 Pointers Associated with Files

There are three important pointers associated with files,

- file pointer,
- buffer pointer,
- file cursor.

10.15.3.1 File Pointer

It is the pointer to the FILE structure. From this pointer only the information regarding the file can be accessed. In other words, it is the address of the allocated stream buffer that is associated with a file on opening the stream.

10.15.3.2 Buffer Pointer

This is the character pointer that points to the character buffer that is internally maintained. Its only when the buffer pointer reaches the end-of-buffer the buffer is flushed.

10.15.3.3 File Cursor

It is the pointer that keeps track of the current access position of the file used. Whenever you use the functions like fread, fseek, fsetpos etc. you are actually updating the value file cursor. Similarly the fgetpos, ftell etc. return the current position of the file cursor.

10.15.4 Streams

Streams, as [Kernighan and Ritchie 1988] defines is a source or destination of data that may be associated with a disk or other peripherals. For usage stream can be considered to be file pointers. For e.g., the prototype for the fflush is

```
void fflush(FILE * stream);
```

and so may be called like this,

```
fflush(stdio);
```

for flushing the standard input which may be a keyboard.

10.15.4.1 Predefined streams

Predefined streams that are opened automatically when the program is opened are stdio, stdout, stderr (standard input, standard output, standard error and devices 0,1,2

respectively). A device (also known as file handle) is a generalized idea of file and can be treated as if it were a file. For example, say the stdio may be the keyboard. C treats the information coming from the keyboard as if it were from a text file (it even returns EOF like a file if the input is not valid!).

Practically both the stdout and stderr write to the console only:

```
fprintf(stdout,"Error : An error occurred");
fprintf(stderr,"Error : An error occurred");
```

have the same effect. But the second one is preferable to the first one for two main reasons.

Consider the case of redirecting the output to other files as input (for example: as pipes in Unix). In such cases, if the error message is written to the stdout that will go as junk input to the receiving file. So, prefer using stderr to stdout in issuing errors.

The second reason is the difference in way stdout and stderr works. The stdout writes to buffer, so it may take some time to send the output to the device, whereas the write to stderr is an un-buffered one. This means, the information written to stderr is immediately sent to the device, without being stored and waited in the buffer to be flushed.

So while writing error messages to stdout, use fflush(stdout) to write the out output fully, to make sure that the message is shown without delay. Because the possibly following abort() function makes program terminate without properly flushing and closing the streams.

```
fprintf(stdout,"Error : An error occurred");
fflush(stdout);
```

```
abort();
```

Or simply use:

```
fprintf(stderr,"Error : An error occurred");
// no necessity of fflush(stderr) because there is no bufering for
// stderr so fflush(stderr) is meaningless
abort();
```

10.15.4.2 Classification of streams

I/O streams can be classified into two types,

1) char I/O or text I/O

   E.g. getch,puts etc.

2) blockI/O

   E.g. fread, fflush, rewind etc.

10.15.5 Streams Vs File-descriptors

When you want to do input or output to a file, you have a choice of two basic mechanisms for representing the connection between your program and the file: file descriptors and streams. File descriptors are represented as objects of type int, while streams are represented as FILE * objects.

Streams provide a higher-level interface to the lower level of file handling. Internally they use the primitive, low-level operations for reading and writing of files. You can use the functions like open that return the file descriptors rather than the file pointers and are considerably low-level. That will increase the efficiency, but the

portability will be lost. The other operating systems may not support the same primitives and facilities.

However, the main advantage is that the streams provide richer, powerful facilities and formatting features in the form of numerous functions. The streams take care of lot of details like buffering internally so that the programmer does not have to bother about them. For example, the Unix OS has lot of system calls that can access and get the job done from the OS directly. But the C programmers for Unix still prefer to use the C standard library functions because of the facilities they provide.

Whereas the operating system primitives (like Unix system calls) provide only the basic facilities like block read and write and programmer have to take care of managing and keeping track of it. The operating system primitives are advantageous in cases. For example, the operations that has to be done, that are particular to a device. In such cases, streams that provide the standardized functionality are of no use.

So, the selection between the OS primitives and standard I/O functions depends on the need. To have higher portability stick to streams unless you want some direct access of special functionality. You can also open a file initially as low-level one and later can convert it to a stream.

In older versions of C, ‘long float’ was a synonym for double. So a format specifier for long float and double remains the same (you use %lf for printing the double value in printf/scanf)

Exercise 9.2 :

There were three records stored in the file pointed by fp, but the following code printed four names. Find out what went wrong?

```
while(!feof(fp))
     {
      fread(&studentRec, sizeof(studentRec), 1 , fp);
 puts(studentRec.name);
}
```

Can you guess why there is no distinct format specifier for double in printf/scanf format string, although it is one of the four basic data types?

10.15.6 Char input/output functions

The functions getchar() and putchar() get/put a character from the stdin/stdout and they are the short-cut versions of getc and putc:

```
getchar() == getc(stdin);
putchar() == putc(int,stdout);
```

Since the character input/output from the stdin/stdout is the are frequently used and using getc and putc versions that take FILE * as one of their arguments is tedious. The declarations for getc() and putc() look like:

```
int getc(FILE *);
int putc(int ,FILE *);
```

and they act to get/put a char from the specified file. They are macros. The equivalent function versions are:

```
int fgetc(FILE *);
```

```
int fputc(int ,FILE *);
```

and the functionality of the macro and the function versions remain the same. This is one of few explicit places in standard library where both the macro and function versions are provided.

As you can see, the arguments and types are ints that is counter-intuitive. There are two main reasons for this:

- to enable the characters of size more than one byte to be handled by the implementations
- to enable the value of EOF (that is normally defined as –1) to be represented in the valid range.

```
char ch;  // char may be signed/unsigned by default
while((ch=getchar())!=EOF)// if char is unsigned then problem
      // do something
```

This problem is eliminated if the return type is int and the ch is of type int,

```
int ch;   // int is signed by default and has big range
while((ch=getchar())!=EOF)     // now no problem
      // do something
```

10.15.7 getchar()

getchar() terminates only when a newline character is got. This may create problems in interactive environments if getchar() is used there for user response. Since getchar() terminates only on seeing a newline and returns only one character the remaining characters that may have been pressed shall remain in input buffer. This may

aggravate the problem. The problem may be solved by replacing getchar() by getch()/getche() (but they are not part of standard header file and are declared in <conio.h> in those implementations supporting these functions).

10.15.8 fseek() & ftell()

The functions in <stdio.h> fseek and ftell return long. This limits the file size that can be handled by these functions to LONG_MAX. To avoid this, ANSI has defined two functions fgetpos() and fsetpos() that returns fpos_t (which is a platform dependent file position type). These two functions are also defined in <stdio.h>

10.15.9 Running one Program from Another

One of the ways to run one program from another is using ‘system’ function:

```
int system(const char *commandStr);
```

where the content of the commandStr depends on the source operating system. It is normally sent as command string for the shell (command interpreter) of the underlying operating system.

The major disadvantage of the system function is that the program cannot access and use the output of the command it runs in the 'shell'. For this there are two routines provided:

```
FILE *popen(const char *commandStr, const char *type);
int pclose(FILE *stream);
```

these are non-standard ones that are available in some systems that return the file pointer for manipulating the output of the result of the execution of the 'commandStr'.

10.16 <stdlib.h>

This header file includes miscillaneous functions that includes,

- numerical conversions (atoi, strtod etc.)
- random number generations(rand, srand etc.)
- memory allocation (malloc, calloc etc.)
- program termination (abort, exit and atexit)
- interaction with OS (system and getenv)
- sort and search (qsort and bsearch respectively)
- others (abs, labs, div and ldiv)

10.16.1 exit()

exit() takes integer as argument. exit(0) indicates normal program termination and exit(1) indicates abnormal program termination. If your program is completed successfully and want to return back use exit(0) (logically to OS and User). If you encountered any abnormal condition or runtime error and want to quit prematurely use exit(1). This return value may be used by the O.S (say UNIX) to see if the program has successfully executed or not (but this value is normally ignored). exit() calls up all the exit handlers that are registered with atexit() in reverse order.

10.16.2 abort()

abort() is used in case of serious program error and thus for abnormal program termination. abort() does not return control to the calling process. By default, it terminates the current process and returns an exit code of 3. Abort doesn't call any exit handlers (atexit()) and immediately returns control to O.S. (using signals abort() can be made to call any cleanup functions). It also doesn't flush the stream buffers. It may be thought of being defined as follows,

```
void abort(void)
{
 printf(stderr,"abnormal program termination");
 raise(SIGABRT);
 exit(EXIT_FAILURE);
}
```

As you can see, there are differences between using exit() and abort() and selection should be based on the requirement. In short,

exit(int) causes atexit() to be called and other termination actions to take place. abort() causes to terminate the program abnormally and immediately without any rollup actions.

10.16.3 atexit()

You may require performing some cleanup tasks or other tasks before the program terminate. For this task atexit() function comes handy. It takes pointer to a

function as argument. The functions pointed by atexit will be called before the program termination (if any such defined and it is a normal program termination).

```
void  rollup()
{
printf("I am being called by exit()");
}
int main()
{
if(atexit(rollup) != 0)
printf("Cannot register with atexit()");
exit(1);
}
```

This prints "I am being called by exit()" because while exiting from program exit calls the functions registered with atexit().

The functions registered with atexit are also called as exit-handlers. Since exit-handlers are called for roll-up activities and resource-freeing activities, they are sometimes called as pseudo-destructors.

10.17 <string.h>

All of the functions that are listed in <string.h> rely on NULL termination of the strings. If NULL is missing then these functions will continue processing possibly corrupting memory as they go. For most of the string family of functions listed in

<string.h> you will see a corresponding ‘n’ family of functions i.e. strcpy and strncpy, stricmp and strnicmp, strcat and strncat etc. The corresponding ‘n’ functions perform the same tasks as their corresponding counter parts with the exception that they take an extra parameter ‘n’ which specifies how many characters in the string to operate on. It is strongly recommended that this ‘n’ family of functions be used rather than their counterparts if you cannot always ensure that the strings you use are properly NULL terminated.

Note: *However use of strncpy (a function in such ‘n’ family of functions) require a special caution. The problem is its inconsistent behavior. Sometimes it terminates the destination string with NULL char and sometimes it doesn’t. So don’t depend on the feature of automatic NULL termination by the strncpy. Do it explicitly by yourself.*

```
char t[10],s[10]=”something”;
strncpy(t,s,4);
// copy some into t
t[4] = ‘\0’;
// NULL terminate yourself.
```

The standard library particularly <string.h> consists of many functions that are weird looking and rarely used. For e.g. the strspn and strcspn functions.

```
size_t strspn(const char *s1, const char *s2);
```

is the prototype for strspn (string span). It returns maximum length of the initial sub-string entirely made up of characters in second one.

```
strspn(“oursystem”,”aeiou”);
```

will return 2 because the initial substring ou is made entirely of the characters in the second string “aeiou”.

strcspn is the compliment of the strspn function. It returns the span of the first substring consisting of characters not in second string.

```
strspn(“system”,”aeiou”);
```

yields 4 because the initial substring syst is made up of the characters that are not in second string.

Exercise 10.1 :

char inputString[100] = {0};

To get string input from the keyboard which one of the following is better?

1) gets(inputString)

2) fgets(inputString, sizeof(inputString), fp)

Exercise 10.2:

Which version do you prefer of the following two,

```
printf(“%s”,str);
// or the more curt one
printf(str);
```

10.18 <time.h>

It has functions for manipulating date and time. One of the many requirements for real-life programming is requirement of a delay/sleep function that waits for some

amount of time for continuing the process. That functionality is not available in the standard library but one can be easily written using the clock() function available in this header file.

```
// waits/sleeps for a specified number of milliseconds.
void sleep( clock_t waitTime )
{
clock_t target;
target = waitTime + clock();
while( target > clock() )
      ; /* null statement */
}
```

This sleep implementation makes program wait in terms of milliseconds. If waiting should be done in seconds, then new function need not be written and the existing itself suffices and the calling method differs:

```
sleep( (clock_t)1 * CLOCKS_PER_SEC );
```

This constant CLOCKS_PER_SEC refers to the number of clock ticks that tick in a second. So the call that is given here makes the program to wait for 1 second (In some systems this constant is CLK_TCK).

clock() function returns the processor time elapsed since the program invocation. So this function can be used to find the efficiency of the code and for testing purposes.

```
clock_t start,finish;
long duration,i;
start = clock();
```

```
for(i=0;i<100000000L;i--)      // the code to be tested
            ;
finish = clock();
duration = (double)(finish - start) / CLOCKS_PER_SEC;
printf( "%f seconds\n", duration );
```

11 OBJECT ORIENTATION AND C

Object-oriented programming rules the programming language world today. Object-oriented design is a good design methodology that helps in overall design of the software. This chapter explores the relationship of object-orientation and C and how to implement object-oriented design in C.

11.1.1 Relationship Between Object Oriented and Procedural Languages

Take an abstract data type (say stack). Object-oriented languages enforces the accessibility of the stack only to its member functions (methods). It prevents the illegal use of the data by the programmer purposefully sometimes and accidentally most of the times. By this way it allows only certain operations on it, the necessary details are known to the user (programmer). The same can be enforced in procedural language like C by strict standards and the careful coding by the programmer (but the same level of design robustness may not be achieved by this approach).

```
// contents of the file "stack.c"
typedef int boolean;
// the following are data elements.
// Note that the static qualifier that limits them to file scope.
static int top;
static elementType array[STK_SIZE];
```

```
// declaration of the functional elements
elementType pop();
void push(elementType );
boolean isEmpty();
boolean isFull();
```

In a high level of abstraction each file can be treated as a class. The global variables shall become ‘public’ variables and the static variables become ‘private’ variables. The functions declared acts on the data available are much like the methods in object-oriented languages that act on the class/object data. Most of the functionality of object-orientation can be viewed like this. But the ‘variables’ of type ‘stack.c’, that is the file, cannot be created in C (because C is not meant for that).

This striking similarity between the object-oriented languages is not accidental. Object-orientation is strongly based on procedural nature (even though this fact may not be as evident in languages like Effiel or Smalltalk than in languages such as C++).

11.2 How Object Orientation is better than Procedural

In many facets, object-orientation spares better than procedural one. As an illustration of this idea, let me show how object-orientation can improve readability. Consider the functions:

```
sscanf(char *,...)
sprintf(char *, ...)
// take first argument as char *.
```

Also consider:

```
fscanf(FILE *,...);
fprintf(FILE *,...);
```

that take FILE * as arguments. A look through the standard library shows that there are many clones for the same scanf and printf functions, that are general purpose (in user's point of view).

In an object-oriented language, if same implementation is made, namely FILE and String classes, the functions may be called like this:

```
str.scanf();
str.printf();
// and
fp.scanf();
fp.printf();
```

etc. where the printf and scanf names are used with the same name, but after a qualification. Invention of new names is not necessary and the readability also increases because of the usage of printf and scanf in different namespaces. Or else a single version of overloaded scanf or printf functions can be provided and depending on the arguments passed, the resolving of the functionality can take place.

Extending the same idea to for the FILE object: you can look fopen() as a construtor, fprintf, fscanf,fgetc etc. as member functions and fclose() as destructor. This leads to simplicity of organization of ideas and encapsulation and power, and is thus the success of object-oriented paradigm.

11.3 Object-Oriented Design and Procedural Languages

Object-oriented designs involve concepts like abstraction, encapsulation that are directly implemented in object-oriented languages.

Some avid object-oriented designers claim that object-oriented design cannot be done in programming languages such as C. Nothing is far from true. Even there is an existence theorem stating that *Objective-C is implemented in C* (there are even some attempts to write object-oriented code in assembly languages!). “Is object-oriented design using procedural languages efficient, easy and maintainable?” shall be a better question to discuss about.

Object oriented designs do not require an object–oriented language to implement the designs. But such designs are best implemented using object-oriented languages. Procedural languages can also be used to implement designs in an object-oriented fashion with some difficulty. Even if you don’t plan or need object-oriented design for your programming it will be useful to use the object-oriented ideas.

[Rumbaugh et al. 1991] puts this as, “Implementing an object-oriented design in a non-object-oriented language requires basically the same steps as implementing a design in an object-oriented language. The programmer using a non-object-oriented language must map object-oriented concepts into the target language, whereas the compiler for an object-oriented language performs such a mapping automatically”.

So procedural languages like C can be used to implement the same design. Object-oriented languages enforce the constraints externally but the base remains the same. To illustrate, the early implementations of C++, converted the C++ code to C code

(does it looks same as object-oriented programs written in C?). Eiffel is an object-oriented language. Eiffel compilers translate source programs into C. A C compiler then compiles the generated code to execute it later. Another such example is the DSM language.

Object-orientation is not the panacea for all problems in programming and one such example is the performance degradation[†]. Of course, the main objective in using object-orientation is not power or efficiency but making the programs more robust, maintainable and reusable etc., and to make the programmers life easier. So it is worth applying object-oriented design for use in procedural languages.

11.3.1 Implementing OO Design in Procedural Languages

Object-oriented languages have class hierarchies built through inheritance to have code reuse. Conventional languages don't have that mechanism, so for code reuse extra work has to be carried out.

In [Martin et al. 1991] the authors have mentioned three basic ways to do this with conventional languages,

1. Physically copy code from super-type modules (copy the code and have proper maintenance procedures for that),
2. Call the routine in super-type modules (call the copied modules from the extra code with proper maintenance code. This works as long as all the information regarding the subtypes are known and clear of what aspects to inherit),

[†] This is not a categorical statement. It's just comparison of performance of procedural Vs object-oriented languages, to give Fortran and C as examples of high-performance languages.

3. Build an inheritance support system.

Similarly issues concerning the handling of methods can be handled.

11.4 Implementing OO Design in C

C is one of the few procedural languages for which object-orientation can be easily implemented. This is because of its features like presence of pointers - particularly function pointers, its loosely typed nature, dynamic memory allocation. Lets now discuss how various object-oriented ideas can be implemented in C using these features theoretically and having an example after that for illustrating how these concepts materialize.

- Representing Classes

In C each class can be implemented as a structure with one-to-one correspondence between the class data members and the structure members. In other words, the classes are converted to equivalent data-structures to be represented procedurally. All the methods of the class with the structure should be put into a separate file unit.

- Encapsulation

C does allow encapsulation, but this requires discipline. To improve the encapsulation in C the following rules have to be followed [Rumbaugh et al. 1991] ,

1. Avoid use of global variables,
2. Package methods for each class into a separate file,

3. Treat objects of other classes as type “void *”.

- Representing Methods

The function calls can be resolved statically and so compile-time polymorphism can be implemented so as to implement class and object methods.

- Creating objects

Creating objects is just the same as creating structure variables. But the initialization (constructor) and destroyer (destructor) functions have to be called explicitly. The dynamic allocation facility (malloc and free) can be used for that.

- Inheritance

C’s one of low-level feature, the function pointer is useful in implementing (actually emulating) inheritance. In C runtime polymorphism cannot be implemented.

- Miscellaneous support

Object-oriented languages support the concept of pointer to the self (like ‘this’ pointer in C++ and ‘self’ in Ada). Passing an extra parameter (as first parameter) to all the object methods in the structure can do that.

11.4.1 OO Implementation of a Stack

Consider the following implementation of a stack that is based on the object-oriented principles.

11.4.1.1 Implementing the basic parts

```
typedef struct stack stack;


struct stack{
     // the data elements
     int top;
     elementType array[STK_SIZE];
     // the functional elements
     elementType (*pop)(stack *);
     void (*push)(stack *,elementType );
     boolean (*isEmpty)(stack *);
     boolean (*isFull)(stack *);
     };
```

This becomes the skeleton or in object-oriented terminology as class. It is mapped as a data-structure in form of a structure that has encapsulation and the methods are implemented using function pointers.

If you want the variable (object) of type stack, it's just as simple as,

```
stack aStack;
```

But this has function pointers that have to be initialized. The function pointers have to be initialized by writing the functions,

```
elementType popFun(stack *stk)
{
```

```
if(stk->top)
     return stk->array[stk->top--];
printf("Error: Nothing to pop from stack");
}
void pushFun(stack *stk,elementType element)
{
if(stk->top < STK_SIZE)
     stk->array[++stk->top] = element;
else
     printf("Error: Stack is full. Cannot push");
}
```

Similarly the functions isEmptyFun and isFullFun is to be written.

Now an initializer function (constructor) has to be called for each variable before it is used that takes the responsibility of initializing variable elements properly.

Here in our example it should be initialize the aStack as follows,

```
void init(stack *stk)
{
     stk->top = 0;
     stk->pop = popFun;
     stk->push = pushFun;
     stk->isEmpty = isEmptyFun;
     stk->isFull = isFullFun;
}
```

The code which uses this stack structure looks like this:

```
int main()
{
     stack aStack;
     init(&aStack);
     aStack.push(&aStack,20);
     printf("%d",aStack.pop(&aStack,20));
}
```

11.4.1.2 Improving the basic structure

This implementation just does what is required and can be improved. The space can be made allocated dynamically and freed when the scope exits. For such dynamic allocation to be done the init function has to be changed.

```
void init(stack *stk)
{
     stk = (stack *) malloc(sizeof(stack ));
     if(stk==0)
          {
          printf("error in allocating memory for
     object");
          exit(0);
          }
     stk->top = 0;
```

```
        stk->pop = popFun;
        stk->push = pushFun;
        stk->isEmpty = isEmptyFun;
        stk->isFull = isFullFun;
    }
```

Subsequently the way of creation of objects is also changed to support dynamic allocation:

```
    stack *someStack;
    init(someStack);
```

In the structure, you can see that for each function (method) supported in the structure the function pointers occupies space. Since the functions are going to remain same for all the objects associated with the class they can be put in a separate structure called as ‘class descriptor’. This will make the Stack structure to contain:

```
    struct stack{
        // the data elements
        int top;
        elementType array[STK_SIZE];
        // the functional elements are in class descriptor.
        struct classDescriptor *structDescriptor;
        };
```

The class decriptor will contain the following methods:

```
    struct classDescriptor{
```

```
elementType (*pop)(Stack *);
void (*push)(Stack *,elementType );
boolean (*isEmpty)(Stack *);
boolean (*isFull)(Stack *);
};
```

Now every object of the stack type is enough to contain the pointer to the classDescriptor. This will greatly minimize the size of the object required to support the member functions. The price is the extra level of indirection that have to be applied for every function call of that class.

11.4.1.3 Implementing Inheritance

The same idea of class descriptor is be used for implementing inheritance. Single inheritance is direct and easy. Add the new data and class descriptors to the new class descriptor. But implementing multiple inheritance is not possible by following this method.

This is an example of how the object-orientation can be implemented in C. Similarly the features like polymorphism, exception handling etc. can be handled. These features are more exploited in the object-oriented languages that generate C as the target code than the C programmers do. The older versions of C++ and Objective-C had similar kind of object-technology implementation.

As a whole C's loose type checking, pointers, dynamic memory allocation functions, its expressive flexible nature allows object-oriented concepts be implemented in it (after some work). Many of the languages and application systems indeed generate C code as output.

The users who are more interested both in the C and the object technology the solutions are the object extensions to the C language like C++ and Java (and possibly C# in near future). But it will be interesting to see how C itself can be used to emulate object-oriented concepts. To elaborate upon how C can be used to implement object-oriented design lets have an example

As I have said, it is possible, but it doesn't mean that we have to use object orientation in C. C is not meant for object-orientation and neither is designed having it in mind. C is better used as a system programming language and that's what it is meant for.

If you want to develop an object-oriented system better use an object-oriented language. But what about the millions of code written in C? If I want object-oriented design then should I start everything from scratch, forget and throw all the hard work previously done? In such cases the idea of using C for object-oriented design is attractive. Nevertheless designing such systems is non-intuitive, hard-to-manage and tough. But it will serve the purpose. The example we just saw explores the possibilities of implementing such object-orientation in C.

11.5 C++, Java, C# and Objective-C

Today C++ and Java are the most famous object-oriented languages.

C++ follows the merged approach with C to that of orthogonal approach by Objective-C. This means that the C programs can still be written in C++ and C is essentially a subset of C++.

C# is a new language from Microsoft. All the three are object-oriented and are based on C. They have the strong base built by C. They improve upon C by having object-orientation, removing the problematic and erroneous parts of C and add upon features that make them full-fledged languages of their own.

Java is commercially successful and is a pure object-oriented language.

Separate chapters are devoted for discussing the languages C++, Java and C# as a comparison between C.

Although Objective-C is a language that is based on C, it was not commercially much successful. So it is discussed here.

11.6 Objective-C

Objective-C is designed at the Stepstone Corporation by Brad Cox. It is an orthogonal addition of object-oriented ideas to C. Orthogonal addition means that there is an additional layer of object orientation and the code is in turn converted to bare C code. This kind of design has a particular advantage. The syntax need not be the same as that of C and can have its own constructs.

Objective-C is based on ordinary C with a library offering much of the object-oriented functionality. It introduces a new data-type, object identifier and a new operation, message passing and is a strict superset of C. Objective-C operates as a

preprocessor which accepts the Objective-C code to convert it into a ordinary C code which is processed then by a C compiler.

12 C and C++

Now the whole earth used the same language

and the same words

- Genesis 11:1

'C with classes' was the answer to the object oriented extension to C language by Bjarne Stroustrup. It was modified and named as C++. C++ is a separate language that is complex and huge compared to C and approach towards the problem solving itself differs from C. One of the main reasons behind the popularity of C++ is due to its backward compatibility with C.

"C was chosen as the base language for C++ because it

1) is versatile, terse and relatively low-level;
2) is adequate for most systems programming tasks;
3) runs everywhere and on everything; and
4) fits into the UNIX programming environment"

- creator of C++ Bjarne Stroustrup [Stroustrup 1986].

Almost every C program is a valid C++ program. C++ was first implemented as translation into C by use of a preprocessor. Now almost all the available C++ compilers convert C++ programs directly to object code. C++ improves upon C by changing the 'problematic' constructs in C. Since both the language support different paradigms it is

not worth writing programs in C++ as ‘pure C’. Unless you intend to do object oriented programming it is better to stick to C programming. Still if you want to do programming in C++, you should remember certain important points that are significantly different from C.

12.1 The Raw Power Of C

C is a systems programming language and so has raw power, and C++ enjoys the same due to its backward compatibility with C. For example the use of pointers in C++ is mainly due to its imminent use in C, that is a very powerful feature.

Another example is of preprocessor. C++ uses preprocessor because C uses it. Preprocessor is a naive tool for serious programming, whereas virtual functions of C++ are very powerful that provides runtime polymorphism. Lets look at an example where preprocessor macros are preferable to virtual functions.

[DJK-95] describes a situation where the overhead of virtual functions is significant. 'Message maps' are used for passing specified messages to derived class member functions. Had MFC used virtual functions for messages, it has to allocate 11,280 bytes for each control that the application needs. Each control has to inherit from a hierarchy of nearly 20 window classes derived from CWnd and CWnd. It also declares virtual functions for more than 140 messages. Assume that sizeof(int)==4 and so it uses a vtbl that needs 4 byte entry for each function. So it comes out that contol needs to get allocated 11,280 bytes (140 * 4 * 20) for supporting virtual functions. So it is better to go for macros, where no such memory overhead is there in such cases. This is an example

for a situation where the selection of a language feature to use based on the requirement at hand.

My point is that, due to its low-level nature, C has much power. Since C++ is superset of C, it makes use of this raw power when there is a need.

12.2 Subtle differences between C and C++

Even though C++ is a superset of C, there are subtle differences between the two. Understanding these difference is crucial in understanding the problems associated with maintaining the compatibility and migration from C to C++ (In particular from ANSI C to ANSI C++).

- Implicit int is removed. The implicit assumption of int is no longer assumed where it is required. Thus,

```
static i = 10;
const i = 10;
foo( ) { return 10;}
// implicit return type int assumed.
```

  are no longer legal. This implicit assumption of int was a subtle source of confusion (and errors) and it reduces the readability.

- Implicit char to int is removed. In C the automatic conversion from char to int is made in the case where char variables involve in expressions. But in C++ this constraint is removed.

This is very common in C. Even the standard library in C does this. The prototype for getchar is,

```
int getchar();
```

and it is common to have assignments like this,

```
char ch = getchar();
```

The implicit conversion from char to int is not valid in C because C++ is strongly typed to C.

- Implicit castings to void * is removed. A very good example is that of malloc(). malloc() returns void * and it is common in C to assign like this,

```
int *iptr = malloc(sizeof(int) * 100);
// implicit conversion from void * to int *.
// valid in C but not C++.
```

The reason is same. C++ is more strongly typed than C. This explicit casting makes programmers intention clearer and prevents unnecessary bugs due to implicit conversion.

- Translation limits are increased.

- Calling main() from within the program is allowed in C, but in C++ it is prohibited. Thus,

```
int main()
{
```

```
        main();
}
// leads to infinite loop in C.
// An error is issued if done in C++.
```

Calling main again makes no sense and C++ corrects this problem by disallowing main to be called form anywhere from the program.

- Taking address of a register variable is now possible in C++.

- wchar_t is a built-in data-type in C++ which was previously ‘typedef’ined in C. This primarily is to increase the support the two byte coding schemes like Unicode that are becoming increasingly popular.

- C Console and File I/O are different. But in C++ there is not much difference between the two.

- consts can be used for specifying size of arrays in C++ (but not in C)

```
const int size = 10;
int array[size]; // legal in C++
```

- In C the size of enumeration constant is sizeof(int) but not in C++. In C++ the size of the enumerated may be smaller or greater than sizeof(int). This is

because the implementations may select the more appropriate size to represent the enumerated constants.

- Using prototypes for functions is optional in C. But in C++ functions are not forward referenced and function prototypes are a must.

- Tentative definitions are not allowed in C++. The example that we saw for tentative definition becomes illegal in C++.

```
// this is in global area
int i;
// tentative definition. This becomes declaration after seeing
// the next definition
int i=0;
// definition. Legal in C not in C++
```

- The use of NULL pointers. In C NULL is normally defined as,

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

In C NULL is used universally for initializing all pointer types.

```
int *iptr = NULL;
// since implicit conversion from void * is removed this is
// invalid in C++.
```

In C++ NULL is usually defined as,

```
const int NULL = 0;       // or
#define NULL 0
```

(this is because of the same reason, in C++ is a strongly typed language)

C++ programmers prefer using plain 0 (or 0L for long) to using NULL.

```
char * cptr = 0;
// C++ programmers prefer this to NULL
long * lptr = 0L;
// this is a long type pointer.
```

- In C the global const variables by default have external linkage. All const variables that are not initialized will be automatically set to zero. But in C++ global const variables have static linkage and all the const variables must be explicitly initialized.

```
const int i;
// error in C++. i has to be explicitly initialized.
const float f = 10.0
// has static linkage
```

Does the following code (in C++) have static or global linkage?

```
const char * str = "something";
```

'str' has global linkage. 'str' is a pointer to a const character. It is not a constant pointer. Hence it has global linkage. To force static linkage, modify the declaration like this,

```
const char * const str = "something";
// now 'str' has static linkage
```

- In C++ there should be enough space for NULL termination character in string constants.

```
char vowels[5] = "aeiou";
//is invalid in C++ but valid in C.
```

By mistake the programmer may have forgotten to give space for the NULL termination character. puts(vowels) will print up-to NULL termination that will be encountered somewhere else. Since the access is beyond the limit of character array this leads to undefined behavior. To prevent such problems this is flagged as an error in C++.

- There exist some very subtle difference between C and C++. One such example is the result of applying a prefix ++ operator to a variable. In C it yields an rvalue [Kernighan and Ritchie 1988]. But in C++ it yields an lvalue.

```
int i = 0;
++i++;
// error in both C and C++
(++i)++;
// error in C but valid in C++
++i = 0;
// error in C but valid in C++
```

- The size of a character constant is the size of int in C. This is followed in C because int is the most efficient data-type that can be handled. But it wastes memory too. If the size of an integer is 4 bytes then the character constant also takes 4 bytes which is weird. In C++ size of a character constant is no more the size of int rather more naturally it is the size of character.

```
if(sizeof('a')==sizeof(char))
     printf("this is C++");
else if(sizeof('a')==sizeof(int))
     printf("this is C");
```

- In C++ you cannot bye-pass any declarations or initializations that are not given within a separate block by using jumps are there. Such jumps can occur in cases like switch-cases, return, continue and break statements.

```
switch(something)
{
case 'a' : int j;
        break;
case 'b' : j = 0;
// declaration of j may be missing. error
        break;
case 'c' : printf("%d",j);
// Both the declaration and initialization of j may be missing.
// error.
```

```
}
```

This is because, if the declarations and initializations are skipped and the destructors may be called without the call of corresponding constructors.

```
goto end;
     int j = 0;
end :
;
// this is an error in C++ and the following one too.
if(1 > 2)
     int j = 0;
```

- In addition to the predefined macro constants __LINE__, __FILE__, __TIME__, __DATE__, C++ compilers must define another preprocessor constant namely __cplusplus.

```
#ifdef __cplusplus
     #define NULL 0
#else
     #define NULL ((void *) 0)
#endif
```

Such code that should be available according to the compiler used for compiling code can be given this way using the constant __cplusplus for conditional compilation.

- Empty parameter list in C means any number of arguments can be passed to the function. But in C++ it means the function takes no arguments.

```
int fun();
// in C it means fun takes any number of arguments.
int  i = fun(10,20);
// this is legal in C. But in C++ empty argument list means the
// function takes no arguments. So this is an error in C++.
```

Thus in C++,

```
int fun();
// and
int fun(void)
```

are equivalent.

The reason why int foo() means that it may take any number of arguments is because of [Kernighan and Ritchie 1978] style function definition. It had the definitions like this,

```
int fun()
int a, int d;
{
// function code here
}
```

To make a prototype for this function it will look like this,

```
int fun();
```

so, it means fun() may take any number of arguments.

Exercise 12.1:

Comment on the following code with respect to C and C++:

```
struct someStruct{
};
printf("%d", sizeof(struct someStruct));
```

12.2.1 C and C++ Linkage

C linkage of functions is different from the linkage of C++ functions. This is because the C++ functions are capable to be overloaded and the information about the arguments should be passed to the linker.

Consider the C function,

```
int fun(int);
```

Since overloading of functions cannot be done in C it is enough for the compiler to tell the compiler that the identifier 'fun' is a function name. The linker just checks for the function name and resolves any function calls.

Consider the case of C++. Functions can be overloaded here.

```
int fun(int);
// and
int fun(float);
// and
int fun(int, int)
// are all different because they are overloaded.
```

This overloaded function ‘fun’ has to be resolved by the linker. So it is essential that not only the compiler pass the information about the function name, it also has to pass the information about the arguments. This is done by a technique called as ‘name mangling’.

‘Name mangling’ means that the information about the function name, argument types and other related information like if it is a const or not all are encoded to give a unique function identifier to the linker. The job of the linker becomes easy to resolve the calls for overloaded functions correctly because of this ‘name mangling’.

If ‘name mangling’ is not done the function has C linkage else it follows C++ linkage.

12.2.2 Forcing C Linkage

If C functions are called from C++ programs then it is likely to show linker errors saying that the function definition is not found. This is because the functions in C++ programs have C++ linkage and the functions compiled in C have C linkage as we have seen just now.

```
// in cProg.c
int cfun(){
// some code
}
//in cppProg.cpp
int cfun();
int main(){
```

```
cfun();
//error. Cfun follows c linkage.
}
```

To make this C function acceptable in C++ code the declaration for ‘cfun’ should be changed as follows,

```
//in cppProg.cpp
extern  "C’  int cfun();
int main(){
cfun();
//Now O.K. cfun can be called from C++ code now
}
```

Preceding the function declaration by extern "C" instructs the C++ compiler to follow C linkage for that function i.e. 'name mangling' is not done for that function. As we have seen the 'name mangling' is necessary for function overloading. So if a function is declared to have C linkage it cannot be overloaded.

```
//in cppProg.cpp
extern "C" int cfun();
extern "C" int cfun(int);
// error. Since C linkage is followed cfun cannot be overloaded.
```

More than one C functions are if necessary to be declared to have C linkage then those functions can be grouped together in a block preceded by extern "C".

```
extern "C" {
     int cfun1();
     void cfun2(struct s *);
```

```
    // other c function declarations go here.
    }
```

Otherwise they can be put in a header file and that inclusion can be declared to have C linkage,

```
extern "C" {
    #include "cfundecl.h"
    }
```

This forces all the functions declared in the header file "cfundecl.h" to be used in this C++ file to have C linkage. If you think preceding every C header file to be preceded with extern "C" is tedious, other tactic can also be followed. If the declarations may have to be used in both C and C++ compilers. C compilers doesn't recognize the keyword extern"C" so it has to be given using conditional compilation as follows,

```
// in "cfundecl.h"
#ifdef __cplusplus
    extern "C"  {
    // this code is invisible to a C compiler
#endif
    int cfun1(int);
    // and all other C function declarations go here
#ifdef __cplusplus
    }
#endif
```

Or this conditional can be still simpler. Just strip the two tokens, *extern* and *"C"* from all function declarations in the source code.

```
// in "cfundecl.h"
#ifndef __cplusplus
      #define extern "C"
      // make extern and "C" invisible to C compilers by replacing
      them
      // to white-space
#endif
      extern "C" int cfun1(int);
       // all the other function declarations go here
```

This conditional compilation structure is necessary to be present only in the starting and ending of the C header files.

Note: This kind of special inclusion of C header files is necessary for the non-standard and user-defined header files only. For standard header files, ordinary inclusion is enough.

```
#include<cstdio.h>
// this ordinary inclusion is enough. No 'extern "C"' job.
// In ANSI C++ c header files are prefixed with 'c'.
```

This kind of using C functions from C++ code has many advantages. One big advantage is that the legacy C code can directly be reused in C++ code.

12.2.3 Accessing C++ Objects from C code

The underlying representation for the C++ classes and plain C structures is almost the same.

```
class cppstring{
     private:
          int size;
          int capacity;
          char *buff;
     public:
          string();
          // other member functions for string class
          // and destructor
     };
```

Comparing with the C equivalent,

```
struct cstring{
          int size;
          int capacity;
          char *buff;
     };
```

The memory layout for the structure ‘cstring’ and ‘cppstring’ are almost the same. In other words the C++ compiler treats the class ‘cppstring’ just as ‘cstring’ structure. It means that the member functions are internally treated as global functions and the calls to the member functions are resolved accordingly. They do not occupy space in the memory layout for the object. This makes C++ object model very efficient. This is to show how close the C and C++ are in their internal representation.

The advantage is that the code like the following can be used,

```
void print(cstring *cs){
     printf("size = %d, capacity = %d", cs->size,
     cs->capacity);
     }
     // the old legacy code for cstring can be used for accessing
     // the C++ object
int main(){
     cppstring *cpps = new cppstring;
     print((cstring *) cpps);
     }
```

This equality between the struct and class is true unless the class has no virtual members, has no virtual base class in its hierarchy and no member objects have either virtual members or virtual base classes. In short the class should not be associated with any thing ‘virtual’ in nature. This is because the memory layout will then have virtual pointer table that makes the class and structure representation no more as equivalents.

Another point to note is that to have the equivalence between the class and struct, the data of the class should *not* be interfered by access specifiers (private/ public/ protected). This restriction is by ANSI because there is a possibility that the layout may differ in case of intervening access specifiers. But almost all compilers available now doesn’t make any difference due to this and so this point can be safely ignored. To put it together, you can safely access a C++ object's data from a C function if the C++ class has,

- no virtual functions (including inherited virtual functions),

- no fully-contained sub-objects with virtual functions,
- all its data in the same access-level section (access specifiers private /protected /public).

Nevertheless this property of the object model of C++ is used in the applications such as storing the data objects in DBMS, network data transfer of objects etc. This makes the legacy C code be used in the object-oriented code, backward compatibility with C and so many other advantages.

12.2.4 Other Differences

Other than the differences discussed between C and C++ there are other subtle differences that have to be understood when mixing C and C++ code.

The main() has to be compiled by a C++ compiler only. Because the code for static initialization for the C++ objects has to be inserted only by the C++ compiler.

When mixing C and C++ functions make sure that both the compilers are from same vendor. For example the compilers will follow similar function calling mechanisms so that the functions will be called correctly.

Most C code can be called from C++ without much problems. Similarly C++ code can also be called from C code under certain constraints. Transition from C to C++ will be smooth if the subtle differences between the two languages are understood well.

The downward compatibility with C is one of the main reasons behind the widespread success of C++. It is probably the topic that creates heated arguments among C++ programmers and each have their own views about this. C++ would have been

certainly different (and ‘better’) if downward compatibility were not the one of the main design goals of C++. But it is to be remembered that C++ is ubiquitous because of C.

13 C and Java

Java is a commendable addition to C based languages. Bill Joy defines Java as, “Java is just a small, simple, safe, object-oriented, interpreted or dynamically optimized, byte-coded, architecture-neutral, garbage-collected, multithreaded programming language with a strongly typed exception-handling mechanism for writing distributed, dynamically extensible programs” [Gosling and Joy 1995].

Java is based on C and borrows a lot from C and so is closely related to it. Of course, one major difference is that Java is an object-oriented language. Java also borrows lot of ideas from C++ but the relationship with C is closer because C is the base language.

13.1 How Java differs from C

Java cuts lots of features from C, modifies some features and also adds more features from C. This part discusses how Java learnt lessons from C by improves upon C and where it fails to gain.

C is a great success. There is no doubt about it. But some cost is involved in that success. C is a programming language for programmers and so it gives importance to 'writability'[4]. Integer is 'int' and 'string copy' is 'strcpy' in C. Java also uses the same keywords in C because, they are accepted and widely used by the programmers. But in

the case of C standard library, it is powerful but very small and C programmers had to reinvent lot of code. Java solves it by having a considerably big library.

Java removes lot of features from C which are either not suitable or problematic for various reasons like readability, portability etc. Pointers are the toughest area to master and is more error prone and night-mare for novice programmers. The designers of Java thought that the preprocessor concept is an antiquated one and so removed from Java. So features like conditional compilation is not there and cannot be done in the pure sense. Java is a pure object-oriented language and use of global variables violates the rules of abstraction, so there is no concept of global variables in Java.

In C the size of most of the data-types is implementation-dependent and so the programmer has to be very cautious in assuming the size of a data-type. As we have seen this may help suit the hardware and improve the efficiency. In Java the sizes of the data-types are well defined.

In C the byte-order may be big-endian or small-endian depending on the underlying platform. But in Java the byte-order is Big-endian. This resolves problems that arise due to the difference in byte order between machines particularly when the data is transferred from one machine to another in networks. This is help Java much because Java is a programming language for Internet.

In C when >> operator is applied to a variable, the filling of the rightmost vacated bits by 0 or 1 is implementation defined. This filling is called as arithmetic or logical fill. So the programmer should not assume anything about the filling followed. Java solves this problem by having separate operator for arithmetic and logical fills.

---

[4] Although there is no such jargon as ‘writability’, here I refer it to the ability to write the programs easily.

>> is for logical fill (vacated rightmost bits are filled by 0)

>>> is for arithmetic fill (vacated rightmost bits are filled by 1)

The problems with side effects are well known in C.

```
int i = 0;

i = i++ + ++i;

// the value of i cannot be predicted in case of C.
```

This is not the case of Java. Java says that the change is immediately available to the next usage.

```
int i = 0;

i = i++ + ++i;

// the value of i = 2 in case of Java
```

This eliminates the most of the bugs the C programmers make.

Due to efficiency and portability considerations C leaves most of the details implementation-dependent. It is a paradox that this is the main reason that the portability of C programs gets affected (even though C programs have reputation of being very portable). This seems to be a less-significant problem, but is really a big one because portability is one of the main reasons for Java's birth, one of the main design goals and that makes it the most portable programming language as of today. Java improves upon C by removing the constructs having various behavioral types in C by having mostly well defined behavior instead.

The C syntax is flexible and there are normally more than one-ways to specify the same thing. Pointers are C's stronghold, is also the problematic and tough feature to be

understood by the programmers. Pointer arithmetic is the place where even the experienced C programmers stumble. Java doesn't have explicit pointers, but have references that can be considered as cut-down version of pointers and arithmetic cannot be done on it. Dynamic memory management needs the programmer to carefully handle and memory explicitly and there is lot of scope to make fatal mistakes. Java has garbage collection that makes the programmer free for worrying about recollecting the allocated memory.

Java improves upon C syntactically and this makes tricky programming hard to write in Java. In any programming language, it is left to the programmer to not to resort to tricky programming and one can always write one such. To give one example in Java consider the following program,

```
class tricky{
      static{
      System.out.println("Hello world");
      }
      public static void main()
      {
            // note the missing arguments in main.
      }
}
```

This program when run prints the message "Hello world" and terminates by raising an exception stating that arguments to main() are missing. Because in Java the command line arguments are not optional.

In Java strings are special objects and so lot of functionality is available with strings. Since they are treated as special objects, one main drawback is also there. Java objects are not as efficient as other primitive data-types in Java itself.

```
public static void main(String argv[])
```

Only one argument is enough to be passed as the command line argument in Java because 's' is a String array so, argv.length == argc. Command line arguments are not optional and so cannot be omitted from declaration.

When giving path-names in include files, explicitly giving the path name can harm the portability of the program. For example,

```
#include "C:\mydir\sys.c"
```

the program using this line written to be used in Windows requires it to be changed to,

```
#include "/mydir/sys.c"
```

for UNIX based systems[†].

The path is for the original system where the file is located and will certainly vary with the path where the file will be available when it is ported and compiled in some other machine. Java solves this problem by having the concept of packages and with the use of the environmental variable CLASSPATH that is used to indicate the compiler where the files are to be searched for.

[†] assuming that the files are stored in both the systems with same directory and file names and relative path

13.2 Java Technology

The basic technology with which the Java works itself is different from the C based languages. C like languages has static linkage and work on the basis of conventional language techniques. But Java is different in the sense it has the platform independence for a greater extent and other advantages that its relative languages lack.

Main components of Java technology are,

- Java compiler,
- Java byte codes,
- Java file format and
- Java virtual machine.

The Java compiler is no different from compilers of other languages that it is firmly based on the conventional compiler techniques. Bytecodes are platform independent codes that form the basis of the platform independence by Java. They are targeted at the stack-oriented approach. All the evaluation is done in the stack by execution of the byte codes.

```
b = a + 10;
```

may be converted to Java bytecodes as follows,

```
iload_1    // stands for integer load the first local
           // variable into the operand stack
bipush 10  // bipush stands for push 'byte integer' 10 into
           // the operand stack.
iadd       // add the topmost two arguments in the stack
istore_2   // store the result in top of the stack to the
```

```
// second variable b
```

A Java compiler, irrespective of the machine it is compiled, generates the same code.

The next part is the Java intermediate file format. This is a standard format that is understood by the Java virtual machine (JVM or Java interpreter) that operates on and executes it. The byte-codes and all other related information for execution of the program are available in a organized way in the intermediate class file. This is similar to the .EXE code that is organized in a particular format that could be understood by the underlying operating system. The Java class file format is very compact and plays very important role in making Java platform independent.

C like languages has source code portability for some extent. Along with full source code portability, Java goes to next level of portability that may be termed as executable file portability. That means that the Java class files that are produced by compiling Java programs in on any compiler and platform will run on any machine provided that JVM is available to run that. This is achieved only through the class file produced by the Java compilers.

The last part and the most important one is the Java interpreter or otherwise called as Java virtual machine. This simulates a virtual machine that may be implemented in any machine. Thus the uniform behavior of the Java programs is assured even across platforms.

13.3 Java Native Interface

Java code has portability and the native codes, like the one written in C, can produce code that is efficient. To get the best of both worlds, the portable Java code can be used and the very frequently used functions like library functions can be written in C and Java Native Interface (JNI) achieves just that. This part of the chapter explores what JNI can provide in the context of C. Using JNI you can:

- Call C code from Java code,
- Call Java code from C code,
- Embed JVM in C programs.

JNI acts as an interface between the C and Java code. With this functionality of C can be achieved from Java programs and vice-versa.

13.3.1 Calling Java code from C code

Calling Java code from C code can be done for and have following advantages,

- To achieve the efficiency of the code and for time-critical applications that is possible through the C/C++ or even assembly code.
- To have platform dependent code written in C to be used by the Java program to achieve the platform dependent functionality indirectly.
- Lot of libraries and tested code that are available in the other more mature languages such as C and C++ becomes available to Java code.

13.3.2 Calling Java code from C code

Calling Java code from C code can be done for and have following advantages,

- If some functionality is already available in Java code the C programs need not be written again. The C code can just call the Java code but through JNI
- Functionality of the Java programming language can be exploited by this. For example C doesn't have sophisticated exception-handling mechanism and through native methods this can be achieved. Runtime type checking is the feature that is not available in C and for that JNI can be used to do the same.

To explain how the Java code can be written to call the C code lets have an example. The process is a little tedious one. The example includes a function written in C to add and display the two float variables that are passed to the native function. There is another method to have a wrapper function to call the C standard library functions. The process explained is the generalized one and the exact detail of how the interface between the C and Java code depends on the implementation.

```
class callCFromJava{
     public native void addTwoFloats(float i,float j);
     public native double sin(double value);
     // these are declarations for the C native functions that will
     // be available at run-time
     static {
        System.loadLibrary("twofloats");
          // the name of the DLL where the code for the native
          // methods is located.
    }
```

```
    public static void main(String[] args) {
          float i=10.0f,j=20.0f;
     new callCFromJava().addTwoFloats(i,j);
     System.out.println(new callCFromJava().sin(1.1));
          // note how the native functions are called
    }
}
```

In the Java code and the notable points to enable calling native code are,

- The C functions to be called, addTwoFloats() and sin() are declared with keyword ‘native’ to indicate the compiler that they are native methods.
- The native methods are treated and called the same way as Java methods.
- Inside the static block (this block is for initialization of static variables and is called before main()) the library where the code for C programs is available is loaded.

This code is compiled as usual with the Java compiler,

```
javac callCFromJava.java
```

and the .class file is generated.

The next step is to generate a header file that contains the information about the methods that should be available for the C/C++ compilers for generating DLLs such that it will be accessible to the JVM. A utility called as javah is available for this purpose and call is made like this:

```
javah callCFromJava
```

This generates the header file “callCFromJava.h” for the native method. This header file has to be included in the C code where the code for the C functions is available.

The next one is the important step of writing the C native methods. The code looks like this:

```
#include <jni.h>
// programs that declare native methods should include this header
file.
#include <stdio.h>
#include <math.h>
// note that the header file for native methods is included here
#include "callCFromJava.h"


// the declaration for the function is made with two
// extra parameters.
JNIEXPORT void JNICALL
Java_callCFromJava_addTwoFloats(JNIEnv *env, jobject
obj, jfloat i, jfloat j)
{
    float k = i+j;
     printf("%f\n",k);
    return;
}
JNIEXPORT jdouble JNICALL
```

```
Java_callCFromJava_sin(JNIEnv *env, jobject obj,
jdouble value)
{
    return sin(value);
}
```

Each function definition follows the format:

```
JNIEXPORT ReturnType JNICALL Java_ClassName_sin(JNIEnv
*env, jobject obj, FormalArguments )
```

The function names also have special way of naming. All the functions start with Java_ followed by the class name to which the native method belongs and that is followed by the actual function name. Also note that all the native functions have first two arguments as mandatory, JNIEnv *env and jobject obj.

The new naming convention, the inclusion of the header files enables the C/C++ compilers to compile the code to a DLL that will be accessed by the Java Interpreter. With this the process of calling Java code from C code ends.

This DLL is used by the Java interpreter at runtime to find and execute the corresponding native method. To invoke the JVM,

```
java callCFromJava
// prints
// 30.000000
// 0.8912073600614354
```

the output shows the execution of the native methods.

13.3.3 Embedding JVM in C programs

This is to achieve the functionality of the Java code through the JVM itself from the C code. For example you may write a browser program and to support applet functionality the JVM has to be embedded into the program. In this case JNI can be used to embed the JVM and whenever an applet have to be displayed, the Java Interpreter can be invoked from the code to do the same. This is through the ‘invocation APIs’ that are available with JNI.

14 C and C#

"The question is," said Humpty Dumpty,

"which is to be master-that's all."

- Lewis Carroll

C#[†] (pronounced "C sharp"), defined by Microsoft as "C# is a simple, modern, object oriented, and type-safe programming language derived from C and C++" [Hejlsberg and Wilamuth 2001], is a new addition to the C based programming languages. For the past two decades, C and C++ have been the most widely used and successful languages for developing software of varied kind. While both languages provide the programmer with a tremendous amount of fine-grained control, this flexibility comes at a cost to productivity and Microsoft claims it has come out with a language offering better balance between power and productivity.

It remains to be seen how successful C# is going to be. It is the idea to get the features of rapid application development of Visual Basic and the power of Visual C++ and the simplicity similar to its competitor Java. This chapter devotes the see the features and ideas of how C# is based on C and improves upon C.

[†] Since the language is being developed when this book is written and the information about the language is not still fully available, the information provided in this chapter may not fully comply to the language that is actually released. Most of the information available is based on the preliminary information available in the Internet and [Hejlsberg and Wilamuth 2001].

14.1 What C# promises?

C# is part of Microsoft Visual Studio 7.0. It has common execution engine that language can be used to run the programs from other languages such as Visual basic and supports scripting languages such as Python, VBScript, Jscript and other languages. It is called as Common Language Subset (CLS). It doesn't have its own library. The already available VC++ and VB libraries are used. C# is not as platform independent as Java and targets at Next Generation Windows Services (NWGS) platform [Hejlsberg and Wilamuth 2001].

14.2 'Hello World' in C#

Lets see how the simple "Hello, world" program looks like in C#.

```
using System;
class Hello
{
static void Main() {
      Console.WriteLine("Hello, world");
      }
}
```

The using directive is from Pascal language and this becomes shortcut for:

```
System.Console.WriteLine("Hello, world");
```

As you can see everything is within a class and so C# is a pure object-oriented programming language. The Main (declared as a static method) is the starting point for the program execution. The WriteLine function greets you “Hello World”.

14.3 Datatypes in C#

C# has two major kinds of types,

- Value types,
- Reference types.

C# doesn’t have explicit pointer types and instead have reference types. Internally references are nothing but pointers but with lot of restrictions imposed on its usage. References have their own merits and demerits. Unlike pointers, pointer arithmetic cannot be done on reference types and the power of the reference type is less than that of pointers. But it is safe and easy to be handled by a beginner in that language.

C# has a high-level of view that all types are objects, it is referred to as some sort of “unified type system”.

14.3.1 Value types

The value types are just like simple C data-types that do not have any object-oriented touch with them. It includes signed types sbyte (8-bits), short (16-bits), int (32-bits), long (64-bits) and their unsigned equivalents byte, ushort, uint, ulong.

There is one character type in C# (like Java) that can represent a Unicode character.

The floating-point types are float, double. bool type (which can take true/false), object (base type for all types), String (made up of sequence of Unicode characters) are available. It also includes types like struct and enum. C# doesn't support unions.

C# implements built-in support for data types like decimal and string (borrowed from SQL), and lets you implement new primitive types that are as efficient as the existing ones. In C for most of the requirements the type 'int' suffices, and when use for such decimal arises it is customary to typedef and use the existing data-type as new type. Strings as you know, there is not much support in C language, it is not a data-type and closely related to pointers.

14.4 Arrays

Arrays in C# are of reference type and mostly follow Java style. C# has the best of both worlds by having the regular C array which is referred in C# as rectangular array and also have Java style 'jagged' arrays.

```
int regular2DArray [ , ];
// this is a rectangular (C like) array.
int jagged2DArray [] [];
// this is a 'jagged' array
```

In C as we have seen, 'ragged' or 'jagged' arrays can be implemented by having a pointer array and allocating memory dynamically for each array. The same idea is followed here except that instead of pointer type, reference type is used. This makes optimal use of space sometimes since the sub-arrays may be of varying length. The

compromise is that additional indirections are needed to refer to access sub-arrays. This access overhead is not there in rectangular array since all the sub-arrays are of same size.

When more than on way of representation is supported then at some point of time the user will require switching from one representation to another. Here to convert from one array type to another, techniques called as boxing and un-boxing are used.

14.5 Structs and Classes

Structs are of value type compared to classes that are of reference type. This means structs are plain structs as in C and the classes are used for object-orientation as in C++ or Java. The advantage here is that if an array of struct type needs to be declared they can fully be contained in the array itself. Whereas an array of class type will allocate the space only for the references and the allocation of space for objects should take palce.

```
structType []  structArray = new structType[10];
```

whereas for the class type,

```
classType [] objectArray  = new classType[10];
for  (int i = 0; i < 10; i++)
     objectArray[i] = new Point( );
```

14.6 Delegates

Delegates are the answer for the function pointers in C. As we have seen function pointer is a very powerful feature in C but can be easily misused and is sometimes

unsafe. Delegates closely resemble function pointers and C# promises that delegates are type-safe, secure and object-oriented.

```
delegate void delegateType();
// delegateType is the type that can be used to instantiate delegates
// that take no arguments and return nothing
void aFun()
{
     Console.WriteLine("Called using aDelegate");
}
delegateType aDelegate = new delegateType(aFun);
aDelegate();    // call aFun
```

14.7 Enums

C# enumerations differ from C enums such that the enumerated constants need to be qualified by the name of the enumeration.

```
enum workingDay {
monday,tuesday,wednesday,thursday,friday };
workingDay today;
today = workingDay.monday;
//note that monday is qualified by workingDay
```

This helps the enumeration constants to remain in a separate namespace.

14.8 Casting

One of the major problems in C is that virtually any type can be casted to other type. This gives power to the programmer when doing low-level programming. C# is strongly typed and arithmetic operations and conversions are not allowed if they overflow the target variable or object. If you are a power programmer C# also has facility to disable such checking explicitly.

14.9 The preprocessor

C# supports preprocessing. The preprocessor elements:

```
#define
#undef
#if
#elif
#else
#endif
```

do the same job as in C. For example:

```
#define PRE1
#undef  PRE2
class SomeClass
{
static void some(){
     #if PRE1
```

```
            DoThisFunction();
        #else
            DoThatFunction();
            #if PRE2
                DoSomeFunction(this.ToString());
            #endif
        #endif
        }
    }
```

C# tries to improve upon the C-preprocessor and one such is ‘conditional methods’ that is discussed now.

14.9.1 Conditional methods

An interesting addition is the conditional methods. Scattering the source program with #ifs and #endifs looks ugly and it becomes hard to test code with possible conditional inclusions. For that C# provides conditional methods that will be included if that preprocessor variable is defined. First the method has to be declared as conditional like this:

```
// in the file "cond.cs"
[Conditional ("PRE1")]
void static DoThisFunction()
{
```

```
System.Console.WriteLine("This method will be executed
only if PRE1 is defined in the place of invocation");
}

// inside "someclass.cs" inside the class definition
#define PRE1
public void static someFunction {
     cond.DoThisFunction();
     // function is called because PRE1 is defined
}
#undef PRE1
public void static someFunction {
     cond.DoThisFunction();
     // this statement is ignored because PRE1 is not defined here
}
```

This is not a very significant addition and this may affect the class hierarchies. For example if the function in base class is a conditional method then depending on the preprocessor variable is defined or not the derived classes may override it or create a new function. In my view conditional methods will help confusing the programmer more than help programmer.

C# also supports #error and #line in addition to a new directive #warning that work in a similar way.

14.10 Using 'native' code

The real-world applications require to work with old code that is available and that is possible in C# through:

- Including native support for the Component Object Model (COM) and Windows-based APIs.
- Low-level programming is possible and basic blocks like C like structures are supported.
- Allowing restricted use of native pointers.

An interesting point to note is that the every object in C# becomes COM component automatically. So the interfaces like IUnknown are automatically implemented and not have to be done by the programmer explicitly. Due to its close relationship with COM objects, the C# programs can natively use the COM components. It should be noted that the COM can be written in any language and that difference doesn't prohibit their use with other components.

At one-level where the full-control over the resources is required, the C/C++ like programming of using pointers and explicit memory management inside marked blocks can be done.

All this means that the tested, legacy C/C++ code need not be discarded and can still be used and build upon.

15 Compiler Design and C

To have a good understanding on the nuances of C language, it is necessary to have some idea about the compilation process.

15.1 An Overview of Compilation Process

Compilation is a process by which the high-level language code is translated into machine understandable code. The software that performs this job is known as a compiler. This machine level code is also known as object code since creating this code is the objective of the compiler. In this part of the chapter we will have an outlook on the general compilation process.

Compilation is not usually done at a single stretch. It can go through several passes, depending upon the complexity. Sometimes these passes may be organized to do it logically rather than actually going through one pass after another. The control flow in a typical compiler is given below.

preprocessing => lexical analysis => syntax analysis => semantic analysis => assembly code generation => optimization =>object code.

At this stage we get a relocatable code, which is given to the linker, in turn produces an executable code. To execute it, we need to load the code in memory. A loader is used to accomplish this job.

15.1.1 Preprocessing

Preprocessor is generally used for substitution of strings and macros. It provides a way to have symbolic constants and inline functions. One of the main features of the preprocessor is conditional compilation.

Preprocessor, is NOT a part of compiler, rather it is a tool that runs before the compiler operates on the code. This feature is mainly used in assemblers. Most of the languages don't have a preprocessing facility, although one can be added to it. The preprocessor in C is designed such that it can be done in a single pass.

15.1.2 Lexical Analysis

'Divide and conquer' is a good rule. If we cannot interpret the meaning of a lengthy sentence as a whole, we divide it into words and try to analyze it. Similarly a compiler divides the entire source program into lexical units, better known as tokens and then process it. A token is a well-defined word of the programming language. It may be a constant, a keyword, an identifier etc…

The lexical analyzer (also called as scanner) accomplishes this breaking up job. In some implementations the lexical analyzers even enter the identifier names into the symbol table for later use by other parts of the compiler.

Consider a typical example, where a spell checker of a word processor tries to read the word "manner". It follows the greedy algorithm, i.e. look for the longest word. When it has reads the characters 'm', 'a' & 'n ', it never immediately interprets it as a word(man). It always assumes that the following characters may be a part of this word, and in this example it is. So 'ner' is also read and 'manner' is considered as a word.

When the assumption fails the word already formed is returned, and the received character is considered as a beginning of the next word.

A C lexical analyzer will always follow this ‘maximal munch’ rule (sometimes referred to as ‘greedy technique’). Understanding this rule is very useful in case of operators like ++. A few examples would explain this:

- `x++y`
- `x+++y`
- `x++++y`
- `x+++++y`

The scanner sees the input x. It recognizes x as an identifier and returns ‘identifier’ to the parser. And then it looks for the next token. It moves to scan +. It doesn’t immediately return ‘plus’. The next character is also ‘+’. As ++ is valid, it returns ‘plus_plus’ and starts scanning for the next token. ‘y’ is read and returned as an identifier.

So the sequence is recognized as `x ++ y` (and not as `x + +y`). This expression is invalid and the compiler issues an error.

Using the same idea, x+++y will be recognized as `x ++ + y` (so is a valid expression).

Considering, `x++++y` it will be read as `x ++ ++ y` (which is illegal).

And finally `x+++++y` will be read as `x ++ ++ + y` (which is also illegal).

So if you want the lexical analyzer to interpret like `x++ + ++y` you should give the same by inserting blanks explicitly.

That’s why `x-->y` is interpreted as `x-- > y` and not as `x- ->y`.

Exercise 15.1:

i+++j is scanned as i ++ + j. Here + is a binary operator. Consider this: i&&&j that is scanned as i && & j. Here is & a unary or binary operator? Similarly consider this: i ** j which is scanned as i * * j. Which * becomes unary and which becomes binary? In general how do you think the resolution of unary and binary operators are made in such cases?

Exercise 15.2:

Comment on the following code:

```
int i = 10;
int * ip = &i;
int **ipp = &&i;
```

Let us consider an interesting problem associated with application of this 'maximal munch' rule,

```
int i=10, j=2;
int *ip= &i, *jp = &j;
int k = *ip/*jp;
printf("%d",k);
```

The compiler issued an error stating that "Unexpected end of file in comment started in line 3". The programmer intended to divide two integers, but by the 'maximum

munch' rule, the compiler treats the operator sequence / and * as /* which happens to be the starting of comment. To force what is intended by the programmer,

```
int k = *ip/ *jp;
// give some space explicitly to separate / and *
//or
int k = *ip/(*jp);
// put braces to force the intention
```

will solve the problem.

15.1.3 Syntax Analyzer

Just like any natural language we use, all the programming languages have their own syntax. The syntax is explained by the unique grammar of the language. Grammar acts as a tool to recognize the program given as input. Syntax analyzer, as the name suggests, verifies the structure of the program against the language specifications with the help of grammar. Every valid program should conform to the corresponding grammar. C also has a well-defined grammar. Grammar is a very powerful tool. For example, the precedence and associativity of the operators in the expressions can be directly specified by the grammar.

The rules of *precedence* are encoded into the production rules for each operator. Production rules in-turn call others so, the precedence is formed.

For example, the syntax for *additive-expression* includes the rule:

```
additive-expression:
```

```
        additive-expression     +(or)-     multiplicative-
    expression
```

First the production rules that involve the lowest levels is called. They in-turn check for the production rules that involve the operators of higher level. Thus the precedence is given through grammar.

Similarly the associativity is also expressed in the production rules. Consider the same additive-expression production. All the +(or)- operators are recognized from left-to right because the left-non-terminal comes in the left-side of the production rule. Compare it to other productions with right-to-left associativity.

```
    ConditionalExpr:
        ConditionalOrExpr
        ConditionalOrExpr ? Expr : ConditionalExpr
```

The left-non-terminal comes in the RHS of the production rule. In this way, the associativity can be expressed in the grammar productions.

Answers to questions related to many questions such as why the compilers require type-name to be put into the parenthesis in sizeof operator as in the following,

```
    sizeof (int);
    // doesn't issue error
    sizeof  int;
    // issues error
```

are related with syntactic issues. A look at the grammar of C will help understand this,

```
    unary-expression:
```

```
        postfix-expression
        ++ unary-expression
        -- unary-expression
        unary-operator cast-expression
        sizeof unary-expression
        sizeof (type-name)
```

The grammar says that unary-expression may be made up of any one of the six options. As you can see that the syntax of the sizeof says that the type-name has to be given only within the parenthesis providing the answer.

Lets see another example,

```
void fun(int i, int j)
{
if(i>j)
goto end;
else
     return;
end :
}
```

The compiler issues a syntax error stating that “; is missing before }”.

Have a look at the grammar for labels,

```
labeled-statement :
     identifier : statement
     case constant-expression : statement
```

```
default : statement
```

So the grammar mandates the label should be followed by a colon and a statement. The syntax analyzer issued an error because the program didn't follow the grammar correctly. Knowing this you can insert any statement or a null statement to make it conformant to the grammar,

```
void fun(int i, int j)
{
if(i>j)
goto end;
else
     return;
end :
     ;
// insert at least a null statement to make it conformant to the
syntax
}
```

Thus, the duty of the syntax analyzer is to make sure that the program concerned conforms fully to the grammar and promptly issue messages to the user if violated.

15.1.4 Semantic Analysis & Code Generation

After verifying the correctness for the syntax, the compiler goes on to the next phase called semantic analysis. In this phase the compiler understands the language constructs and the appropriate object code is produced, either explicitly or implicitly. The

grammar has the limitation that it can only check the syntactic validity of the program. Inferring the meaning of a statement involves careful analysis of the source program. Only if the statements make any sense the code is generated, else an error is issued.

Parser is the part of the compiler that takes care of syntax and semantic analysis. Syntax and semantic analysis normally go hand-in-hand.

```
const int i = 1;
if( i > 10)
      i++;
```

If this code is executed the compiler will issue a warning message like "unreachable code i++". It predicts that the if condition will never be executed and this information is found in semantic analysis phase.

Code generation involves generating the code that is for the target machine where the program runs.

15.1.5 Optimization

This is an optional, but desirable part of the compiler. Actually the generated code may not be the optimal code to do the job. So the compiler will use well-known algorithms to analyze and produce a smaller and efficient code that does the same job.

Lets see few examples of how the compilers optimize the code that we write.

15.1.5.1 Dead-code elimination

Consider the code,

```
i * j;
```

It has to be remembered that the statements are executed for their side effects. In this case, i and j are multiplied with no side-effects and so has no effect in the program execution. So the optimizer can safely omit the code generated for the statement.

15.1.5.2 Strength reduction

Strength reduction involves replacing of constructs that take more resources with the one that takes lesser.

```
int i = 3,j;
for(j = 0; j < i; j++)
     a[j] = j;
```

The for loop can be replaced with more efficient and simpler statements as follows,

```
a[0] = 0;
a[1] = 1;
a[2] = 2;
```

This makes the code more efficient.

A similar technique is to replace a small switch statement with if-then-else statements.

15.1.5.3 Common Sub-expressions

When in a statement, expressions if repeated can be replaced if the variables involved in both the expressions provided that no variable that is part of the expression occurred in the LHS of any assignment statement (i.e. didn’t undergo any change). Consider the following example:

```
int j = k * 20;
l = m * k * 20 + k;
```

Here the expression k * 20 is enough to be evaluated once and the result can be used in the next expression assuming that k's value is not modified before that replacement occurs.

```
int j = k * 20;
l = m * j + k;
// note optimization here
```

15.1.5.4 Replacing run-time evaluations with compile-time ones

This is the idea we have already discussed in the form of constant-expression evaluation. Substitution of constant variables with its equivalent constants is also one such optimization.

15.1.5.5 Unreachable code

Semantic analysis can reveal many important details that may be useful in optimizations and also in giving intelligent messages to the programmer.

```
const int TRUE = 1;
if( TRUE == -1)
      // this line will never be executed
```

Here at compile time compiler can identify that the code given contains statements that will never be executed. An intelligent optimizer will not produce any code for those

unreachable statements. That's why you sometimes get warning messages like "unreachable code in function _xyz", and "the code has no effect in function _pqr".

15.1.5.6 Loop optimizations

Since most of the execution time is spent in executing the statements inside the loops in almost any program, optimizations on loops have significant effect in the execution time efficiency.

```
while(i <= j)
{
if(i<100){
      i +=10;
      // do something
      }
else {
      // do something else
       i +=10;
      }
}
```

This loop can be optimized to,

```
while(i <= j)
{
i +=10;
if(i<100){
```

```
        // do something
        }
    else {
        // do something else
        }
    }
```

because the statement i += 10; is part of both the if and else statements and the code becomes compact.

There are several loop optimization techniques like this are available, [Aho and Ullman 1977] is one good reference that will be useful in applying optimizations even the programs we write.

Compilers have various options to force the optimizations or to aviod optimizations. For debugging purposes the optimizations can be switched off. For release verions of the software and for testing it is better to have optimizations be done on the code.

Exercise 15.3:

Consider the two codes,

```
a. for(i=0;i<num;i++)
b. for(i=num;i>0;i--)
```

Which one de you think executes faster? (if necessary assume that no code optimization is done, the microprocessor has flags etc.)

15.1.6 Other Important Parts

The other important parts of the compiler are,

15.1.6.1 Table-management routines

Symbol table is the structure that will be accessed by almost all stages of the compiler and lot of other tables has to be managed. The table-management modules take care of the management of the various tables involved in the process of compiling.

15.1.6.2 Error Handler

While compiling errors can occur at anytime and the error-handling module takes care of the process of issuing the error messages to the user. It also has the important job of recovering form the error and continues the compilation process if possible.

15.2 An example

Lets look at an example of taking a code and see it through various stages of how it is compiled. The source code is:

```
int i = four * 8 + fun();
```

The lexical analyzer is the first part of the compiler to attack it and it returns the tokens to the parser. The tokens returned are:

```
KEYWORD (int)
IDENTIFIER (i)
EQUAL_TO_OP
```

```
IDENTIFIER (four)
MULTI_OP
INT_CONSTANT (8)
ADD_OP
IDENTIFIER (fun)
OPEN_PARENS
CLOSE_PARENS
SEMI_COLON
```

The syntax analyzer checks for the correctness of the code program code and finds it to be acceptable. It sees that the operators are having valid operands and the statement ends with semicolon.

Next the semantic analyzer operates on it to see what the code means. Here it enters the variable i into the symbol table. It checks if the identifiers four and fun are already declared and looks for its details in the symbol table.

Now the intermediate code generator creates code for this code segment that may look like:

```
MOV _temp1, 8
// mov 8 to temp1
MUL _temp1, _four
// multiply four and temp1 and store result in temp1
CALL _temp2, _fun
// call the fun() and store the result in
ADD _temp1, temp2
// add temp1 and temp2, store the result in temp1
```

```
MOV _i, _temp1
// move the final result to i
```

The optimizer may optionally operate on this intermediate code to generate a more efficient version of the code.

The final part is the code generator that converts the platform independent intermediate code to platform dependent object code. With this the compilation process ends and the linker takes care to link the object files to generate the final executable code.

15.3 Compilers for C

Thousands of compilers have been written for C language. A wide range of compilers is available commercially to be chosen from. C is a relatively small language so it requires a comparatively small compiler. In fact, Dennis M. Ritchie wrote the first C compiler for PDP-11 that can be stored in just 4k of memory. It was basically a two-pass compiler and optional third pass for optimization was also available. It used operator precedence for expressions and recursive descent parsing for all other constructs.

As I said earlier, most of the compilers in the market have two pass structure. The first pass is used for lexical and syntax analysis and produces intermediate code. The second pass employs the optimization and code generation.

15.4 Parsing

Parsing is the general term for the part of the compiler that involve the parts of syntax-analyzer, semantic analyzer and sometimes the code-generator. This serves as the next component to lexical analyzer. Parsing techniques

involve in analyzing the syntactic and semantic parts of the program from the tokens returned from the lexical analyzer and serves to generate code by the intermediate code generator. There are lots of parsing techniques available from simple to complex and suitable for hand-written or automated.

15.4.1 Recursive descent parsing

This type of parser is one of the easiest parser to implement. As the name indicates it employs the recursion as the basis for parsing the given source. The production rules of the given grammar can directly be converted to functions in the parser. In other words it can have one-to-one relationship between the grammar productions and the functions that have to be written in the recursive-descent parser. So it becomes easy to hand-code the whole parser and has more readability and is clearer for the compiler writer.

The production rules involve left-recursion, which means that the same non-terminals that are in the LHS appear in the RHS of the production also. Since the parser employs recursion, this may lead to infinite loop.

consider the production:

```
additive-expression:
     additive-expression +(or)- multiplicative-
expression
```

if it is converted directly to the recursive function as,

```
additiveExpr()
```

```
{
        additiveExpr();
        while( additiveOperators(token) )
        {
                advance();
                multiplicativeExpr()
        }
}
```

as you can see in this code, it leads to infinite-loop.

The grammar production means something different. It means that all the multiplication expressions have to be compiled before handling the additive-symbols. The multiplicative expression production in-turn will in-turn check for the operators with still higher precedence than them to get them recognized first through similar grammar productions.

To remove the left-recursion, the grammar production can be written as,

```
additive-expression:
        multiplicative-expression     additive-expression-prime

additive-expression-prime:
        +(or)-     multiplicative-expression     additive-expression-prime
```

Now it can be implemented using equivalent functions as,

```
additiveExpr()
{
     multiplicativeExpr();
     additiveExprPrime();
}
additiveExprPrime()
{
if( additiveOperators(token) )
     {
          advance();
          multiplicativeExpr();
          additiveExprPrime();
     }
}
```

As you can see, the first function, `additiveExpr()` is called only once and it inturn calls `additiveExprPrime()`. This second function has recursion and calls over again and again till all the additive operators are exhausted.

The both can be combined and the equivalent function can be written more non-formally as,

```
additiveExpr()
{
     multiplicativeExpr();
```

```
        while( additiveOperators(token) )
        {
              advance();
              multiplicativeExpr();
              additiveExprPrime();
        }
  }
```

Look at the original grammar production and the available function implementation. This is how the recursive-descent parsers can be implemented directly from grammar productions.

15.5 The 'tinyExpr' Expression Compiler

Java inherits most of the operators from C and so the C and Java expressions work very similar. 'tinyExpr' is a small expression compiler that compiles the expressions to the intermediate code by the standard compilation process. This implementation of the compiler serves multiple purposes,

- to explain the overall working of the standard compilation process, by implementing various parts of the compiler: the lexical analyzer, parser etc and how they interact and work together to convert the source code to the target code (here bytecode),
- as one application to show how powerful recursion is and explain the working of a recursive descent parser,

- to understand how compilers work to understand the expressions we write in our programs and understanding the working of this compiler may serve to unravel some 'secrets' of how expressions are evaluated and why sometimes they give some 'weird' results.to explain how Java bytecodes are produced and how they serve to make Java platform independent,
- the basics of how bytecode interpreters/ virtual machines (Java) works and how the platform independent behavior is achieved.
- to serve as a primitive starting point for write a full-fledged C or Java compiler

Yes. The implementation is so small and serves its purpose of compiling, executing expressions and at the same time fulfills all the purposes in one. The full implementation is given as appendix and the concepts are explained in the remaining of the chapter.

'tinyExpr' compiler takes expressions as input and convert them to equivalent bytecodes. These byte-codes are subset of actual Java byte-codes. The compiler generates code for a virtual stack oriented machine.

The features of the compiler are,

- support most of the operators (arithmetic, bit-wise and assignment) operators are allowed that are available in C and Java,
- error messages are issued whenever possible,
- automatic variable declaration and the variables are assumed to be of type int.
- more than one expression can be separated by commas.
- produces bytecodes that are subset of Java bytecodes.

The features missing from this compiler implementation are,

- optimization,
- full-fledged support for error handling,
- support for any control structures,
- support for data-types other than integer,
- support for comments,
- constant-expression evaluation

The module distribution is as follows,

tinyexpr.h - this header file in-turn includes standard header files and contains function prototypes

mnemonic.h - the mnemonic codes for bytecodes are available in this header file

tinyexpr.c - this is the main file that contains the recursive-descent parser and its code- generator.

lex.c - a tiny lexical analyzer is available in this file

symtable.c - this file contains the simple symbol table and the functions operating on it

codegen.c - this file contains few code-generation routines used in main file

errhandl.c - this file contains an error handler function

'tinyExpr' also has a small interpreter to load and execute the bytecodes generated. Such full fledged interpreter is also known as 'virtual machine' because the mnemonic codes are executed in a simulated manner and the behavior of the bytecodes can be made platform independent because of this.

The module distribution for this 'tinyExpr' interpreter is as follows,

interpre.h - the function declarations and inclusion of standard header files

mnemonic.h - this header file contains the bytecode values of various mnemonics used, and is the same one used in the ‘tinyExpr’ compiler also.

intrepre.c - this is the main interpreter file where the interpreter loop is there and the mnemonic functions are implemented here.

ostack.c - this program has operand stack (where operands are stored and evaluated) and functions operating on it

error.c - this program has a tiny error handler routine

To give an example of valid ‘tinyExpr’ expression:

```
a = a + b * c
```

Or expressions like:

var1 = 10 , var2 = 20 , var3 = (var1 + var2) * 20 , write (var3)

can be written. Since the variables are implicitly declared, any variable name can be just used. However it should be noted that the variable names are case-sensitive. Here the variables var1, var2 are initialized and then used to initialize the value of another variable var3 which is subsequently printed using the API ‘write’. It is the only API supported in ‘tinyExpr’ as of now.

15.5.1 The Lexical Analyzer

'tinyExpr' requires that the lexical elements to be explicitly separated by white space. For example, for giving the expression a*b+c it has to be given as a * b + c, explicitly.

The 'tinyExpr' compiler implements a simple lexical analyzer. It employs the standard library function strtok() to separate the tokens that is available in the source expression. Since this method is use instead of writing a lexical analyzer from scratch, the lexical analyzer becomes very small and its structure, how it works and its use become very clear and serves the purpose on-hand. The function of the lexical analyzer is thus simplified significantly because the user while giving the expression itself explicitly separates the lexical elements.

The lexical analyzer gives unique ID in the form of enumerator constants to all the tokens. The tokens are returned to the parser. It modifies two global values that contain the name of the identifier and the integer value of the current token that is analyzed.

For the expression a + b * c it returns,

```
IDENTIFIER // with side-effect of modifying the global
           // variable 'currIdentifier' to contain 'a'
PLUS       // the operator plus
IDENTIFIER // with side-effect of modifying the global
           //variable 'currIdentifier' to contain 'b'
STAR       // the multiplication operator
IDENTIFIER // with side-effect of modifying the global
```

```
// variable 'currIdentifier' to contain 'c'
```

15.5.2 Recursive Descent Parser

'tinyExpr' employs a recursive descent parser to generate the intermediate code. In this parser the scanner becomes just a function that it is called whenever a new token is required. It integrates the functionality of syntax analysis, semantic analysis and code-generation.

The example grammar production for additive-expression we saw also is there in this compiler. Let us see how it is implemented,

```
void additiveExpr()
{
     multiplicativeExpr();
     while(token==PLUS||token==MINUS)
     {
          int index=token;
          advance();
          multiplicativeExpr();
          if(index == PLUS)
               fprintf(outFile,"%c",IADD);
          else
               fprintf(outFile,"%c",ISUB);
     }
}
```

As you can see it looks just as similar to what we have done previously. The extra information is for integrated code-generator. As the process of parsing goes, the code is generated in parallel, and is stored in an intermediate file for later execution by the interpreter. Here in this example, the two mnemonics associated with the additive-expression are IADD and ISUB that stands for integers-addition and integers-subtraction respectively. When the parser sees the + or – symbol, it acts just as to generate the code for the corresponding symbols. ‘token’ stands for the representation of the token that is returned from the lexical analyzer.

15.5.3 The ‘tinyExpr’ Interpreter (the virtual machine)

Coming back to our expression a = a + b * c, as we have seen, it is converted as intermediate code to,

```
iload_0
iload_1
iload_2
imul
iadd
istore_0
```

This is available in the intermediate code file “a.out”. The bytecodes are loaded into memory and are to be executed.

The variables are assigned numbers based on the appearance of the variables in the expression. "iload_0" refers to "load the integer value of the variable numbered 0".

This code is for a stack-oriented machine. So the variable is loaded into the runtime stack while execution.

Similarly the values of b and c are also loaded into the stack. After that the instruction "imul" is encountered.

"imul" refers to "integer multiplication" and has the effect of popping two integers on the top of the stack, multiply the result and push them back on the stack. So the total result is of replacing the top two values in the stack by their multiplied value. Now the stack contains two integers. The value of 'a' in the bottom and the multiplied result of b and c in the top of it.

Now the "iadd" is encountered and it has the similar functionality as of "imul". It pops the two values that are on the stack, adds them and pushes the result in the stack.

Now the stack contains the end result of evaluating the expression a + b * c.

The final mnemonic code istore_0 pops the result from the operand stack and stores it in the local variable 'a' (whose index is 0, that's what he suffix _0 of the opcode istore_0 is meant for).

15.5.4 Extending 'tinyExpr'

The current implementation of the expression compiler is very limited in many ways. For example it doesn't support even statements and Boolean expressions. Looping constructs and other features can be added easily if labels are supported. For example the 'if' control structure can be implemented in the following way,

```
if(i<j)
      ++i;
```

It can be compiled to bytecode as follows,

```
  iload_0
  iload_1
  if_icmpl T_0
F_0 :
  iconst_0
  goto E_0
T_0 :
  iconst_1
E_0 :
  ifeq OUT_OF_IF_1
  iinc 0 1
  iload_0
OUT_OF_IF_1 :
  return
```

Similarly the functions can be supported and the list of extensions possible is endless. See [Lindholm and Yellin 1990] for more information.

16 UNIX and WINDOWS PROGRAMMING IN C

The genius of the UNIX is its framework, which enables programmers to stand on the work of others.

- 1983 Turing Award Selection Committee

C was originally intended to be a system programming language is still widely used for writing OS and compilers. Most of the Windows programming is done today using MFC and C++. On the other hand, most of the C programmers are well versed in using it for programming in UNIX. So this chapters provides a short session on how the two of the famous operating systems that makes use of the power of C.

16.1 UNIX and C

UNIX systems normally provide compilers for languages such as BASIC, FORTRAN, Pascal and Ratfor etc. Such compilers translate the language code to C intermediate code. The ordinary C compiler then compiles this C intermediate code. Advantage of using such approach is that new compilers are particularly easy to write. It is just enough to write a translator from that source language to C. All the issues such as portability are taken care by the C compiler that is working at the backend. Another big advantage is that the code written in different source languages can be integrated with

each others with no problem and can be interfaced with various applications written in various languages such as graphics programs, utilities etc.

UNIX also provides utilities such as LEX and YACC (Yet Another Compiler Compiler). LEX is a lexical analyzer generator and YACC is a parser generator both generating C code as output.

16.1.1 Benefits of UNIX being written in C

There were major benefits associated with writing the whole UNIX operating system in C. D.M. Ritchie [Ritchie 1978] lists the benefits by using C as the implementing language,

"…One of the reasons why UNIX is so popular is because of its myraid of software tools it provides to the users of the UNIX. Writing such software packages was possible only because of using high-level languages like C. These software packages, 'that would have been never written at all if their authors had to write assembly code; many of our most inventive contributors do not know, and do not wish to learn, the instruction set of the machine'.

Writing the code for operating systems involve thousands of lines of code that is hard to maintain and debug if assembly language were used. 'The C versions of programs that were rewritten after C became available are much more easily understood, repaired, and extended than the assembler versions. This applies especially to the operating system itself. The original system was very difficult to modify, especially to add new devices, but also to make minor changes'.

UNIX runs in various platforms and its portability is attributed to C. ‘An extremely valuable, though originally unplanned benefit of writing in C is the portability of the system... It appears to be possible to produce an operating system and set of software that runs on several machines and whose expression in source code is except for few modules identical on each machine’.”

To have a taste of how C can be used directly by providing special header files for UNIX, lets see about semaphores in C.

16.1.2 Semaphores in C

C under UNIX has semaphores and the functions are defined in the header files <sys/types.h>, <sys/ipc.h> and <sys/sem.h>.

‘semget’ is a function used to create a semaphore.

```
semctl(newsem,value);
```

(re)sets newsem to value.

```
semop(semaphor_send,…);
```

will send a signal for functionality like semaphore_wait and semaphore_send (corresponding to wait and signal of semaphores).

16.1.3 System calls in UNIX

System calls are used to access the services from the kernel. For the programmer it is just like a library function and invokes it in the same way as C functions are called.

If you program for UNIX in C, you will be using the UNIX specific functions innumerable times. If the code for the function is included in the executable code, that

will unnecessarily bloating the code size. To avoid this overhead, in UNIX, you have system calls. You just declare them and use it in your code, but the code will not be included in the executable file. Instead of that, the system calls will be called at runtime.

To put in other words, the essential difference between the C functions and system call is that when a program calls a subroutine the code executed is always part of the final object program (even if it is a library function). With the system call it is available with the kernel and not with the program (similar to the APIs in the Windows Programming).

System calls constitute primitive operations on the objects like file or processes and so are very efficient.

Although the standard I/O is very efficient they ultimately uses system calls only to achieve the functionality.

Point to Ponder:

Any process that interacts with its environment (however small way it is) must make use of system calls at some point.

16.2 Windows Programming in C

Windows is one of the most famous OS and we (programmers) to get optimal use of Windows can do programming. One important point to note is that the Windows APIs are written in C itself and Windows SDK fully uses C for Windows programming. Using C and the native APIs is not the only way to write programs for Windows. However, this

approach offers the best performance, the most power, and the greatest versatility in exploiting the features of Windows. Executables are relatively small and don't require external libraries to run (except Windows DLLs) as opposed to other approaches like using MFC for achieving the same functionality. In this chapter we are going to see how C is related to programming Windows and what we can learn from that.

16.2.1 'Hello World' in Windows Programming

```
     #include <windows.h>
     int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE
hPrevInstance, PSTR szCmdLine, int iCmdShow){
      MessageBox (NULL, TEXT ("Hello World"), TEXT
("FirstProgram"), 0) ;
     return 0 ;
     }
```

The starting function in Windows is WinMain as opposed to main in C. The output in Windows OS is through Windows APIs. It is no longer valid to direct output to devices such as stdout and stderr and get input using stdin (if you want such facilities like using plain C functions like gets, scanf, printf etc. you should do Windows Console applications). These devices can be taken for granted for being present, already opened and used in OS such as DOS and UNIX but not in Windows. File handling like the calls fopen to open a file etc. still works.

In your Windows program, you use the Windows function calls in generally the same way you use C library functions such as strlen. The primary difference is that the code for C library functions is linked into your program code, whereas the code for Windows functions is located outside of your program in the DLLs.

16.2.2 Hungarian Notation

As you may have noticed the arguments of the WinMain have very different beginnings (like sz in szCmdLine). This is a variable-naming convention called as "Hungarian Notation", attributed to Charles Simonyi. It just means that every variable name is prefixed with letter(s) denoting the data type of the variable. In szCmdLine, sz stands for "string terminated by zero".

Prefix	Data-type
By	Unsigned char – stands for Byte
C	Char
Dw	Unsigned long – stands for DoubleWord or Dword
I	Int
L	Long – stands for Long
S	String
Sz	String terminated by 0 character
Fn	Function
P	Pointer

Suggested Prefixes for Hungarian Notation

The preceding table lists the prefixes used for various types that may be useful for writing C programs.

Hungarian Notation helps in identifying the types from their prefixes and so can help in debugging and avoid mistakes. But as you can see it makes the programs less readable. The idea behind the Hungarian notation is that conveying information about the variable is important. And for that sometimes readability is affected. But it certainly suits for requirements like Windows programming where a couple of thousands of APIs are there such notation comes handy in understanding the types of the arguments. But for me it doesn't make much sense in using Hungarian notation for ordinary programs.

16.2.3 Derived data-types

Windows programming uses lots of 'derived data-types', beyond plain ints and chars. They are mostly typedefined and sometimes #defined. They are all in capital letters. Just look at the third argument PSTR in WinMain. It is a derived datatype that says it is a pointer to a string (char *). There are few non-intuitive derived types like LPARAM and WPARAM. Even though they look awkward to declare variables like,

```
HINSTANCE hInstance; or
WNDCLASS wndClass;
```

they serve a few important purposes and use a very good idea of C. They allow the variables to be defined by the programmer without any necessity to know how or what type it is defined as. They make usage abstract and encapsulate the structure definitions.

Near universally the Windows programmers use HINSTANCE without knowing what it is defined as. They serve to improve maintainability and portability. For e.g.

```
typedef struct {
int x;
int y;
} POINT;
```

was the definition of the POINT derived type when defined in Windows 3.1 a 16 bit OS. When Windows 95, which is a 32 bit OS came, POINT was redefined as,

```
typedef struct {
long x;
long y;
} POINT;
```

Again in 32-bit programming in Windows all the pointers are 32-bits, so all the pointers are capable of pointing anywhere in the memory. near and far pointers are just meaningless. This too was easily tackled by this approach.

The Windows programs written for Windows 3.1 still remained valid and remain unchanged even if radical changes like this were made.

Windows runs not only English but also various languages like Chinese, Italian etc. and for that Unicode is used.

```
#ifdef UNICODE
     typedef WNDCLASSW WNDCLASS;
#else
```

```
        typedef WNDCLASSA WNDCLASS;
    #endif
```

This defines WNDCLASS to be either supporting Unicode (WNDCLASSW, where W stands for ‘wide’) or the old one supporting ASCII (WNDCLASSA, where A stands for ‘ASCII’). And the programs still remain unchanged even-if whole internal representation changes. The porting and maintenance thus becomes easy.

16.2.4 DLLs in Windows

In C the linker links all functions used in the source program with the corresponding code and becomes part of the EXE file generated. The EXE files are thus large in size but they are very fast to execute (because the code for the functions is contained within). This is called as static linking. Whereas in Windows programming, the calls to the API are only provided. The linker just attaches the name of the DLL and the calling information with the EXE. This is called as dynamic linking. This makes the EXE code generated very compact. The code for the APIs is present in the DLLs that are available in every system installed with Windows. They are called whenever the program is loaded into the memory and those functions are needed to be executed. So the code is not redundantly stored in every file that is present in the system. When updating is needed is enough to update the DLLs and all the applications that are using the APIs are now with updated functions!

An interesting question to ask is, “if Windows use DLLs that will be available only at runtime, can their names be used for assigning to function pointers?”

The answer is Yes. Calling of such functions using function pointers whose value is determined at runtime is known as 'call back'. This may be indicated by a special keyword CALLBACK in some compilers supporting Windows programming.

16.2.5 The Concept of Versioning

When a Windows application program is written it uses lots of Windows API. If in the newer version of Windows any change in that API should affect that application. i.e. it might fail if it encountered newer or older version of API than it might expect. As the functions are modified to contain the improved functionality, the interfaces need to be changed sometimes. Such change in interfaces will make the applications making use of old interfaces have to be recompiled to support the functions with modified interfaces or else the applications become invalid. This is called as versioning problem.

The problems due to versioning occur because it is not the part of the language itself. This is a general problem encountered in maintaining any application. Lets look at it in the case of Windows programming with the help of a historical example.

MoveTo was a graphics function in 16-bit versions of Windows to set the current position for drawing,

```
unsigned long MoveTo(HDC, int , int);
```

First argument is the device context handle (that is used as a control for drawing purpose) followed by values of x- and y-coordinates. It returns the previous current position and is packed as two 16-bit values (unsigned long takes 32-bits in 16-bit version).

In the 32-bit versions of Windows, the coordinates are 32-bit values. So the problem occurred because the 32-bit versions of C was not available with 64-bit integral type. This meant that the previous current coordinate cannot be returned as 64-bit (two 32-bit values packed together) and the function interface needed to be changed. The solution is to declare the new function as follows,

```
BOOL MoveTo (HDC, int , int, LPPOINT );
```

The last argument is a pointer to a POINT structure that contains previous current position after its call.

Had C had the capability of function polymorphism, there would be no problem and both the old and new MoveTo functions will coexist. So the interface changes and such change will require all the applications that use MoveTo to be recompiled to use the improved version of MoveTo. The alternate solution is to change to the new name *MoveToEx* with this modified interface and retain the old MoveTo function as it was. That is what introducing the MoveToEx function having declaration exactly does in this case,

```
BOOL MoveToEx (HDC, int , int, LPPOINT );
```

Thus the problem of introducing the new functionality was solved without affecting the old ones. The new MoveToEx need not be written again and can work as a wrapper function (that is already discussed). Lot such other examples are there in Windows programming and one such is discussed here to explain the versioning concept.

Most of the versioning problems can be avoided if the interfaces are designed properly. Interfaces should be designed keeping future revisions in mind and the t should be . Lets look at a similar situation in C standard library and how that problem is tackled:

```
div_t div (int num, int denom);
```

where div_t may be declared as,

```
struct div_t {
     int quot;
     int rem;
     };
```

The standard library function div (declared in <stdlib.h>) returns object of type div_t to contain quotient and remainder and not of type long which may contain two integers in upper and lower parts of the long.

Applying the same idea to MoveTo function, a careful design would have looked like this,

```
POINT MoveTo(HDC, int , int);
```

where POINT is a derived type(typedefed structure) that is readily available in Windows! Thus a problem of changing the interface would have been avoided in later stages.

The naming solution in case of Windows is not the best one either. Let's say that some more modifications are required to the function MoveToEx in later date. How shall it be named? MoveToExEx, MoveToExExEx… etc.? The better naming would be MoveTo, MoveTo2, MoveTo3 etc. And this kind of naming convention is of course a matter of taste.

17 THE NEW C9X STANDARD

Oh what times! Oh what standards!

-Marcus Tullius Cicero

The new standard C9X is defined and is very recently got approved. This is a significant development to the C community. As this book is written there are no compilers conforming to this new standard. It formalizes many practices, improves upon the previous ANSI standard that referred to as C89 because this first standard for C was approved in 1989.

One of the basic goals of the C9X committee was to make sure that the old code remains legal and remain unaffected in the new standard. It adds long awaited basic facilities like single line comments and mixing of declarations and statements.

This chapter is not a exhaustive coverage of changes and only discusses the most important of the changes and that are introduced in C9X. The base documents for this chapter is [ANSI C 1998] and [Rationale ANSI C 1999] and for more information the readers are encouraged to go through these documents.

17.1 Principles of Standardization Process

Standardization process is not redesigning the language or modifying the language, as opposed to the popular belief. Its main aim is to promote portability and give a common standard that *codifies the common existing practice.*

17.1.1 Overall Goal

The Committee's overall goal was to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments [Rationale ANSI C 1999].

17.1.2 Underlying Principles

The committee considered and followed several principles in achieving the overall goal of specifying the standard of the language. Most important of these principles listed here are based upon the ideas given in [Rationale ANSI C 1999]:

- Existing code is important, existing implementations are not.

When change is to be suggested the most important criteria to be considered is the change of code. The existing code should not get or should be least affected. But if the same change can be achieved by affecting the

change in the existing implementations (say compiler implementations), it is preferably be done.

- C code can be portable

C has been successfully ported to varied variety of operating systems and platforms so the standard tries to make the language as widely implementable and as general as possible.

- C code can be non-portable

The ability to write non-portable code is always a distinguishing mark and can get the maximum utility from any target system. The standard doesn't want to make C not possible to be used as 'high-level-assembly' language due to standardization.

- Avoid "quiet changes."

Any changes should be visible and the implementations should be able to give clear diagnostic messages to the users. The change in the behavior of code without any evident reason and diagnostic is strictly avoided.

- A standard is a treaty between implementor and programmer

This is the basic reason why the standardization process is required, exists and developed and so the committee understands this fact.

- Keep the spirit of C

The changes can be done to the language but it should not violate the basic design principles involved in the language.

17.2 Most Important Changes by C9X

17.2.1 New Keywords and Types

The new keywords introduced are:

inline, restrict, _Bool, _Complex and _Imaginary.

The new types added in C9X are:_Bool, long long, unsigned long long, float _Imaginary, float _Complex, double _Imaginary, double _Complex, long double _Imaginary, long double _Complex. Note that many of these new types uses already available keywords.

The new type long long that has at-least 64-bit precision is introduced in C9X primarily to support the new hardware available:

1) The disk-capacity has increased a lot and there is a requirement that its memory be directly addressed.
2) 64-bit microprocessors are becoming common that the new type will help exploit its power.

A new type `long long` is useful in platforms where the size (int) == 2 and sizeof(long) == 4. Even in other machines having greater word size the integral values

occupying more than the sizeof(long) bytes may be required. In such cases there is a requirement to represent integral value occupying 8 bytes and the new type satisfies that.

Many implementations provide a header file like <complex.h> to provide the functionality of complex numbers because C is also used heavily in the mathematical and scientific areas and there complex types are often required. For that C9X introduced the complex type as basic data-type itself.

17.2.2 Mixing Declarations and Statements

Inside the blocks mixing of declarations and statements is allowed. Again this is from C++.

```
int i;
…. // after lot of code,
printf("%d",i);
 // by the time of point of usage,
 // we forget that i is not initialized.
```

The problems of forgetting to initialize is the mistake almost every C programmer makes. Since in most cases, the value to be initialized is available only in the later part of the code.

It is a very natural to forget initializing the variables. And the simple rule now with C9X standard is *to declare the variable and initialize it just before its use*. With this simple rule it becomes very convenient for the programmer to declare and use variables throughout the block.

Mixing declarations and initialization leads to another serious problem. It is due to goto statements. What should happen if the declarations are bye-passed by a goto statement?

```
goto out:
int i=0;
out: printf("i=%d", i);
// prints i = 'some garbage value'
```

The standard says that *a jump forward past a declaration leaves it uninitialized*. Similarly, what if the declaration is encountered twice?

```
in:
int i=0,j;
printf("i=%d, j=%d", i ,j);
// when it is encountered the second time it prints
// i=0, j= 'some garbage value'
j=20;
goto in;
```

If the declaration has the initialization value, then it is initialized twice. If the variable is un-initialized, it may be set to some garbage value.

17.2.3 Declarations in 'for' Statements

The 'init' part of the 'for' loop can have declarations. The scope of the variables thus declared is only within the loop.

```
int i;
```

```
for(i=0; i<10; i++)
     printf("%d", a[i]);
```

Here the variable i is required only within the block. This is the case for most of the 'for' loops we write in C. Declaring the variable outside the 'for' loop makes the scope of the variable to the enclosing block. Allowing the variables be declared inside the 'for' loop makes programming easier.

```
for(int i = 0; i<10; i++)
     printf("%d", a[i]);
```

is convenient.

17.2.4 Compound Literals

Consider the problem of finding the biggest element in an array.

```
int tempArr[] = { 6,1,8,3,4,10};
int j = biggest(tempArr);
```

In such cases it would be helpful if you can pass the array directly without creating a temporary array just to call the function.

For this C9x introduces the idea of compound literals. Compound literals provide a mechanism for specifying constants of aggregate or union type. So no temporary variables are required when an aggregate or union value will be needed only once. For example, an array can be created in the argument itself and passed to the function:

```
int j = biggest( (int []){ 6,1,8,3,4,10} );
// using compound literals.
```

Size can be specified while casting also:

```
ptr = (int[5]){9, 8, 7, 6, 5};
```

17.2.5 Variable Length Arrays

C9X adds a new array type called a variable length array type. The inability to declare arrays whose size is known only at execution time was often considered as one of the major drawbacks associated with C. By the inclusion of variable length array this restriction is removed. So now it is not necessary for having one's own implementation for growing/variable length arrays.

```
void sizingFun(int n)
{
int array[n];  // is possible
....
}
```

The number of elements specified in the declaration of a variable length array type is a runtime expression. Previously the length of the arrays was fixed and the array dimension was required to be a constant expression.

It should be noted that since the space for variable length array is allocated at runtime, it couldn't be static or extern. But a pointer to the variable length array can be static.

```
static int array[n];
// error, variable length arrays cannot be static
```

sizeof operator can be applied to such arrays to find the sizeof the array,

```
int sizeOfVarArr = sizeof( int (*)[n] );
```

It can be specified if the argument passed to a function is a variable length array or not with the notation:

```
void varLenArrFun(int vArr1[static 10], int vArr2
[static 10]);
```

This declaration assures that the variable length arrays passed are surely of length at-least 10 elements, and the static indicates just that.

We already saw that, to specify that the arrays are non-overlapping, restrict qualifier is used.

```
void varLenArrFun(int vArr1[static restrict 10], int
vArr2[static restrict 10]);
```

So this specifies that the arrays passed are of length at-least 10 (so it indirectly specifies that the address passed cannot be NULL) and they are non-overlapping.

And as usual the const can be used to specify that the pointer always will point to the same array.

```
void varLenArrFun(int vArr1[static const 10], int
vArr2[static const 10]);
```

This is similar to what we used to declare as for passing array as pointers:

```
void varLenArrFun(int *const arr1, int *const arr2);
```

The function prototype for using variable length argumenst is similar:

```
void varLenArrFun(int vArr1[*], int vArr2[*]);
```

Similar syntax is used for casting variable length arrays:

```
int (*vArrPtr)[n] = (int (*)[n]) somePtr;
```

typedef for variable length arrays is made in the following example:

```
typedef int iVarArr[n];
iVarArr array; // array is of type int (*)[n]
```

Now lets consider that value of 'n' is changed:

```
n+=5;
```

Now what happens to the declarations like this:

```
iVarArr another;
// array is of type int (*)[n] and not of type int (*)[n+5]
```

So the type remains the same and such side-effects doesn't affect typedefs.

17.2.6 Variable Length Macro Arguments

One of the main design consideration C9X for macros is that there should be some way provided such that what-ever that is possible to do with functions should also be made possible using macros. Functions have variable length argument passing mechanism and C9X introduces the same way of using ellipsis in macro definition to indicate variable length list.

The identifier `__VA_ARGS__` is used to replace such variable length list in C9X. For example:

```
#define varLenMacro(filePtr,...) \
     fprintf(filePtr, "VarLenList: "__VA_ARGS__)
```

and can be called like:

```
varLenMacro(stdout,"%d",someThing);
```

this will be expanded as:

```
fprintf(stdout, "VarLenList: ""%d",someThing);
// due to stringization it becomes
fprintf(stdout, "VarLenList: %d",someThing);
```

Additional rule for variable length macro list is: *there must be at least one argument to match the ellipsis*.

17.2.7 The 'restrict' Qualifier

[ANSIC-88] added 'const' and 'volatile' qualifiers to the language. C9X adds the 'restrict' type qualifier to allow programs to be written so that translators can produce significantly faster code for certain cases.

What is the requirement of this new qualifier? For that remember what we have discussed of the difference between the following two library functions:

```
void *memcpy(void * s1, const void * s2,size_t n);
void *memmove(void * s1, const void * s2, size_t n);
```

"The only difference between these two functions is that memmove() can be used with overlapping memory area, whereas memcpy() for non-overlapping memory areas (of course with some loss of efficiency)"

In other words, the problem memmove suffers is called as the problem of 'aliasing'. The implementation becomes slow because it cannot assume that the locations pointed by the

’s1’ and ’s2’ are disjoint and so cannot do optimizations on it. Consider the case of the memcpy(). It explicitly specifies that the ’s1’ and ’s2’ are of disjoint arrays and so efficient code can be generated for it.

This is just one such example of ’aliasing’. If a translator cannot determine that two different pointers are being used to reference different objects, then it cannot apply optimizations such as maintaining the values of the objects in registers rather than in memory, or reordering loads and stores of these values. If does so it without considering ‘aliasing’, that may give rise to wrong results. In other words, the compiler cannot do much optimization because of that optimization may affect the alias. For example:

```
void f1(int * a1, const int * a2,int n)
{
int i;
for ( i = 0; i < n; i++ )
     a1[i] += a2[i];
}
```

here there is a chance that the pointers a1 and a2 refer to the same position. So optimization cannot be done on this. For such cases if the restrict qualifier is specified then it means that the pointers a2 and a1 are the pointers that primarily point to the memory and aliases point only ’through’ them. Since both are restrict that means they should be the primary means of the pointing the area, so no chance of overlapping.

Now the function looks like this:

```
void f1(int n, float * restrict a1, const float * restrict a2)
```

This permits optimizations be done by the compilers generating code that is many times faster.

With the restriction visible to a translator, a straightforward implementation of memcpy in C can now give a level of performance that previously required assembly language or other non-standard means. Thus the 'restrict' qualifier provides a standard means with which to make, in the definition of any function, an aliasing assertion of a type that could previously be made only for library functions. The 'restrict' keyword allows the prototype to express what was previously expressed by words [Rationale ANSI C 1999].

So the new prototype for the memcpy looks like this:

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

Now lets look how the ordinary declarations be made with restrict qualifier. To specify that the integer pointer is restricted, you should declare as:

```
int *restrict rp;
      // read as 'rp is a restricted pointer to integers
```

This is different from

```
restrict int *rp;
      // reads as 'rp is a pointer to restricted int' that is invalid
```

With the addition of restrict to list of qualifiers, the ‘cv qualifier’ (stands for: ‘const-volatile qualifier’) now becomes ‘cvr qualifier’. Most of the semantics of ‘cv qualifiers’ apply to ‘restrict’, for example:

```
typedef float *floatPtr;
restrict floatPtr ptr;
// ok. Similar to what we did for const.
```

Point to Ponder:

‘restrict’ is only for additional optimization, so it can safely be deleted from the program.

17.2.8 The ‘struct hack’ technique standardized

The ‘struct hack’ technique we saw in the chapter on ‘structure and unions’ is a useful one. Since this is a popular and widely used technique, C9X has standardized the method of such usage. So ‘struct hack’ is now valid code but with some little change in the syntax.

The last member of structures now may have an incomplete array type.

Consider the structure,

```
struct system{
      char type[10];
      char manufacturer[20];
      int numOfPeripherals;
      // this field has the number of items that are pointed by
      // the following ‘nameOfPeripherals’ member.
```

```
        int  perpheralID[];
    };
```

Here the size of the member ‘peripheralID’ can differ according to the requirement. But (as the standard specifies) only the last element can be such member. This feature is now called as “flexible array members”.

Another point to be noted is that, sizeof applied to the structure ignores the array but counts any padding before it. This makes the malloc call as simple as possible. That is the allocation is as follows:

```
    int num = 5;
    structPtr = malloc( sizeof(struct system) + num * sizeof(int));
```

Here note the change that it is num * sizeof(int) and not (num-1) * sizeof(int) because of the reason that the sizeof operator doesn’t include the ‘flexible array member’ making the malloc simple.

17.2.9 Inline Functions

Inline functions are now possible with the help of the keyword ‘inline’ (adapted from C++). It should however be noted that the ‘inline’ keyword is a is a *function-specifier* that can be used only in function declarations.

This is to make the small functions inline such that the overhead for function calls is avoided. This an alternative for using macros and by using such inline functions the disadvantages in using macros is avoided (macros Vs functions is discussed in chapter on preprocessor).

17.2.10 Designated initializers

Designated initializers provide a mechanism for initializing structures, unions and arrays by name.

Previously unions can only be initialized with their first member. But now with the help of designated initializers, unions can be initialized via any member, regardless of whether or not it is the first member.

17.2.11 Line-comments // as in C++

This one of the most basic and expected facilities the C programmers have longed for. For writing short comments single line comments are very convenient. It is a nasty bug (that is frequently made) is to forget to give the closing */ comment for starting /*. With the new // comment that is borrowed from C++ will help programmers to easily write single line comments.

17.3 Other Significant Changes in C9X

- Significant improvement is support for Unicode characters in the form of \u and \U followed by four hexadecimal digits.

  e. g. `\Uaa00`

  to specify some Unicode character.

- The return type int as default type in functions (if omitted) is no longer legal.

  This formalizes the idea that the return types have to be explicitly specified. This will help avoiding errors due to implicit assumption of return types (discussed in chapter on functions). This will also help increase readability.

- Using function prototypes was a desirable feature with [ANSIC-88] but now using them becomes a necessity. Because, implicit function declarations is removed in C9X.

- true and false are predefined macros

- C9X introduces *predefined identifier*s, which have block scope. This is similar to the predefined macros such as __FILE__, but these are identifiers (also note that the predefined macros have file scope. E.g. `__func__`, that allows the function name to be used at execution time.

```
void funName()
{
if(someError)
    printf("Error at function : %s",__func__);
}
// prints
// Error at function : funName
```

- hexadecimal notation is now available for floating constants.

- bool is typedefined in <stdbool.h>

  Using integers for booleans as in traditional C has lot of advantages and disadvantages. Now integers need not do all the work of the boolean types since a bool is available now.

APPENDIX I – ANSWERS TO SELECTED EXERCISES

The numbers before the answers refer to the question numbers in the text.

0.2 In BSS.

```
static int i = 0;
static int i;
```

both are equivalent. The variable ‘i’ being explicitly initialized to 0 in the first case makes no difference.

0.4 Compile time error. Constant expressions are evaluated at compile-time itself.

2.2 Yes. Char data type can take part in expressions much like as int data type. If the char is unsigned then no problem arises. However, for the other case a problem arises. Because the representation of signed variable in the memory (bit pattern) cannot be predicted and so is not portable.

2.5 Yes.

2.6. Yes. The keyword to remember is non-decreasing order of size and this is the only restriction laid for the size of these data types. Equal sizes for all these types do not violate this, and so is possible.)

2.7. The output depends on whether the implementation follows arithmetic or logical shift.

2.8 Although some implementations allow it, it is not assured that the value will be changed. It may even produce a runtime error.

2.10 No.

2.11 Most of the implementations typedef time_t as int or as long int. If the sizeof long int or int happens to be 4 bytes, there is a problem. Because time_t stores number of seconds lapsed after midnight of Jan 1, 1970 and can contain upto 2^32-1 seconds. The number of years that can be represented like this ends up at Jan 18, 2038. So the next day of this day will be interpreted as Jan 1, 1970, and this is referred to as year 2038 problem. This problem can be solved by typedefining time_t with an integeral type occupying more than 4 bytes.

2.12 The output depends on whether value or signed preserving is followed. sizeof returns size_t. If it is ‘typedef’ined as unsigned int then there is a problem of value/sign preserving.

3.2 No Linkage

3.4 No, because the local/register variables have no linkage and are allocated in stack and microprocessor registers respectively. Linker links the name and its associated space at link time and since local/register variables are available at runtime, they cannot be extern.

4.4 The functions can never pass an array. So in this function actually an array pointer is passed (it happened here that size was 2 bytes both for int * and int).

4.5 Compiler error. sizeof cannot be applied on void. Since the function returns void, the function call expression becomes a void expression.

5.2 If the body of the loop never executes p is assigned no address. So p remains NULL where *p =0 may result in problem (may rise to runtime error NULL pointer assignment)

7.1 If it were a small-endian machine it will print the ASCII equivalent of 0 i.e. 48. If it were a big-endian machine, it will print the value equivalent of reversing the bytes.

9.1
```
while(!feof(fp)){
      ch = fgetc(fp);
      if( ch=='/' )   /*if chance for beginning a comment */
      {
```

```
        ch=fgetc(fp);
        if(ch=='*')
        {
        while( ch=fgetc(fp),!feof(fp) ){
                if(ch=='*')
                if(checkif(fp,'/')){
        ch=fgetc(fp);
        break;
        }
       }
)
```

9.2 fread reads three records successfully. It will return EOF only when fread tries to read another record and fails reading EOF (and returning EOF). So it prints the last record two times and comes out after seeing EOF.)

10.1 The second one is better because gets(inputString) doesn't know the size of the string passed and so, if a very big input (here, more than 100 chars) the characters will be written past the input string. When fgets is used with stdin performs the same operation as gets but is safe.)

10.2 If the str contains any formatting characters like %d then it will result in a subtle bug. So always prefer the first one.

12.1 ANSI C mandates atleast one member to be present in a structure, so this should issue an error in C. This is acceptable in case of C++; but an object should be associated with some memory and the object should be of size atleast 1; so the printf may print the value 1 or a bigger number as the size of the structure.

APPENDIX - II 'TINYEXPR' EXPRESSION COMPILER

& INTERPRETER IMPLEMENTATION

```
/*********************************************************************
This "tinyExpr" expression compiler and interpreter implementation is a
sample implementation for explaining the concepts presented in the
chapters "Compiler Design in C" and "C and Java". This indents to be a
simple and easy to understand implementation for a compiler and a
runtime interpreter and is not intended to be a full-fledged one. For
example support for error detection and recovery are very poor.
Similarly, it is little bit platform dependent too.
```


Author: S G Ganesh

Date : 3-4-2001

```
*********************************************************/

/*************************** mnemonic.h ***********************/

/*Header file for opcode mnemonic constants used both in the 'tinyExpr'
compiler and interpreter. These bytecodes and equivalent mnemonics are
as in Java */

enum mnemonic{
      ICONST_M1 = 2,
```

```
ICONST_0 = 3,
ICONST_1 = 4,
ICONST_2 = 5,
ICONST_3 = 6,
ICONST_4 = 7,
ICONST_5 = 8,
BIPUSH   = 16,
SIPUSH   = 17,

ILOAD    = 21,
ILOAD_0  = 26,
ILOAD_1  = 27,
ILOAD_2  = 28,
ILOAD_3  = 29,

ISTORE   = 54,
ISTORE_0  = 59,
ISTORE_1  = 60,
ISTORE_2  = 61,
ISTORE_3  = 62,

IADD      = 96,
ISUB      = 100,
IMUL      = 104,
IDIV      = 108,
IREM      = 112,
INEG      = 116,
```

```
      ISHL      = 120,
      ISHR      = 122,
      IAND      = 126,
      IOR       = 128,
      IXOR      = 130,
      IINC      = 132,

      INVOKEMETHOD =186,
      RETURN    = 177,
      };
```

```
/*****************code for the compiler part ***********************/

/*************************** tinyexpr.h ***********************/

/* all the standard header files required are included here */
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include <limits.h>

/* function declarations from the lexical analyser module (lex.c) */
typedef enum lexVals lexToken;
```

```
lexToken searchSymbols(char *tokenString);
lexToken advance(void);
int verify(enum lexVals expected);
lexToken getTokenFromString(char *tokenStr);

/* function declaration from the error handler module (errhandl.c) */
lexToken errorHandler(enum errorType errType, char *str);

/*  function declarations from the main module of recursive descent
parser
      and integrated code generated modules (tinyexpr.c) */
void statementExpression();
void preIncrementExpression();
void preDecrementExpression();
void postIncrementExpression();
void postDecrementExpression();
void assignment();
void expr();
void assignmentExpr();
void assignment();
int isEquOperator(int index);
void inclusiveOrExpr();
void exclusiveOrExpr();
void shiftExpr();
void additiveExpr();
void andExpr();
void multiplicativeExpr();
```

```
void unaryExpr();
void preIncrementExpr();
void preDecrementExpr();
void unaryExprNotPlusMinus();
void postfixExpr();
void primary();
void name();
void postIncrementExpr();
void postDecrementExpr();
void methodInvocation();

/*  function declarations from the few code generated modules
(codegen.c) */
void emitPreIncDecCode(int offset,int incOrDec);
void emitNumConstCode();
void emitLoadCode(int offset);
void emitStoreCode(int offset);



/*************************** errhandl.c ***********************/

enum errorType{WARNING, ERROR};
enum lexVals errorHandler(enum errorType errType, char *str){
      if(errType==WARNING)
            {
            fprintf(stderr,"\n WARNING : %s",str);
            return advance();
```

```
            }
      else /* error */
            {
            fprintf(stderr,"\n ERROR   : %s\n",str);
            fprintf(stderr,"\n Exiting... Press any key ");
            getchar();
            exit(0);
            }
}

/*****************************lex.c**********************************/
/* this is the source file for the simple lexical analyzer */

#define MAXLEN 80

enum bool{false=0,true};

int initLex = true;     /* to initiate the lexical analyser */

extern char sourceExpr[MAXLEN];
/* this is the string that holds the expression that is to be compiled
*/

enum lexVals token;     /* this holds the current token's lexical value
*/
char *currIdentifier;   /* the identifier string if it is a variable */
int  currInteger;
```

```
/* the integer value if the current one is a integer constant */

enum lexVals nextToken;
/* this lexical analyzer supports the look-ahead of one token */

/* these are the numbers given to the various lexical items */
typedef enum lexVals    {
      IDENTIFIER=0,     INTEGER_CONST,    EQUAL,            B_PARENS,
      E_PARENS,         LEFT_SHIFT,       RIGHT_SHIFT,      PLUS,
      PLUS_PLUS,        PLUS_EQ,          MINUS,
      MINUS_MINUS,
      MINUS_EQ,         STAR,             STAR_EQ,          DIV,
      DIV_EQ,           MOD,              MOD_EQ,           BIT_AND,
      BIT_OR,           BIT_NOT,          EXOR,
      LEFT_SHIFT_EQ,
      RIGHT_SHIFT_EQ,   EXOR_EQ,          OR_EQ,            AND_EQ,
      UNSIGNED_RIGHT_SHIFT, UNSIGNED_RIGHT_SHIFT_EQ,  COMMA,
      ILLEGAL_TOKEN = -1,
}lexToken;

/* these are the equivalent symbols to be matched to find the token */
static char *symbol[]=
{
      " ",        " ",        "=",        "(",
      ")",        "<<",       ">>",       "+",
      "++",       "+=",       "-",        "--",
      "-=",       "*",        "*=",       "/",
```

```
      "/=",       "%",        "%=",       "&",
      "|",        "~",        "^",        "<<=",
      ">>=",      "^=",       "|=",       "&=",
      ">>>",      ">>>=",     ",",
      " ",
};

enum lexVals searchSymbols(char *tokenString)
{
      enum lexVals i;
      for(i = EQUAL; i <= COMMA; i++)     /* simple linear search */
            if(!strcmp(symbol[i],tokenString))
                  return i;   /* return table location - one*/
      return ILLEGAL_TOKEN;
}

enum lexVals getTokenFromString(char *tokenStr)
{
   enum lexVals temp;
   if(tokenStr)
   {
      if(isalpha(tokenStr[0]))
      /* if it contains alphabetic characters, it is a variable name */
      {
      currIdentifier = tokenStr;
      temp = IDENTIFIER;
      }
```

```
      else if(isdigit(tokenStr[0])) /* if it starts with a digit then
                                       it is a integer constant */
      temp = INTEGER_CONST;
      currInteger = atoi(tokenStr);
      }
   else                        /* else search for the operator strings
*/
      temp = searchSymbols(tokenStr);
   if(temp == ILLEGAL_TOKEN)
      temp = errorHandler(WARNING,"Unknown symbol in the given
expression");
   return temp;
   }
   else
      return ILLEGAL_TOKEN;
}

enum lexVals advance(void)    /* discard current value of token and get
new
                              values by reading more */
{
   char *tokenStr;
   if(initLex)          /* if it is for the first time in the
expression */
      {
      initLex = false;
```

```
      /* strtok places a NULL terminator in front of the token, if
found */
      tokenStr = strtok(sourceExpr, " \t");
      nextToken = getTokenFromString(tokenStr);
      }
   token = nextToken;

   /* A second call to strtok using a NULL as the first parameter
returns a pointer to the character following the token */
   tokenStr = strtok(NULL, " \t");
   nextToken = getTokenFromString(tokenStr);
   return token;
}

int verify(enum lexVals expected)   /* used in places where some token
is
                                    required to be present */
{
 if(token == expected)
      {
      advance();
      return true;
      }
 else
      {
      errorHandler(ERROR,"Expected token is missing");
      return false;
```

```
	}
}

#ifdef LEX_DEBUG		/* only for debugging purpose */
int main()
{
int i;
clrscr();
while((i=advance())>=0)
	{
	if(i==0)
		printf("\n %s",currIdentifier);
	else if(i==1)
		printf("\n %d",currInteger);
	else
		printf("\n %s",symbol[i]);
	}
}
#endif



/*************************symtable.c****************************/
/* this program contains code for the symbol table management that is
used by the main module */

enum sizes{tableSize=32,tokenSize = 32};
```

```
/*    global variables required for symbol table. symbol table is just
an array of strings */
char symbolTable[tableSize][tokenSize];
static int symbolTop;

/* sets a new variable provided that name and type are available*/
int setVariable(char *string)
{
      if(symbolTop >= tableSize)
      {
      errorHandler(ERROR,"\n Internal - Symbol table overflow - Cannot
            insert entry");
      return -1;
      }
      else
      {
      strcpy(symbolTable[symbolTop],string);
      return symbolTop++;
      }
}

/* see in symbol table if the string is already been declared else set
it. employs direct linear search to find the symbol   */
int lookUpSet(char *string)
{
 int i = symbolTop-1;
 while(i >= 0)
```

```
      {
      if(strcmp(symbolTable[i],string)==0)
            return i;
      i--;
      }
 /* if it reaches here it means that the symbol is a new one */
 return setVariable(string);
}

/* the code modules in codegen.c , the code generation functions */

extern FILE *outFile;

void emitPrePostIncDecCode(int offset,int incOrDec)
{
      fprintf(outFile,"%c%d%d",IINC,offset,incOrDec);
}

void emitNumConstCode()
{
      if(currInteger>=0 && currInteger <=5)
      {
        switch(currInteger)
            {
            case 0: fprintf(outFile,"%c",ICONST_0); break;
            case 1: fprintf(outFile,"%c",ICONST_1); break;
            case 2: fprintf(outFile,"%c",ICONST_2); break;
```

```
            case 3: fprintf(outFile,"%c",ICONST_3); break;
            case 4: fprintf(outFile,"%c",ICONST_4); break;
            case 5: fprintf(outFile,"%c",ICONST_5); break;
            }
      }
      else if(currInteger >= -128 && currInteger<=127)
            fprintf(outFile,"%c%c", BIPUSH,currInteger);
      else if(currInteger >= INT_MIN && currInteger <= INT_MAX)
            fprintf(outFile,"%c%d", SIPUSH,currInteger);
}

void emitLoadCode(int offset)
 {
      if(offset < 0)   /*less than zero indicates unknown identifier */
             errorHandler(ERROR, "Undefined symbol in expression");
      else if(offset <= 3)
      {
            switch(offset)
            {
            case 0: fprintf(outFile,"%c",ILOAD_0); break;
            case 1: fprintf(outFile,"%c",ILOAD_1); break;
            case 2: fprintf(outFile,"%c",ILOAD_2); break;
            case 3: fprintf(outFile,"%c",ILOAD_3); break;
            }
      }
      else
            fprintf(outFile,"%c%d",ILOAD,offset);
```

```
}


void emitStoreCode(int offset)
 {
      if(offset < 0)/* less than zero indicates unknown identifier */
             errorHandler(ERROR, "Undefined symbol in expression");
      else if(offset <= 3)
      {
            switch(offset)
            {
            case 0: fprintf(outFile,"%c",ISTORE_0); break;
            case 1: fprintf(outFile,"%c",ISTORE_1); break;
            case 2: fprintf(outFile,"%c",ISTORE_2); break;
            case 3: fprintf(outFile,"%c",ISTORE_3); break;
            }
      }
      else
            fprintf(outFile,"%c%d",ISTORE,offset);
}

/* ********************************tinyexpr.c ************************/
/* this is the main source file that contains the implementation of the
   recursive descent parser */

#include "mnemonic.h"
/* mnemonic codes for bytecodes are available in this header file */
```

```
#include "tinyexpr.h"
/* this header file inturn includes standard header files and contains
function declarations*/

#include "errhandl.c"
/* this file contains a error handler function */

#include "lex.c"
/* a tiny lexical analyser is available in this file */

#include "symtable.c"
/* this file contains the simple symbol table and the functions
operating on it */

#include "codegen.c"
/* this file contains few code-generation routines used here */

char sourceExpr[MAXLEN];
/* this is the string that holds the expression to be compiled */

FILE *outFile;
/* this is pointer to FILE where the output bytecodes should be sene */

/*    functions available here represent grammar prodcutions.     these
functions collectively make the recursive descent parser. */
```

```
void statementExpression()
 {
/*statementExpression can be any one of these but only one is selected
and other are optional so the function are designed in that way */
      preIncrementExpression();
      preDecrementExpression();
      postIncrementExpression();
      postDecrementExpression();
      methodInvocation();
      assignment();
 }

void preIncrementExpression()
 {
      if(token==PLUS_PLUS)
      {
            int i;
            advance();
            i=lookUpSet(currIdentifier);
            emitPrePostIncDecCode(i,1);
            emitLoadCode(i);
            advance();
      }
 }

void preDecrementExpression()
 {
```

```
	if(token==MINUS_MINUS)
	{
		int i;
		advance();
		i=lookUpSet(currIdentifier);
		emitPrePostIncDecCode(i,-1);
		emitLoadCode(i);
		advance();
	}
 }

void postIncrementExpression()
 {
	if(token==IDENTIFIER && nextToken==PLUS_PLUS)
	{
		int i=lookUpSet(currIdentifier);
		emitPrePostIncDecCode(i,1);
		advance();
		advance();
	}
 }

void postDecrementExpression()
 {
	if(token==IDENTIFIER && nextToken==MINUS_MINUS)
	{
	int i=lookUpSet(currIdentifier);
```

```
      emitPrePostIncDecCode(i,-1);
      advance();
      advance();
      }
 }

void expr()
 {
      assignmentExpr();
      if(token == COMMA)
            {
            advance();
            expr();
            }
 }

void assignmentExpr()
 {
      assignment();
      inclusiveOrExpr();
 }

void assignment()
 {
      if( (token==IDENTIFIER) && isEquOperator(nextToken) )
      {
            int i,index;
```

```
            i=lookUpSet(currIdentifier);
            advance(); /* eat identifier name */
            if(token != EQUAL)
                  emitLoadCode(i);
            index = token;
            advance();
            assignmentExpr();
            switch(index)
            {
            case STAR_EQ  :  fprintf(outFile,"%c",IMUL); break;
            case DIV_EQ   :  fprintf(outFile,"%c", IDIV); break;
            case MOD_EQ   :  fprintf(outFile,"%c", IREM); break;
            case PLUS_EQ  :  fprintf(outFile,"%c", IADD); break;
            case MINUS_EQ :  fprintf(outFile,"%c", ISUB); break;
            case LEFT_SHIFT_EQ : fprintf(outFile,"%c",ISHL); break;
            case RIGHT_SHIFT_EQ: fprintf(outFile,"%c",ISHR); break;
            case AND_EQ   :  fprintf(outFile,"%c",IAND); break;
            case EXOR_EQ  :  fprintf(outFile,"%c",IXOR);break;
            case OR_EQ    :  fprintf(outFile,"%c",IOR);break;
            }
            emitStoreCode(i);
      } /* end switch */
 }

int isEquOperator(int index)
 {
```

```
      if(index==EQUAL || index==STAR_EQ || index==DIV_EQ ||
index==MOD_EQ ||  index==PLUS_EQ|| index==MINUS_EQ ||
index==LEFT_SHIFT_EQ || index==RIGHT_SHIFT_EQ ||
index==UNSIGNED_RIGHT_SHIFT_EQ || index==AND_EQ || index==EXOR_EQ ||
index==OR_EQ)
            return 1;
return 0;
 }

void inclusiveOrExpr()
 {
      exclusiveOrExpr();
      while(token==BIT_OR)
      {
      advance();
      exclusiveOrExpr();
      fprintf(outFile,"%c",IOR);
      }
 }

void exclusiveOrExpr()
 {
      andExpr();
      while(token==EXOR)
      {
      advance();
      andExpr();
```

```
      fprintf(outFile,"%c",IXOR);
      }
 }

void andExpr()
 {
      shiftExpr();
      while(token==BIT_AND)
      {
      advance();
      shiftExpr();
      fprintf(outFile,"%c",IAND);
      }
 }

void shiftExpr()
 {
      additiveExpr();
      while(token==LEFT_SHIFT||token==RIGHT_SHIFT)
      {
      int index;
      index=token;
      advance();
      additiveExpr();
      if(index==LEFT_SHIFT)
            fprintf(outFile,"%c",ISHL);
      else
```

```
            fprintf(outFile,"%c",ISHR);
      }
 }

void additiveExpr()
 {
      multiplicativeExpr();
      while(token==PLUS||token==MINUS)
      {
      int index;
      index=token;
      advance();
      multiplicativeExpr();
      if(index == PLUS)
            fprintf(outFile,"%c",IADD);
      else
            fprintf(outFile,"%c",ISUB);
      }
 }

void multiplicativeExpr()
 {
       unaryExpr();
       if(token ==STAR || token==DIV || token==MOD)
       {
       int index;
       index=token;
```

```
        advance();
        unaryExpr();
        switch(index)
        {
        case STAR  : fprintf(outFile,"%c", IMUL); break;
        case DIV   : fprintf(outFile,"%c", IDIV); break;
        case MOD   : fprintf(outFile,"%c", IREM); break;
        }
        }
}

void unaryExpr()
 {
      if(token==PLUS || token==MINUS)
      {
      int index=token;
      advance();
      unaryExpr();
      if(index==PLUS)
             ; /* do nothing  so no opcode */
      else if(index==MINUS)
            fprintf(outFile,"%c",INEG);
      }
      else if(token==PLUS_PLUS || token==MINUS_MINUS)
       {
            if(token==PLUS_PLUS)
                  preIncrementExpr();
```

```
            else if(token==MINUS_MINUS)
                  preDecrementExpr();
            advance();
        }
      else
             unaryExprNotPlusMinus();
 }

void preIncrementExpr()
 {
      int i;
      advance();
      i=lookUpSet(currIdentifier);
      if(i < 0)
            errorHandler(ERROR,"Identifier expected after ++");
      else
      {
            emitPrePostIncDecCode(i,1);
            emitLoadCode(i);
      }
}

void preDecrementExpr()
 {
      int i;
      advance();
      i=lookUpSet(currIdentifier);
```

```
      emitPrePostIncDecCode(i,-1);
      emitLoadCode(i);
 }

void unaryExprNotPlusMinus()
 {
      if(token==BIT_NOT)
      {
            advance();
            unaryExpr();
            fprintf(outFile,"%c",ICONST_M1);
            fprintf(outFile,"%c",IXOR);
      }
      else
            postfixExpr();
 }

void postfixExpr()
 {
      primary();
      name();
      postIncrementExpr();
      postDecrementExpr();
 }

void name()
{
```

```
 if(token==IDENTIFIER)
      {
            int i = lookUpSet(currIdentifier);
            emitLoadCode(i);
            advance();
      }
}

void primary()
{
      methodInvocation();
      if(token==B_PARENS)
      {
            advance();
            expr();
            verify(E_PARENS);
      }
      else if(token==INTEGER_CONST)
      {
            emitNumConstCode();
            advance();
      }
}

void postIncrementExpr()
{
      if(token==PLUS_PLUS)
```

```
	{
		int i;
		i=lookUpSet(currIdentifier);
		emitPrePostIncDecCode(i,1);
		advance();
		postfixExpr();
	}
}

void postDecrementExpr()
 {
	if(token==MINUS_MINUS)
	{
	int i;
	i=lookUpSet(currIdentifier);
	emitPrePostIncDecCode(i,-1);
	advance();
	postfixExpr();
	}
 }

static char *APIList[] = { "write",NULL};
/*	at present supports only one API namely "write". You can include
your  required list of APIs here and subsequently support in the
interpreter */

int lookupAPI(char *s)
```

```
 {
      int i=0;
      while(APIList[i]) /* simple linear search */
            if(!strcmp(APIList[i++],s))
                  return i-1; /* return table location - one */
      return -1;
 }

void argumentList()
 {
      expr();
      while(token == COMMA)
      {
            advance();
            argumentList();
      }
 }

void methodInvocation()
 {
      if(token==IDENTIFIER && nextToken == B_PARENS)
      {
            int index = lookupAPI(currIdentifier);
            if( index >=0 )
            {
            advance();           /* eat object name */
            verify(B_PARENS);    /* eat beginning parans */
```

```
            argumentList();      /* optional */
            fprintf(outFile,"%c",INVOKEMETHOD);
            fprintf(outFile,"%c",index);
            verify(E_PARENS);
            }
            else
                  errorHandler(ERROR,"Unknown function name");
      }
 }

void compile()
{
      outFile = fopen("a.out","wb+");
      if(outFile==NULL)
            errorHandler(ERROR,"Cannot open output file");
      initLex = true;
      advance();
      expr();
      fprintf(outFile,"%c",RETURN);
      fclose(outFile);
      printf("Successfully compiled");
}

int main()
 {
      char *prompt="expr :>";
      printf("\n This is the tinyExpr expression compiler. \
```

```
      \n Only arithmetic, bitwise and assignment operators are allowed\
      \n More than one expression can be separated by commas. \
      \n Make sure that the tokens are separated by white-spaces. \
      \n For example, a valid expression is : \
      \n var1 = 10 , var2 = 20 , var3 = var1 * var2 , write ( var3 ) \
      \n The compiled intermediate file is stored in file \"a.out\" \
      \n Type \"exec\" to execute the intermediate file (or)\
      \n Use \"interpre.exe\" to invoke it later from command prompt. \
      \n\n Type 'quit' to return back to command prompt");

      while(1) /* loop till 'quit' is opted */
      {
      printf("\n %s ",prompt);
      gets(sourceExpr);
      if(strcmp(sourceExpr,"quit")==0)
            break;
      else if (strcmp(sourceExpr,"exec")==0)
            system("interpre.exe");
      else
            compile();
      }
 }
/*************************end of compiler part **********************/
/********************code for intepreter starts here*****************/

/*****************************interpre.h*******************************/
```

```
/* the header files inclusion and the prototype declarations for the
compiler go here */

      /* these are the standard header files required */
      #include<stdio.h>
      #include<stdlib.h>

      /* declaration for error handling routine go here */
      void error(char *str);
      /* declarations for operations on operand stack */
      void push(int oper);
      int pop();
      /* declarations for mnemonic functions go here */
      void istoreCode();
      void istore(int i);
      void iadd();
      void isub();
      void imul();
      void idiv();
      void irem();
      void ineg();
      void ishl();
      void ishr();
      void iand();
      void ior();
      void ixor();
      void iinc();
```

```
void jreturn();
void invokemethod();

/* declarations for other functions */
void callFunction(int functionCode);
long fileSize(FILE *stream);
void atexitFree();

/************************error.c**********************/
/* this program has a tiny error handler routine */

void error(char *str)
      {
      fprintf(stderr,"\n ERROR   : %s\n",str);
      fprintf(stderr,"\n Exiting... Press any key");
      getchar();
      exit(0);
      }

/************************ostack.c**********************/
/* this program has stack and functions operating on it */

#define MAXSIZE 32

/* this is the operand stack where all execution takes place */
static int operand[MAXSIZE];
```

```
	/* this points to the top of the stack */
	static unsigned int oTop;

	void push(int oper)
	{
	if(oTop < MAXSIZE)
		operand[oTop++]=oper;
	else
		error("Runtime Error : Operand stack overflow");
	}

	int pop()
	{
	if(oTop > 0)
		return operand[--oTop];
	else
		error("Runtime Error : Operand stack underflow");
	}

	/************************interpre.c***********************/
	#include"interpre.h"
	/* function declarations & inclusion of standard header files */

	#include "mnemonic.h"
	/* this header file contains the bytecode values of various
mnemonics used */
```

```
	#include "error.c"
	/* this program has a tiny error handler routine */

	#include "ostack.c"
	/* this program has stack and functions operating on it */

	static unsigned char *byteCode;
	/* the actual bytecodes read from the file is pointed by this
variable */

	static int localVariable[MAXSIZE];
	/* this is the local variable table and in this data can be
stored
	and retrieved by the bytecode operating on it */

	static int PC;
	/* Program Counter keeps track of the current point of bytecode
execution */

	/* the following mnemonic functions operate on local variable
table to store the values of the corresponding variables */
	void istoreCode()
	{
	int value = pop();
	localVariable[PC++] = value;
	}
```

```
	void istore(int i)
	{
	int value = pop();
	localVariable[i] = value;
	}

	/* the following mnemonic functions operate on the operand stack
to do arithmetic operations on it by popping top two values and pushes
the result  back on the stack */
	void iadd()
	{
	int value2 = pop();
	int value1 = pop();
	push(value1+value2);
	}

	void isub()
	{
	int value2 = pop();
	int value1 = pop();
	push(value1-value2);
	}

	void imul()
	{
	int value2 = pop();
	int value1 = pop();
```

```
        push(value1*value2);
        }

        void idiv()
        {
        int value2 = pop();
        int value1 = pop();
        push(value1/value2);
        }

        void irem()
        {
        int value2=pop();
        int value1=pop();
        int val = (value1-(value2*(value1/value2)));
        push(val);
        }

        void ineg()
        {
        int value=pop();
        push(-value);
        }

        /* the following mnemonic functions operate on the operand stack
to do bit-wise operations on it in a similar way to arithmetic
operators */
```

```
void ishl()
{
int value2=pop();
int value1=pop();
push(value1<<value2);
}

void ishr()
{
int value2=pop();
int value1=pop();
push(value1>>value2);
}

void iand()
{
int value2 = pop();
int value1 = pop();
push(value1 & value2);
}

void ior()
{
int value2 = pop();
int value1 = pop();
push(value1 | value2);
}
```

```
	void ixor()
	{
	int value2 = pop();
	int value1 = pop();
	push(value1 ^ value2);
	}

	/* the following iinc mnemonic function directly operate on the
local variable table and increments or decrements its value */
	void iinc()
	{
	int index  = byteCode[PC++];
	signed char jconst = (signed char)byteCode[PC++];
	/* signed because it may be -ve or +ve */
	localVariable[index] += jconst;
	}

	/* return function does nothing. Had 'tinyExpr' supported
functions,  it would have been very useful */
	void jreturn()
	{
	}

	/* this mnemonic function gives support for API's. When
'tinyExpr' is extended to support functions, this will become very
useful */
```

```
void invokemethod()
{
if(byteCode[PC++] == 0)
     /* write API. print the integer argument passed */
       printf("%d",pop());
else   /* extend this to support more APIs */
       error("Illegal operand for invokemethod instruction");
}

/* this function works as the interpreter function to execute the
actions corresponding to the mnemonics or to identify and call the
corresponding mnemonic function */
void callFunction(int functionCode)
{
switch(functionCode)
{
case ICONST_M1 : push(-1); break;
case ICONST_0 : push(0); break;
case ICONST_1 : push(1); break;
case ICONST_2 : push(2); break;
case ICONST_3 : push(3); break;
case ICONST_4 : push(4); break;
case ICONST_5 : push(5); break;

case BIPUSH   : push(byteCode[PC++]);     break;
case SIPUSH   : push(*((int*)&byteCode[PC]));
                          PC += 2; break;
```

```
case ILOAD    : push(localVariable[PC++]); break;
case ILOAD_0  : push(localVariable[0]);    break;
case ILOAD_1  : push(localVariable[1]);    break;
case ILOAD_2  : push(localVariable[2]);    break;
case ILOAD_3  : push(localVariable[3]);    break;

case ISTORE   : istoreCode(); break;
case ISTORE_0 : istore(0); break;
case ISTORE_1 : istore(1); break;
case ISTORE_2 : istore(2); break;
case ISTORE_3 : istore(3); break;

case IADD :      iadd(); break;
case ISUB :      isub(); break;
case IMUL :      imul(); break;
case IDIV :      idiv(); break;
case IREM :      irem(); break;
case INEG :      ineg(); break;
case ISHL :      ishl(); break;
case ISHR :      ishr(); break;
case IAND :      iand(); break;
case IOR  :      ior(); break;
case IXOR :      ixor(); break;

case IINC :      iinc(); break;
```

```
case INVOKEMETHOD : invokemethod(); break;
case RETURN :       jreturn();break;
default : error("FATAL - Illegal Instruction encountered");
}
}

long fileSize(FILE *stream)
{
long curpos, length;

curpos = ftell(stream);
fseek(stream, 0L, SEEK_END);
length = ftell(stream);
fseek(stream, curpos, SEEK_SET);
return length;
}

void atexitFree()
{
free(byteCode);
/* before exiting, free the allocated memory */
}

int main()
{
long length;
int size;
```

```
/* open the bytecode file */
FILE *stream = fopen("a.out","rb");
if(stream==NULL)
      error("FATAL - Cannot open input file \"a.out\" ");

 /* read the bytecodes into memory from the bytecode file */
 length = fileSize(stream);
 byteCode = (char *)malloc(length);
 if(byteCode==NULL)
      error("FATAL - Cannot allocate enough memory");

 atexit(atexitFree);
 size = fread(byteCode,1,length,stream);
 if(size==0)
      error("FATAL - Cannot read bytecodes from the file
\"a.out\"");

 /* execute each and every mnemonic function */
 while( PC < length ) /* this is the main interpreter loop */
      callFunction(byteCode[PC++]);

 printf("\nProgram successfully executed (interpreted)");
 fclose(stream);
}
```

APPENDIX III - ANSI C Declarations

17.4 Standard C global variables

<errno.h>

```
extern volatile int errno;
```

17.5 Standard C type declarations

typedef	header file in which it is declared
clock_t	<time.h>
div_t	<stdlib.h>
FILE	<stdio.h>
fpos_t	<stdio.h>
jmp_buf	<setjmp.h>
ldiv_t	<stdlib.h>
ptrdiff_t	<stddef.h>
sig_atomic_t	<signal.h>
size_t	<stddef.h>
time_t	<time.h>
va_list	<stdarg.h>
wchar_t	<stddef.h>

17.6 Standard C structure and union declarations

17.6.1 <stdlib.h>

```
typedef struct {
long int quot;    /* quotient */
long int rem;     /* remainder */
} div_t;
typedef struct {
long int quot;    /* quotient */
long int rem;     /* remainder */
} ldiv_t;
```

17.6.2 <locale.h>

```
struct lconv {
char *decimal_point;
char *thousands_sep;
char *grouping;
char *int_curr_symbol;
char *currency_symbol;
char *mon_decimal_point;
char *mon_thousands_sep;
char *mon_grouping;
char *positive_sign;
char *negative_sign;
```

```
char int_frac_digits;
char frac_digits;
char p_cs_precedes;
char p_sep_by_space;
char n_cs_precedes;
char n_sep_by_space;
char p_sign_posn;
char n_sign_posn;
};
```

17.6.3 <time.h>

```
struct tm {
int tm_sec;    /* Seconds */
int tm_min;    /* Minutes */
int tm_hour;   /* Hour (0--23) */
int tm_mday;   /* Day of month (1--31) */
int tm_mon;    /* Month (0--11) */
int tm_year;   /* Year (calendar year minus 1900) */
int tm_wday;   /* Weekday (0--6; Sunday = 0) */
int tm_yday;   /* Day of year (0--365) */
int tm_isdst;  /* 0 if daylight savings time is not in
effect) */
};
```

APPENDIX IV - ANSI C IMPLEMENTATION-SPECIFIC STANDARDS

Certain aspects of the ANSI C standard are not defined exactly by ANSI. Instead, each implementor of a C compiler is free to define these aspects individually.

In C many details are left as implementation-dependent because, implementation may take the maximum advantage of the underlying hardware. A well-known example is the size of integer. The programmer can take advantage of the underlying hardware for maximum efficiency and to access the resources of underlying hardware. This is at the cost of portability sometimes, for example the case of bit-fields.

2.1.1.3 How to identify a diagnostic.

2.1.2.2.1 The semantics of the arguments to main.

2.1.2.3 What constitutes an interactive device.

2.2.1 The collation sequence of the execution character set.

2.2.1 Members of the source and execution character sets.

2.2.1.2 Multibyte characters.

2.2.2 The direction of printing.

2.2.4.2 The number of bits in a character in the execution character set.

3.1.2 The number of significant initial characters in identifiers.

3.1.2 Whether case distinctions are significant in external identifiers.

3.1.2.5 The representations and sets of values of the various types of floating-point numbers.

3.1.2.5 The representations and sets of values of the various types of integers.

3.1.3.4 The mapping between source and execution character sets.

3.1.3.4 The value of an integer character constant that contains a character/escape sequence not represented in the basic/extended execution character set for a wide character constant.

3.1.3.4 The current locale used to convert multibyte characters into corresponding wide characters for a wide character constant.

3.1.3.4 The value of an integer constant that contains more than one character, or a wide character constant that contains more than one multibyte character.

3.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented.

3.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value.

3.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number.

3.3 The results of bitwise operations on signed integers.

3.3.2.3 What happens when a member of a union object is accessed using a member of a different type.

3.3.3.4 The type of integer required to hold the maximum size of an array.

3.3.4 The result of casting a pointer to an integer or vice versa.

3.3.5 The sign of the remainder on integer division.

3.3.6 The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t.

3.3.7 The result of a right shift of a negative signed integral type.

3.5.1 The extent to which objects can actually be placed in registers by using the register storage-class specifier.

3.5.2.1 Whether a plain int bit-field is treated as a signed int or as an unsigned int bit field.

3.5.2.1 The order of allocation of bit fields within an int.

3.5.2.1 The padding and alignment of members of structures.

3.5.2.1 Whether a bit-field can straddle a storage-unit boundary.

3.5.2.2 The integer type chosen to represent the values of an enumeration type.

3.5.3 What constitutes an access to an object that has volatile-qualified type.

3.5.4 The maximum number of declarators that can modify an arithmetic, structure, or union type.

3.6.4.2 The maximum number of case values in a switch statement.

3.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value.

3.8.2 The method for locating includable source files.

3.8.2 The support for quoted names for includable source files.

3.8.2 The mapping of source file name character sequences.

3.8.8 The definitions for __DATE__ and __TIME__ when they are unavailable.

4.1.1 The decimal point character.

4.1.5 The type of the sizeof operator, size_t.

4.1.5 The null pointer constant to which the macro NULL expands.

4.2 The diagnostic printed by and the termination behavior of the assert function.

4.3 The implementation-defined aspects of character testing and case mapping functions.

4.3.1 The sets of characters tested for by isalnum, isalpha, iscntrl, islower, isprint and isupper functions.

4.5.1 The values returned by the mathematics functions on domain errors.

4.5.1 Whether the mathematics functions set the integer expression errno to the value of the macro ERANGE on underflow range errors.

4.5.6.4 Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero.

4.7.1.1 The set of signals for the signal function.

4.7.1.1 The semantics for each signal recognized by the signal function.

4.7.1.1 The default handling and the handling at program startup for each signal recognized by the signal function.

4.7.1.1 If the equivalent of signal(sig, SIG_DFL); is not executed prior to the call of a signal handler, the blocking of the signal that is performed.

4.7.1.1 Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function.

4.9.2 Whether the last line of a text stream requires a terminating newline character.

4.9.2 Whether space characters that are written out to a text stream immediately before a newline character appear when read in.

4.9.2 The number of null characters that may be appended to data written to a binary stream.

4.9.3 Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file.

4.9.3 Whether a write on a text stream causes the associated file to be truncated beyond that point.

4.9.3 The characteristics of file buffering.

4.9.3 Whether a zero-length file actually exists.

4.9.3 Whether the same file can be open multiple times.

4.9.4.1 The effect of the remove function on an open file.

4.9.4.2 The effect if a file with the new name exists prior to a call to rename.

4.9.6.1 The output for %p conversion in fprintf.

4.9.6.2 The input for %p conversion in fscanf.

4.9.6.2 The interpretation of an - (hyphen) character that is neither the first nor the last character in the scanlist for a %[conversion in fscanf.

4.9.9.1 The value to which the macro errno is set by the fgetpos or ftell function on failure.

4.9.10.4 The messages generated by perror.

4.10.3 The behavior of calloc, malloc, or realloc if the size requested is zero.

4.10.4.1 The behavior of the abort function with regard to open and temporary files.

4.10.4.3 The status returned by exit if the value of the argument is] other than zero, EXIT_SUCCESS, or EXIT_FAILURE.

4.10.4.4 The set of environment names and the method for altering the environment list used by getenv.

4.10.4.5 The contents and mode of execution of the string by the system function.

4.11.6.2 The contents of the error message strings returned by strerror.

4.12.1 The local time zone and Daylight Saving Time.

4.12.2.1 The era for clock. The formats for date and time.

Note: The section numbers refer to the ANSI Standard as on February 1990.

.

Suggested Readings

[Kernighan and Ritchie 1978], [Kernighan and Ritchie 1988]

This is a ‘must to read’ for anyone serious about C programming. The first edition served as the only primary source of C reference for nearly a decade. Many intricate parts of the language can be understood from this book because this book is from the author of the language itself. Its appendix has a compact reference manual for C.

[Harbison and Steele 1987]

Now this C reference manual is in its fourth edition. This is an excellent reference next to `[Kernighan and Ritchie 1988]` and it has full coverage of basic building blocks of C like operators and expressions. Its second part explores the C standard library fully.

[Koenig 1989]

Andrew Koenig is a skilled author who really knows how to effectively communicate ideas. This is a small book, worth reading because C is the language where programmers can easily make mistakes. The book explores the intricate parts of the language to make you aware of the traps and to avoid pitfalls that are in C.

[Allison 1998]

This is a very good book for knowing various issues associated with the language features with innumerable examples. It has a good coverage of practical usage of various features and has a free-flowing language for explanation.

References

[Aho and Ullman 1977]

Alfred V. Aho and Jeffrey D. Ullman, “Principles of Compiler Design”, Addison-Wesley Publishing Company, 1977

[Allison 1998]

Chuck Allison, “C/C++ Code Capsules” – A guide to Practitioners, Prentice Hall of India Pvt. Ltd., 1998

[ANSI C 1989]

American National Standards Institute. ANSI X3J11 Committee. ”Programming Language C”, Document X3.159-198x, 14 Nov 1988.

[ANSI C 1998]

American National Standards Institute. ”Programming Language C”, Committee Draft - WG14/N843, 3 August 1998.

[Binsock 1987]

Andrew Binsock, “the simple ‘Clockwise’ rule helps decipher complex declarations”, The C user’s group newsletter, June 1987

[Bohm and Jacobini 1966]

Bohm C. and Jacobini G., Flow Diagams, “Turing Machines and languages with only two formation rules”, Communications to ACM, 1966

[Dijkstra 1968]

Edsgar Dijstra, “GOTO Statement Considered Harmful”, CACM 11:3, March 1968

[Duff 1984]

Tom Duff, “netnews”, May 1984

[Gosling and Joy 1995]

James Gosling and Bill Joy, “The Java Programming Language”, Addison-Wesley, 1995

[Harbison and Steele 1987]

Samuel P. Harbison and Guy L. Steele, Jr. “C : A Reference Manual”, Second edition, Prentice-Hall, Inc., Englewood Cliffs, 1987.

[Henricson and Nyquist 1997]

Industrial Strength C++ - Rules and Recommendations, Mats Henricson, Eric Nyquist, Prentice Hall of India Pvt. Ltd., 1997

[Hejlsberg and Wilamuth 2001]

Anders Hejlsberg and Scott Wiltamuth, “C# Language Reference”, Microsoft Corporation, 2001

[Holub 1990]

Allen I Holub, “Compiler design in C”, Second edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.

[Johnson and Ritchie1981]

Johnson S.C. and Ritchie D.M., “The C language calling sequence”, Computing Science Technical Report No. 102, AT&T Bell Laboratories, Murray Hill, N.J, 1981.

[Kernighan and Ritchie 1978]

Brian W. Kernighan and Dennis M. Ritchie, “The C Programming Language”, First edition, Prentice-Hall, Inc., Englewood Cliffs, 1978

[Kernighan and Ritchie 1988]

Brian W. Kernighan and Dennis M. Ritchie, “The C Programming Language”,Second edition, Prentice-Hall, Inc., Englewood Cliffs, 1988

[Koenig 1989]

Andrew Koenig, “C Traps and Pitfalls”, Addison-Wesley, 1989

[Kruglinski 1995]

David J. Kruglinski, “Inside VC++”, 3 ed, Microsoft Press, 1995

[Lindholm and Yellin 1990]

Tim Lindolm, Frank Yellin, “Java Virtual machine specification”, Addison-Wesley, 1995

[Martin et al. 1991]

Martin, James and James Odell, “Object-oriented Modeling and Design”, Englewood Cliffs, Prentice Hall Inc., NJ, 1991

[Rationale ANSI C 1999]

Rationale for International Standard-Programming Languages – C, Revision 2, American National Standards Institute, 20 October 1999.

[Ritchie 1978]

Ritchie D.M , "UNIX time sharing system: A retrospective" , Bell Systems Technical Journal, 1978.

[Ritchie1982]

Dennis M. Ritchie, “Operator precedence”, net.lang.c, 1982

[Ritchie et al.]

D.M.Ritchie, S.C.Johnson, M.E.Lesk and B.W.Kernighan, “UNIX Time-sharing System: The C programming language”

[Rumbaugh et al. 1991]

James Rumbaugh, Michael Blaha, William Perarlani, Frederick Eddy and William Lorensen, “Object-Oriented Modeling and Design”, Prentice Hall,Inc., Englewood Cliffs, NJ, 1991

[Stroustrup 1986]

Bjarne Stroustrup, “The C++ programming language”, Addison-Wesley, Reading, MA, 1986

[Stroustrup 1994]

Bjarne Stroustrup, “The Design and Evolution of C++”, Addison-Wesley, 1994.

INDEX

A great part of information I have was acquired by

looking up something,

and finding something else on the way.

- Franklin P. Adams