Industrial C

Smart & Efficient Guidelines to plug you into the IT & Electronics Industry

Gear-up with necessities of

Programming

Fundamentals

What's the difference between these candidates?

One took

90% Industrial Skills

with proper academic qualification,
other one took excellence in academic only

"Don't insult yourself in front of you; bring out your inborn courage to be competitive in the world. Just live your life with your own passion."

Streamline your next priority with us
How much time & money could you save with 'Team Rapidcode's proven solutions?

If you like to optimize your projects efficiently & industrial skills without sacrificing availability or driving up costs, its a question you should be asking yourself, whether you need:

- Comprehensive & Industrial learning
- Real-Time development of project
- Handling SDLC, GCC, GIT, RT-Linux etc. industrial tools
- Getting 2+ years experience feel even being fresher

Discover our new improved Industrial training program

- **SI2.00x**
- **SI2.01x**
- **SI2.02x**

Explore today & transmute future!

Content

Preface

"At least for the people who send me mail about a new language that they're designing, the general advice is: do it to learn about how to write a compiler". ~ Dennis Ritchie

Whenever we're talking seriously about engineering, we get little confused,

Is the theoretical book which we are studying actually what we call "engineering"?

One of the Professors from MIT says –"**purposeful use of science is engineering**". When it came to my mind, I started looking back into my graduation days, full of academics, full of dreams & innovations. My Lecturer used to say when an UG candidate joins an engineering college, he is full of confidence, enthusiasm, and innovation but as his knowledge increases in the theoretical engineering, he starts feeling bored and in turn the pace of innovative thinking decreases in his mind. It is true in most of the cases.

So the lecturer was not wrong! I tried to find out the root cause of decreasing zeal in the engineering students towards their skills & hope.
I concluded that our course curriculum was full of theories & solved exercises on old pattern. It was my common survey on the engineering students of middle level institutions where 85% of whole is studying. Then I came up with the idea to have a professional corporate training to encourage the industrial feel & experience among the candidates.

Programmers have a lot on their minds. Programming languages, programming techniques, development environments, coding style, tools, development processes, deadlines, meetings, software architecture, design patterns, team dynamics, code, requirements, bugs, code quality and a lot more.There is an art, craft, and science to programming that extends far beyond the program. Considering all above industrial activity of a professional mind, I designed -

1. SI2.00x : 1. SI2.00x : For System Software Design Group
2. SI2.01x : For Embedded Software Design Group
3. SI2.02x : For Electronics & Embedded Design Group

The most important part of this training is C. C is perhaps the most popular programming language created in 1972 by Dennis Ritchie at the Bell Labs in USA as a part of UNIX operating system. C is used to develop core parts of this operating system. From that time C programming language has been the de facto programming language when fast programs are needed or the software needs to interact with the hardware in some way.

Most of the operating systems like Linux, Windows™, and Mac™ are either developed in C language or use this language for most parts of the operating system and the tools coming with it. When it comes to introduction & origin we'll get to know – "**Why is C most important basic tool of engineering in digital world."**

Acknowledgement

How 'quick wins' in IT/ Management can boost morale nevertheless we are acting!

Many people have contributed their time and their insight, both directly and indirectly, to the SourceIn & Rapidcode Technologies as project.

They all deserve credit, first inspiration to Great Mr. Dennis M. Ritchie who is the godfather of C.
All the mentors, institution & open source community who helped me to learn the applied concept in depth up to my current state deserve the true credit.

I would like to thank all those who devoted the time and effort to contribute items to this project, proof readings, professional ideas & moral support.
Special thanks to my students for typing & proof-reading support
Miss Mani Sharma
Miss Pooja Chauhan
Miss Archna Malik
Mr. Prabhat Sharma
Mr. Sunny Juneja

I believe in the fact that company i.e. 'association with' can never be an actual picture without mates, we're team, we're transmuting future!

~ Ritesham Shastri

1

PREREQUSITE

Linux Install, VI Editor+ shortcut knowledge, Man pages
Typing, Learning & Coding Passion

Getting Started with Linux

Part 1 of 3: Preparing the Installation DVD

Backup your data.
Download the Linux Distributions (e.g. Mint, Fedora, RH etc) ISO.
Download an image burning program.
Burn the disc.

Part 2 of 3: Installing Linux Distributions

Set your computer to boot from your DVD drive.
Start the Linux Distributions live system.
Start the installation.
Review the basic installer requirements.
Choose your installation type.
Specify the hard drive you want to install to.
Set your location and keyboard settings.
Create your user.
Wait for the installation is complete.
Click "Restart Now".

Part 3 of 3: Configuring Linux Distributions

Boot into Linux Distributions.
Review the Welcome screen.
Configure your desktop.
Install more programs.
Start your new programs.
Change your desktop background.
Open the Terminal.
Keep exploring and tinkering.

Warnings

When installing, you can wipe all or just part of your hard drive. If you're careful and use some common sense, you won't get anything wrong. Just bear in mind that if you're not careful, you can potentially lose data.

[Note: Same process you can use in Virtual Machine like VMWare, VirtualBox etc.]

Installation on Mac OS

If you use Mac OS X, the easiest way to obtain GCC is to download the Xcode development environment from Apple's web site and follow the simple installation instructions. Once you have Xcode setup, you will be able to use GNU compiler for C/C++.

Xcode is currently available at developer.apple.com/technologies/tools/.

Installation on Windows

To install GCC at Windows you need to install MinGW. To install MinGW, go to the MinGW homepage, www.mingw.org, and follow the link to the MinGW download page. Download the latest version of the MinGW installation program, which should be named MinGW-<version>.exe.

While installing MinWG, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.

Add the bin subdirectory of your MinGW installation to your PATH environment variable, so that you can specify these tools on the command line by their simple names.

When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.

VI Editor+ shortcut knowledge

What is vi?
The default editor that comes with the UNIX operating system is called vi (visual editor). [Alternate editors for UNIX environments include pico and emacs, a product of GNU.]

The UNIX vi editor is a full screen editor and has two modes of operation:
1. Command mode commands which cause action to be taken on the file, and
2. Insert mode in which entered text is inserted into the file.

In the command mode, every character typed is a command that does something to the text file being edited; a character typed in the command mode may even cause the vi editor to enter the insert mode. In the insert mode, every character typed is added to the text in the file; pressing the <Esc> (Escape) key turns off the Insert mode.

VI Keyboard Shortcuts

Insert	
i	Inserts text to the left of the cursor.
I	Inserts text at the beginning of the line, no matter where the cursor is positioned on the current line.

Append	
a	Begins inserting after the character (append) on which the cursor is positioned.
A	Begins inserting at the end of the current line, no matter where the cursor is positioned on that line.

Open	
o	Begins inserting text on a new, empty line that is opened for you, below the current line. This is the only command that will allow you to insert text BELOW the LAST line of the file.
O	Begins inserting text on a new, empty line that is opened for you, above the current line. This is the only command that will allow you to insert text ABOVE the FIRST line of the file.

Deleting, copying and changing	
d	Delete text. (see explanation above)
y	Copy text (that is, yank it into a holding area for later use). (see explanation above)
c	Change text from one thing to another, which you will type. (see explanation above)
!	Filter text through a program.
<	Shift a region of text to the left.
>	Shift a region of text to the right.

Single Key Movements	
h	Move cursor to the left one character.
l	Move cursor to the right one character.
j	Move cursor down one line.
k	Move cursor up one line.
^	Move cursor to the beginning of the line.
$	Move cursor to the end of the current line.
1G	Move cursor to the first line of your document. Other numbers will move to the line specified by number (ex. 50G goes to the 50th line).
G	Move cursor to the last line of your file.
CTRL U	Move cursor up in file 12 lines. Hold down the key marked CTRL (stands for control) and type U. CTRL is like another shift key.
CTRL D	Move cursor down in file 15 lines.
w	Move cursor forward to the next word, stopping at punctuation.
W	Move cursor forward to the next word, ignoring punctuation.
e	Move cursor forward to the end of the word, stopping at punctuation.
E	Move cursor forward to the end of the word, ignores punctuation.
b	Move cursor backwards to the previous word, stopping at punctuation.
B	Move cursor backwards to the previous word, ignores punctuation.
H	Move cursor to the top line of the screen, (as opposed to the top of the document which may not be the same place).
M	Move cursor to the middle of the screen.
L	Move cursor to the last line on the screen.
%	Move cursor to the matching parenthesis, bracket or brace. Great for debugging programs.
(	Move cursor to the beginning of the previous sentence (where a punctuation mark and two spaces define a sentence).
)	Move cursor to the beginning of the next sentence.
{	Move cursor to the beginning of the current paragraph.
}	Move cursor to the beginning of the next paragraph.
;	Repeat the last f or F command (see below).

Almost Single Key Movements	
'	Move cursor to a previously marked location in the file. (ex. ma marks the location with the letter a, so a (apostrophe a) moves back to that location).
f	Find the character corresponding to the next keystroke typed. Move the cursor to the next

	occurrence of that character (on the current line only).
F	Same as f but movement is backwards.

Useful	
x	Delete character(s) to the right of the cursor, starting with the one beneath it.
r	Replace the character under the cursor with the next character you type. This can be a very useful command. If you wanted to split up a line between two words, you might put the cursor on the blank space before the word you would like to go on the next line and type r . This would replace the space between the words with a carriage return and put the rest of the line onto a new line.
J	Join lines; the opposite of the line splitting operation above. This will join the current line with the next line in your file. Also very useful.
R	Replace lines; puts you in INSERT mode but types over the characters that are already on the current line.
p	Paste line(s) you deleted (or yanked) back into the file. This is an excellent command if you want to move a few lines somewhere else in your file. Just type 3dd to delete three lines, for example, and then move to where you want those lines to be and type p to paste the lines back into your file below the cursor.
.	The period . command repeats the last text modification command, whatever it may have been (insert, deletion, etc).
:r filename RETURN	Read a file into the current file being edited. The file be added gets placed below the current cursor position. Please note the colon : before the r in this command.
CTRL L	Redraw the screen. If somebody writes to you while you are in the middle of vi and junk appears all over your screen, dont panic, it did not hurt your file, but you will have to hold down the CTRL key and type L to clean it up (CTRL L).
d$	Delete (including the current character), to the end of the line.
d^	Delete (excluding the current character), to the beginning of the line.
dw	Delete a word(s), stops at punctuation.
dW	Delete a word(s), ignoring punctuation.
de	Delete to the end of next word.
dd	Delete a line(s).
dG	Delete from the current line to the end of the document. CAREFUL: Slightly dangerous.
dH	Delete from the current line to the line shown at the top of the screen.

Search and Replace	
/the	Finds the next occurence of the. This will also find their, them, another, etc.
?the	Finds the previous occurence of the.
N	previous
n	Repeats the last search command. Finds the Next occurence.

d/the	Deletes until the next occurence of the. This is to demonstrate how the delete prefix can be used with any cursor movement command.
:g/oldword/s//newword/gc	This will find all occurences of oldword and replace them with newword. The optional c at the end of the command tells vi that you would like to confirm each change. Vi will want you to type in y to make the change or n to skip that replacement. Great for spelling fixes.
Exit & Clipboard	
ESC :wq RETURN	Save and exit VI
ESC :q! RETURN	Exit WITHOUT saving changes
:e <filename>	open <filename>
dw	delete word
dd	delete line
P	put before cursor
p	put after cursor
yy	copy line
yw	copy word

Using MAN Pages

"man pages" is short for "manual pages", and is one of the oldest forms of help documentation.

Most of the time, you'll see man pages stored as compressed text files in the /usr/share/man directory. Some pages may also be stored in /usr/local/share/man.

```
ritesh@localhost man]$ pwd
/usr/share/man
[ritesh@localhost man]$ ls
bg de en fi hu ja man0p man2 man4 man7 mann pt ru sv
cs de.UTF-8 en_GB fr id jp man1 man3 man5 man8 nl pt_BR sk
da el es hr it ko man1p man3p man6 man9 pl ro sl
[ritesh@localhost man]$ cd man1
[ritesh@localhost man1]$ ls
:.1.gz jw.1.gz pnmsplit.1.gz
[.1.gz jwhois.1.gz pnmstitch.1.gz

----------------
access.1.gz knock.1.gz pnmtopng.1.gz
aconnect.1.gz knockd.1.gz pnmtopnm.1.gz
acyclic.1.gz kpsepath.1.gz pnmtops.1.gz
adcfw-log.1.gz kpsestat.1.gz pnmtorast.1.gz
addftinfo.1.gz kpsetool.1.gz pnmtorle.1.gz
. . .
```

Man Page Sections

There are nine categories, or sections, of man pages in common use. (You may also, at times, see other more specialized sections.)

Section Number	Description
1	Executable user programs and shell commands.
2	Kernel functions or system calls.
3	Library calls that are provided by program libraries.
4	Information on device files. (Mostly, this will cover files found in the /dev directory.
5	Descriptions of file formats. (This can include Message of the Day files, configuration files, keymap files, etc.
6	Games. (Of course, most newer games come with built-in help as part of the graphical interface.)
7	Miscellaneous topics. (Macro packages, conventions, regex, etc.)
8	System administration utilities. (Most of these will require root privileges. Examples: fsck, fdisk, mount, renice, rpm, dpkg.)
9	Linux kernel documentation.

Typing, Learning & Coding Passion

Don't just learn the language, understand its culture!

It comes out when you really love your coding, try to generate the feeling that the cod you are going to develop will definitely be one of your achievement in terms of knowledge, project, profession or contribution to society.

First step to learning prerequisite is Typing. As you are well enough in typing, you can actually feel your code as early as possible, only then you can give proper time to make it generic/ optimized.

Experimenting with your programming, trying to modify the open-source available codes, entering into core of Kernel are some real time passion of any C Programmer. Once you've learned the ropes of a new language, you'll be surprised how you'll start using languages you already know in new ways.

Explore the potential of language play with it, battle with it, be constructive! Be a good analyzer, have sharp edge on the bug fixing.

Say to yourself with determination – **"I am a C developer because it's my passion to code"**

2

INTRODUCTION

How much do you know about origin of programming?
Hand on Practice to Digital Arithmetic

How much do you know about origin of programming?

There have been discovery of zero i.e. *'shoonya'* by the great scientist *'Aryabhatta'* & several other great mathematician developed the theorems, algorithms & proofs of several unsolved complex problems which results into development of qualified mechanics & quantum research. Some of scientist discovers the dual nature of particle & periodic table. In this way we confirmed about the special quality of the semiconductors especially the hidden power of Silicon.

Does it really the origin of programming?

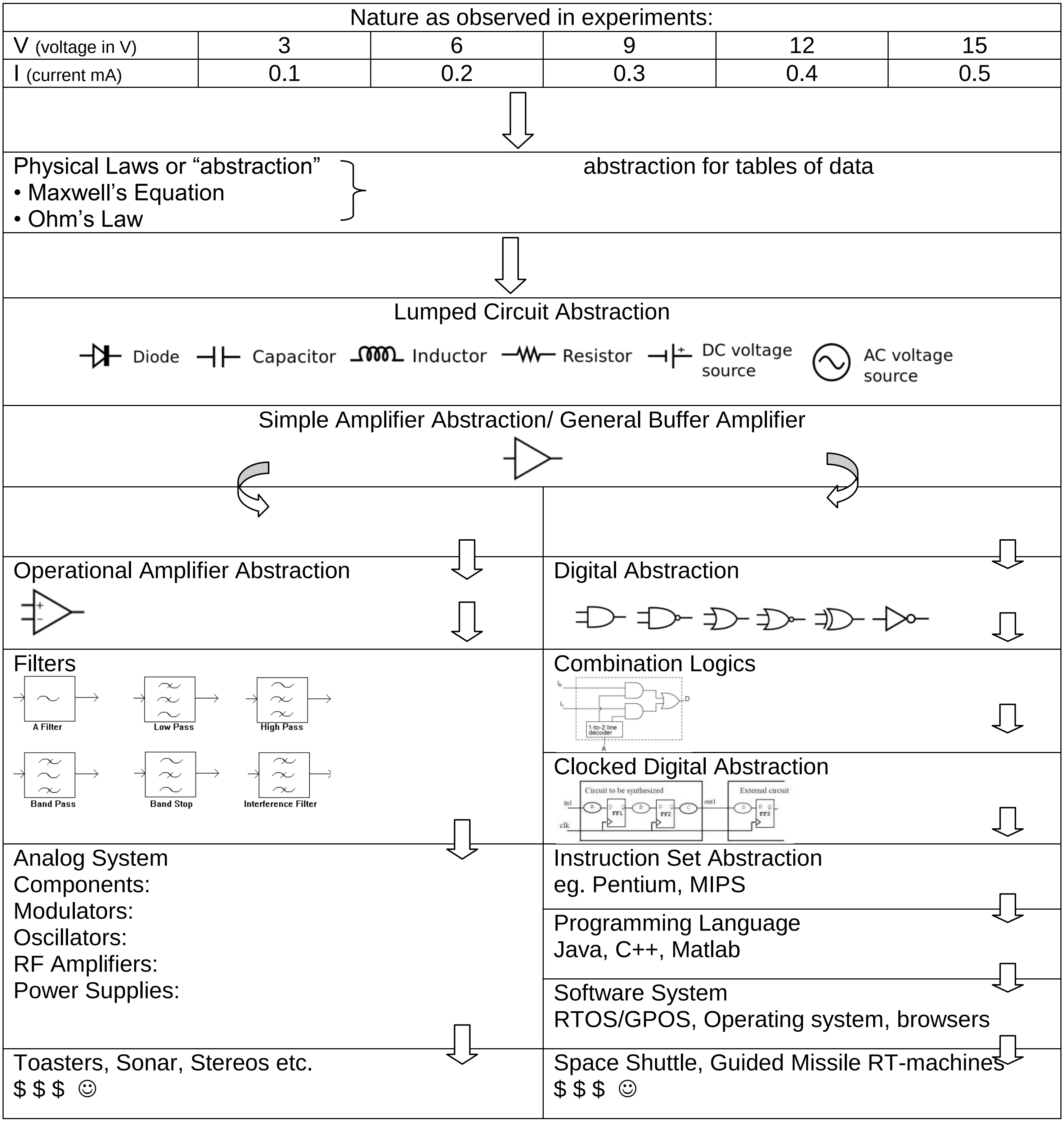
Hold on! We're coming to the main topic, in fact it can be the grand origin of any programming language. After the discovery of zero & charge, we became familiar with the Boolean concept 0 & 1, which is a milestone to programming. In all the cases we've to achieve the desired sequence of 0&1 but Engineers/Researchers named it Assembly level coding.

So one more question arise here,

Do the developers should know the quantum mechanics/ electronics while coding?

The answer is –"**Mostly not required**" to justify this statement we've the abstraction layers of engineering given by MIT professors.

Three mistakes of my life R resistance L inductance C capacitance



Nature as observed in experiments:

V (voltage in V)	3	6	9	12	15
I (current mA)	0.1	0.2	0.3	0.4	0.5

Physical Laws or “abstraction”
- Maxwell’s Equation
- Ohm’s Law

abstraction for tables of data

Lumped Circuit Abstraction

Diode Capacitor Inductor Resistor DC voltage source AC voltage source

Simple Amplifier Abstraction/ General Buffer Amplifier

Operational Amplifier Abstraction

Digital Abstraction

Filters

Combination Logics

Clocked Digital Abstraction

Analog System
Components:
Modulators:
Oscillators:
RF Amplifiers:
Power Supplies:

Instruction Set Abstraction
eg. Pentium, MIPS

Programming Language
Java, C++, Matlab

Software System
RTOS/GPOS, Operating system, browsers

Toasters, Sonar, Stereos etc.
$ $ $ ☺

Space Shuttle, Guided Missile RT-machines
$ $ $ ☺

Fig 2: Abstraction Layers (Source: MIT 6.002 OpenCourseWare)

Hence, we can conclude how the entire scenario developed, have look on the following progress-

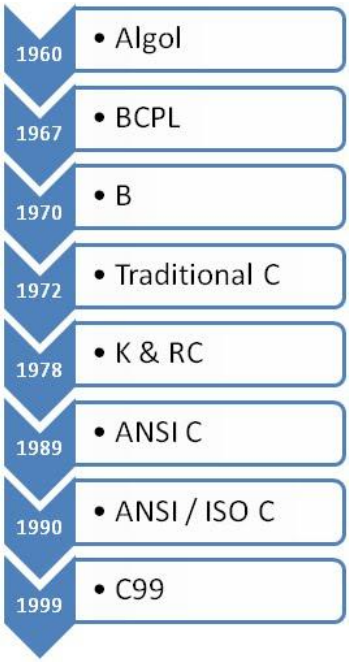
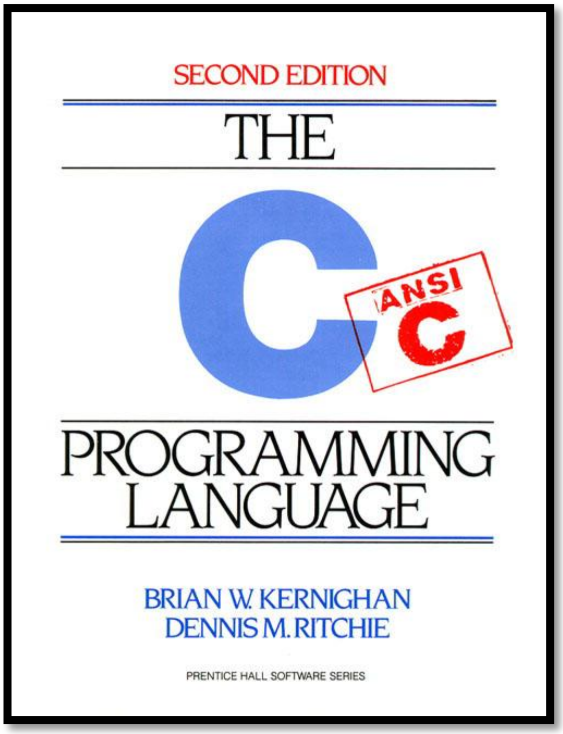


Fig 3: C Language Development (1960-1999)

We recommend “The C Programming Language” Book by
Brian Kernighan and Dennis Ritchie



Hand on Practice to Digital Arithmetic

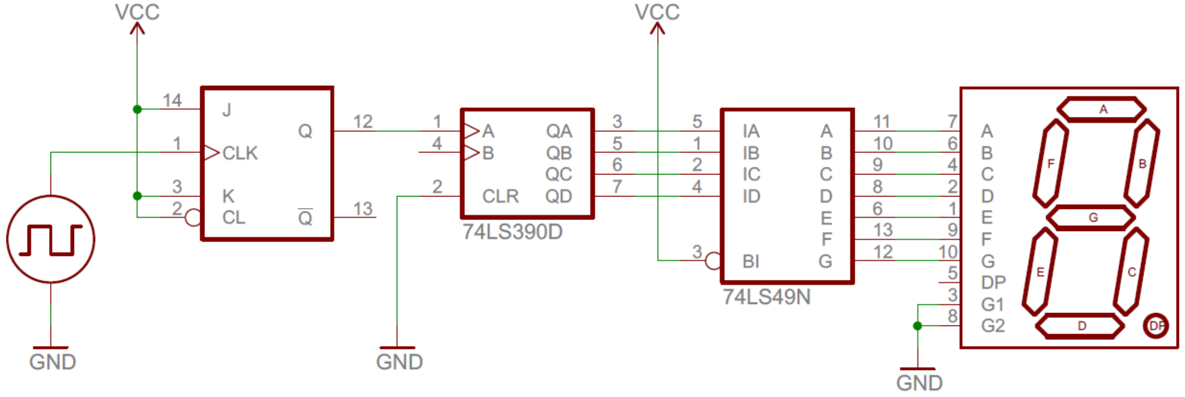


Fig 4: Seven Segement Display using 74LS49N & 74LS390D using simple digital logics

Don’t worry of the circuit; we’ll learn things step by step.

The unit which we’re familiar with ‘bit/ Byte’
It is for measuring memory in the digital world explains as
1 Byte = 8 bit

Let us understand the Digital system:

Digital systems are used in communication, business transactions, traffic control, spacecraft guidance, medical treatment, weather monitoring, the Internet, and many other commercial, industrial, and scientific enterprises.

We have digital telephones, digital televisions, digital versatile discs, digital cameras, handheld devices, and, of course, digital computers. We enjoy music downloaded to our portable media player (e.g., iPod Touch™) and other handheld devices having high resolution displays.

These devices have graphical user interfaces (GUIs), which enable them to execute commands that appear to the user to be simple, but which, in fact, involve precise execution of a sequence of complex internal instructions. Most, if not all, of these devices have a special-purpose digital computer embedded within them.

The most striking property of the digital computer is its generality. It can follow a sequence of instructions, called a program that operates on given data.

Now think of the 0 &1 in following circuits

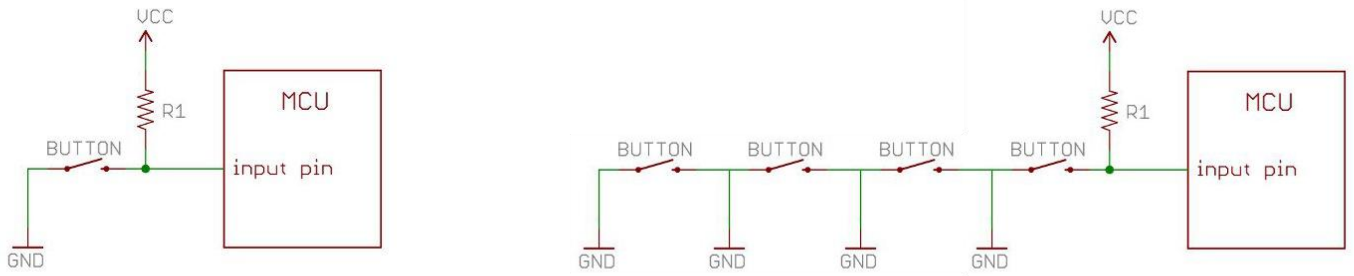


Fig 5: General Understanding of digital no representation & their circuit simulation

Do it yourself

Can we store Data 2 , 5, 128, 129, 255, 256, 135 in 1 byte of memory ? Also justify all storing with the below 8 bit shift register.

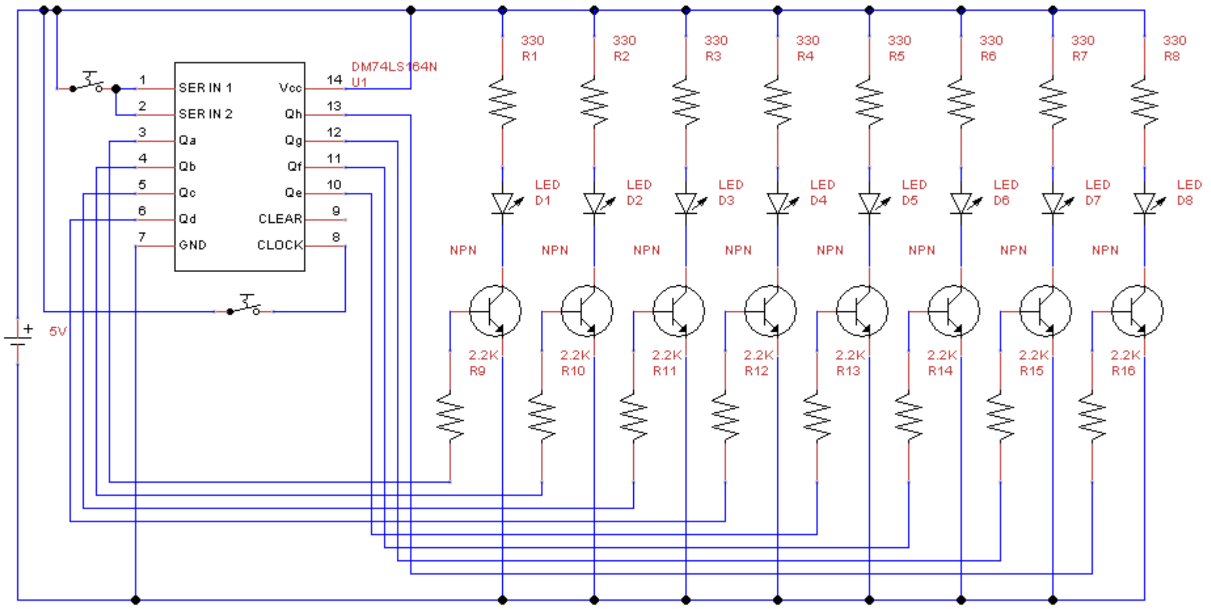


Fig 6: General Understanding of digital storing using 8 bit shift register

Number System

There are many different number systems. In general, a number expressed in a base-*r* system has coefficients multiplied by powers of *r*:

$$a_n \bullet r^{n} + a_{n-1} \bullet r^{n-1} + \;----------\; + a_2 \bullet r^{2} + a_1 \bullet r + a^{0} + a_{-1} \bullet r^{-1} + a_{-2} \bullet r^{-2} + \;----------\; + a_{-m} \bullet r^{-m}$$

The coefficients *aj* range in value from 0 to *r* - 1. To distinguish between numbers of different bases, we enclose the coefficients in parentheses and write a subscript equal to the base used (except sometimes for decimal numbers, where the content makes it obvious that the base is decimal).

Know about ‘radix point’

Have exercise on-

Binary to decimal conversions

Octal to decimal conversions

Hexadecimal to decimal conversions

Decimal to Binary conversions

Octal to Binary conversions

Hexadecimal to Binary conversions

Find out 1’s Complement, 2’s complement, 10’s complement & 9’s complement of given number. & subtraction methodology with two cases, when subtracting higher no to lower no & wise versa.

Logic Gates

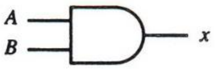
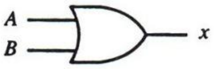
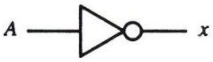
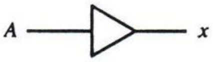
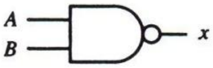
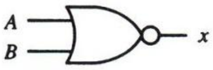
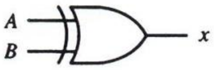
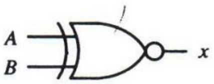
Name	Graphic symbol	Algebraic function	Truth table	
AND		$x = A \cdot B$ or $x = AB$	A B | x 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1	IC 7408
OR		$x = A + B$	A B | x 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1	IC7432
Inverter		$x = A'$	A | x 0 | 1 1 | 0	IC7404
Buffer		$x = A$	A | x 0 | 0 1 | 1	IC7407
NAND		$x = (AB)'$	A B | x 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0	IC7400
NOR		$x = (A + B)'$	A B | x 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0	IC7402
Exclusive-OR (XOR)		$x = A \oplus B$ or $x = A'B + AB'$	A B | x 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0	IC7486
Exclusive-NOR or equivalence		$x = (A \oplus B)'$ or $x = A'B' + AB$	A B | x 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1	

Fig 7– Symbols, algebric function & Truth Table

“an unambiguous & machine- independent definition of the language C,” while still retaining its spirit. The result is the ANSI standard for C~

Brian W. Kernighan
Dennis M. Ritchie

3

C Programming Language

C: Brief Theory

In general sense, *Programming is writing a secret code for making life simpler ~Anonymous*

C is a general-purpose programming language with features economy of expression, modern flow control and data structures, and a rich set of operators. C is not a ‘‘very high level’’ language, nor a ‘‘big’’ one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

In our experience, C has proven to be a pleasant, expressive and versatile language for a wide variety of programs. It is easy to learn, and it wears well as one’s experience with it grows.

C is a general-purpose programming language. It has been closely associated with the UNIX operating system where it was developed, since both the system and most of the programs that run on it are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a ‘‘system programming language’’ because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains.

In 1983, the American National Standards Institute (ANSI) established a committee whose goal was to produce ‘‘an unambiguous and machine-independent definition of the language C’’, while still retaining its spirit. The result is the ANSI standard for C.

It is better to know at least **one Programming Language & One Operating System in Depth.**

Scope of Advance/Industrial C:

- C language is used for creating computer applications
- Used in writing Embedded Firmware/ Software
- Firmware for various electronics, industrial & communications products which use micro-controller
- It is also used in developing verification software, test code, simulators etc for various applications & hardware products.
- For creating complier of different languages which can take input from other language and convert it into lower level machine dependent language
- C is used to implement various Operating Systems & their applications.
- UNIX kernel is completely dependent of C Language

▪ Operating System
▪ Network Drivers
▪ Client-Server Implementation
▪ Print Spoolers
▪ Language Compliers
▪ Assemblers
▪ Text Editors
▪ Modern Program
▪ Database
▪ Intermediate Language –for portability on machine dependent targets
▪ directly or indirectly influenced- C#, D, Go, Java, JavaScript, Limbo, LPC, Perl, PHP, Python, and Unix's C Shell
▪ GNU Multi-Precision Library, the GNU Scientific Library, Mathematica and MATLAB are completely or partially written in C
▪ Language Interpreters
▪ Simulators
▪ Utilities
▪ Embedded System
▪ Aviation
▪ Military
▪ Telecom
▪ Automate
▪ Healthcare
▪ ERP
▪ Banking & Finance

Short Note on SDLC & Version Control Tool

A *version control system* (also known as a *Revision Control System*) is a repository of files, often the files for the source code of computer programs, with monitored access. Every change made to the source is tracked, along with who made the change, why they made it, and references to problems fixed, or enhancements introduced, by the change.

Version control systems are essential for any form of distributed, collaborative development. Whether it is the history of a wiki page or large software development project, the ability to track each change as it was made, and to reverse changes when necessary can make all the difference between a well managed and controlled process and an uncontrolled ‘first come, first served’ system. It can also serve as a mechanism for due diligence for software projects.

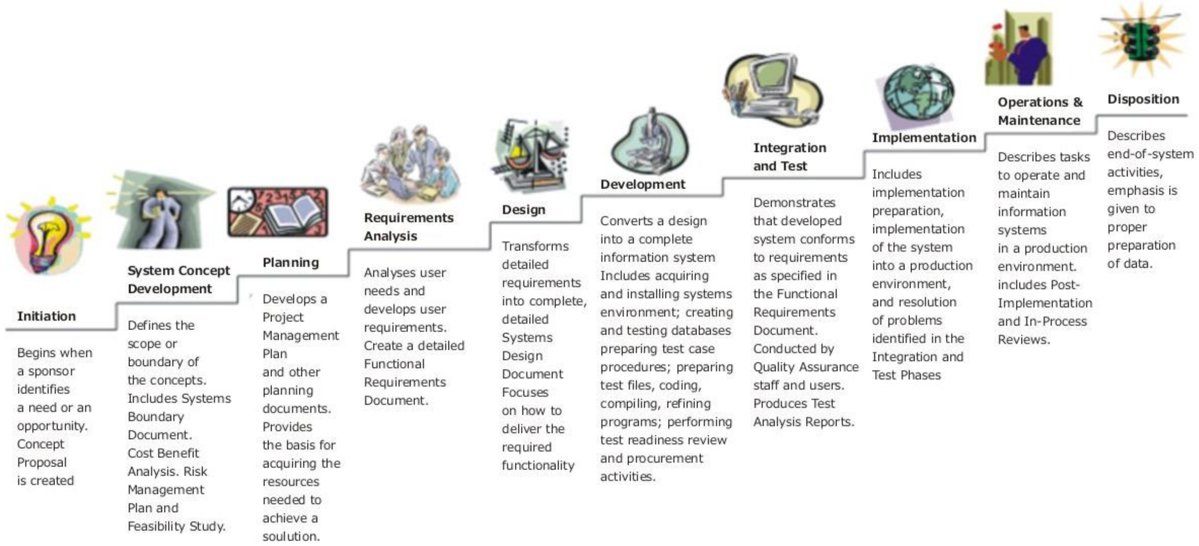
Main advantages

Version Tracking
Coordinating Teams
Due Diligence

Example:

Github [https://github.com/]
SourceForge [http://sourceforge.net/]
Google Code [http://code.google.com/]
Git (open source) [http://git-scm.com/]
CVS (open source) [http://www.nongnu.org/cvs/]
Arch (open source) [http://www.gnu.org/software/gnu-arch/]
Subversion (open source) [http://subversion.apache.org/]
BitKeeper [http://www.bitkeeper.com/]
Perforce [http://www.perforce.com/]
AccuRev [http://www.accurev.com/accurev-version-control.html]
Visual Source Safe [http://www.microsoft.com/vstudio/previous/ssafe/]
ClearCase/DOORS- IBM[http://www.ibm.com/software/products/en/clearcase]

System Development Life Cycle



Initiation
Begins when a sponsor identifies a need or an opportunity. Concept Proposal is created

System Concept Development
Defines the scope or boundary of the concepts. Includes Systems Boundary Document. Cost Benefit Analysis. Risk Management Plan and Feasibility Study.

Planning
Develops a Project Management Plan and other planning documents. Provides the basis for acquiring the resources needed to achieve a soulution.

Requirements Analysis
Analyses user needs and develops user requirements. Create a detailed Functional Requirements Document.

Design
Transforms detailed requirements into complete, detailed Systems Design Document Focuses on how to deliver the required functionality

Development
Converts a design into a complete information system Includes acquiring and installing systems environment; creating and testing databases preparing test case procedures; preparing test files, coding, compiling, refining programs; performing test readiness review and procurement activities.

Integration and Test
Demonstrates that developed system conforms to requirements as specified in the Functional Requirements Document. Conducted by Quality Assurance staff and users. Produces Test Analysis Reports.

Implementation
Includes implementation preparation, implementation of the system into a production environment, and resolution of problems identified in the Integration and Test Phases

Operations & Maintenance
Describes tasks to operate and maintain information systems in a production environment. includes Post-Implementation and In-Process Reviews.

Disposition
Describes end-of-system activities, emphasis is given to proper preparation of data.

Fig 8: SDLC Cycle

C Programming

First Program

Objective: Just for being familiar with vi-editor

```
/* First Program       */
/*Author: Ritesham S   */
/*www.rapidcode.co.in */

#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

Output:

```
$hello, world
```

Objective: Just for being familiar with vi-editor & observing the program.

```
/* Calculator-Sum Program      */
/*Author: Ritesham S   */
/*www.rapidcode.co.in */

# include<stdio.h>
int main()
{
     int a,b,sum;
     a=5;
     b=2;
     sum = a + b;
     printf(“Result = %d\n”,sum);
     return 0;
 }
```

```
Output:     $ cc Add.c
            $ ./a.out
            Result = 7
```

Comment Lines

1. Comment Lines- Human readable descriptions
2. Comment Lines- Ignored by compiler, useful in testing & Debugging
3. Comment Lines - Description about report/version/Tester/author

Compiling and Running on Unix/Linux

To compile program under Unix operating system use the command:

```
$ cc test.c
```

and under linux type

```
$ gcc test.c
```

The resulting executable file is a.out file. To run this executable you must type:

```
$./a.out
```

Program output must appear on your screen.

```
Hello World!
```

Features of C

Robust language, which can be used to write any complex program.
Has rich set of built-in functions and operators.
Well-suited for writing both system software and business applications.
Efficient and faster in execution.
Highly portable.
Well-suited for structured programming.
Dynamic Memory Allocation.

C Program Compilation Steps

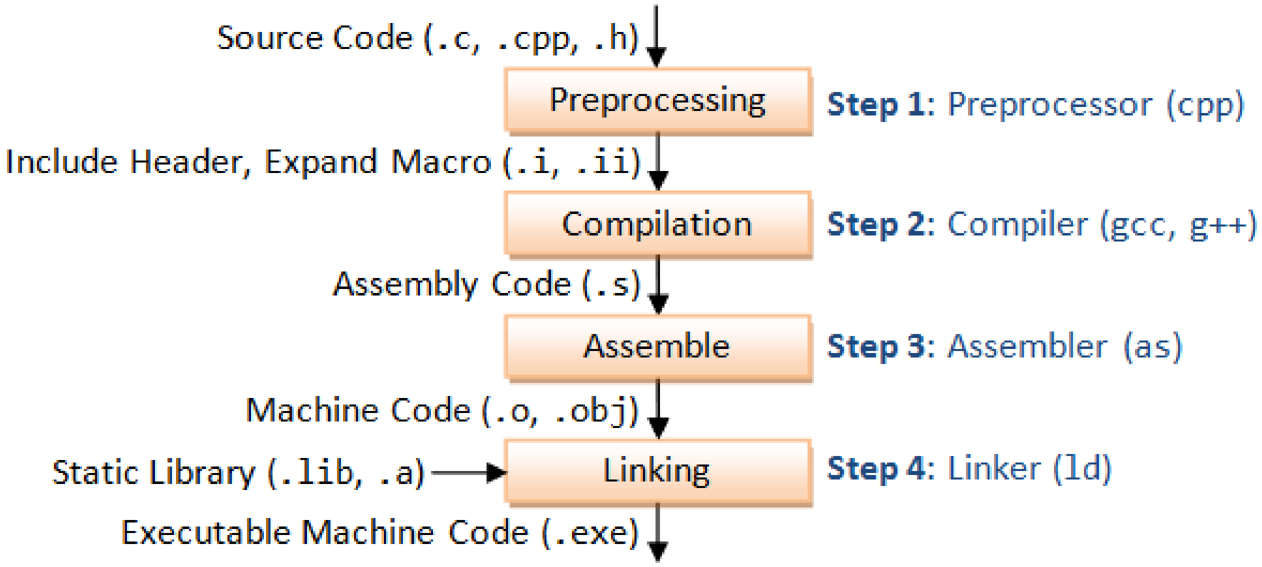

Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

Fig 9: C Program Compilation Steps

Details of Test program

- `#include <stdio.h>`

Tells C compiler to include the file "stdio.h" in this point of your C program before starting compile step. This "include file" contains several definitions, declarations etc.

- `main()`

C program consist of one or more functions. Functions are building blocks of C programs. main() function is different from other functions by that it is the start point of program execution. Our program contains only function while complicated programs may contain thousands.

- {

Opening brace marks the start of a block. Closing brace will mark its end. This one marks main() function start

- `printf("Hello world!");`

This line of code prints the statement between quotation marks on your output screen. \n tells program to start a new line in output screen.

- Each command line in C ends with ";" character. Control statements are exceptions. You will soon be able to determine when you must use ; to end a line of code.

- }

closes main() function.

Data Types and Variables

These include characters, integer numbers and float numbers. In C language you must declare a variable before you can use it. By declaring a variable to be an integer or a character for example will let computer to allocate memory space for storing and interpreting data properly.

S.N.	Types and Description
1	**Basic Types:** They are arithmetic types and consists of the two types: (a) integer types and (b) floating-point types.
2	**Enumerated types:** They are again arithmetic types and they are used to define variables that can only be assigned certain discrete integer values throughout the program.
3	**The type void:** The type specifier *void* indicates that no value is available.
4	**Derived types:** They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

Fig 10: Table Data type & Description

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Fig 11: Table Integer Data type & Value Range

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Fig 12: Table Float Data type & Precision

Naming a variable

- use meaningful names for your variables
- Also take this in mind that C is case sensitive.
- Functions and commands are all case sensitive in C
- You can use letters, digits and underscore _ character to make your variable names. Variable names can be up to 31 characters in ANSI C language.

Declaration & Definition Variable

Declaring Variable-

Whenever we write declaration statement then memory will not be allocated for the variable.

- Variable declaration will randomly specify the memory location.
- Compiler will not look for other details such as definition of the variable.
- Handy way to write code in which actual memory is not allowed.

- Identify the data type of identifier

Example:

```
        int rit;
        float f_rit;
        struct book
        {
                int page;
                float price;
                char name;
        };
```

Definition Variable-

Whenever we write declaration statement then memory will be allocated for the variable.

- Definition of variable will assign some value to it.

Example:

```
        struct book b1;
```

Note- Re-Declaration & Re-Definition is illegal in C

Initialization & Assignment

Initialization is assigning value while declaring the variable / at the time of birth itself.

```
        int a = 1
```

Assignment is just assigning value to a variable –

```
        int a;
        a = 1;
```

Format specifiers

Format specifier	Characters matched	Argument type
%c	any single character	char
%d, %i	integer	integer type
%u	integer	unsigned
%o	octal integer	unsigned
%x, %X	hex integer	unsigned
%e, %E, %f, %g, %G	floating point number	floating type
%p	address format	void *
%s	any sequence of non-whitespace characters	char

Fig 13: Table of format specifier

%d print as decimal integer

%6d print as decimal integer, at least 6 characters wide

%f print as floating point

%6f print as floating point, at least 6 characters wide

%.2f print as floating point, 2 characters after decimal point
%6.2f print as floating point, at least 6 wide and 2 after decimal point

Know this arithmetic while writing simple program of calculator-

```
     int   operator   int   =    int
     Int   operator   float =    float
     float operator   int   =    float
     float operator   float =    float
```

Lvalues and Rvalues in C:
There are two kinds of expressions in C:

lvalue : Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.

rvalue : The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement:

Escape Sequence
Certain characters can be represented in character and string constants by escape sequences like \n (newline); these sequences look like two characters, but represent only one.

- \a alert (bell) character
- \\ backslash
- \b backspace
- \? question mark
- \f formfeed
- \’ single quote
- \n newline
- \" double quote
- \r carriage return
- \ooo octal number
- \t horizontal tab
- \xhh hexadecimal number
- \v vertical tab
- The character constant ’\0’ represents the character with value zero the null character. ’\0’ is often written instead of 0 to emphasize the character nature of some expression, but the numeric value is just 0.

Internal Typecasting & External Typecasting

Typecasting is a way to make a variable of one type, such as an int, act like another type, such as a char, for one single operation.

Internal typecasting

```
      float b = 2;
      printf(“b = %f”,b);
```

External typecasting

```
      int a = 6;
      void *ptr = &a;
      printf(“a = %d” , *(int *)ptr);
```

Precedence and Order of Evaluation

Operators Associatively

```
      () [] -> . left to right
      ! ~ ++ -- + - * (type) sizeof right to left
      * / % left to right
      + - left to right
      << >> left to right
      < <= > >= left to right
      == != left to right
      & left to right
      ^ left to right
      | left to right
      && left to right
      || left to right
      ?: right to left
      = += -= *= /= %= &= ^= |= <<= >>= right to left
      , left to right
      Unary & +, -, and * have higher precedence than the binary forms.
```

UARL

sizeof is the only compile time operator

→Binary → Unary → Ternary

Arithmetic:

$\underbrace{+\ (2) \qquad -\ (2) \qquad *\ (1) \qquad \%\ (1) \qquad /\ (1)}_{\text{Binary}}$ (L → R)

Relational:

$\underbrace{<\ (1) \quad <=\ (1) \quad >\ (1) \quad >=\ (1) \quad ==\ (2) \quad !=\ (2)}_{\text{Binary}}$ (L → R)

Logical:

(R → L) $\underbrace{!\ (1)}_{\text{Unary}}$ $\underbrace{\&\&\ (2)\qquad ||(3)}_{\text{Binary}}$ (L → R)

Bitwise [Distributed]

$\underbrace{\&\quad |\quad \wedge\quad <<\quad >>}_{\text{Binary}}$ $\qquad\qquad\underbrace{\sim}_{\text{Unary}}$

Ternary:

? :

C Loops Explained with Examples (For Loop, Do While and While)

Loops are very basic and very useful programming facility that facilitates programmer to execute any block of code lines repeatedly and can be controlled as per conditions added by programmer. It saves writing code several times for same task.

There are three types of loops in C.

For loop
Do while loop
While loop

1. For Loop Examples

Basic syntax to use 'for' loop is:

```
for (variable initialization; condition to control loop; iteration of variable)
      {
      statement 1;
      statement 2;
      ..
      ..
      }
```

In the pseudo code above :

Variable initialization is the initialization of counter of loop.
Condition is any logical condition that controls the number of times the loop statements are executed.
Iteration is the increment/decrement of counter.
It is noted that when 'for' loop execution starts, first variable initialization is done, then condition is checked before execution of statements; if and only if condition is TRUE, statements are executed; after all statements are executed, iteration of counter of loop is done either increment or decrement.

Here is a basic C program covering usage of ‘for’ loop in several cases:

```
#include <stdio.h>
int main ()
{
    int i = 0, k = 0;
    float j = 0;
    int loop_count = 5;

    printf("Case1:\n");
    for (i=0; i < loop_count; i++)
    {
        printf("%d\n",i);
    }

    printf("Case2:\n");
    for (j=5.5; j > 0; j--)
    {
        printf("%f\n",j);
    }

    printf("Case3:\n");
    for (i=2; (i < 5 && i >=2); i++)
    {
        printf("%d\n",i);
    }

    printf("Case4:\n");
    for (i=0; (i != 5); i++)
    {
        printf("%d\n",i);
    }

    printf("Case5:\n");
    /* Blank loop  */
    for (i=0; i < loop_count; i++) ;

    printf("Case6:\n");
    for (i=0, k=0; (i < 5 && k < 3); i++, k++)
    {
        printf("%d\n",i);
    }

    printf("Case7:\n");
    i=5;
    for (; 0; i++)
    {
        printf("%d\n",i);
    }
    return 0;
}
```

Here is the output of the above program :

```
# ./a.out
Case1:
0
1
2
3
4
Case2:
5.500000
4.500000
3.500000
2.500000
1.500000
0.500000
Case3:
2
3
4
Case4:
0
1
2
3
4
Case5:
Case6:
0
1
2
```

Case7:
Loop can run infinitely if condition is set to TRUE always or no condition is specified.
For example:

```
for (;;)
```

If loop contain only one statement then braces are optional; generally it is preferred to use braces from readability point of view.
For example:

```
for (j=0;j<5;j++)
     printf("j”);
```

Loops can be nested too. There can be loop inside another loop. Given below is example for nested loop to display right angle triangle of ‘@’ symbol.

```
for (i=0; i < 5; i++)
{
    for (j=0;j<=i;j++)
    {
            printf("@”);
      }
}
```

Just like For Loops, it is also important for you to understand C Pointers fundamentals.

2. Do While Loop Examples

It is another loop like ‘for’ loop in C. But do-while loop allows execution of statements inside block of loop for one time for sure even if condition in loop fails.

Basic syntax to use ‘do-while’ loop is:

```
variable initialization;
do {
statement 1;
statement 2;
..
..
iteration of variable;
} while (condition to control loop)
```

In the pseudo code above :

Variable initialization is the initialization of counter of loop before start of ‘do-while’ loop.
Condition is any logical condition that controls the number of times execution of loop statements
Iteration is the increment/decrement of counter
Here is a basic C program covering usage of ‘do-while’ loop in several cases:

```
#include <stdio.h>
int main ()
{
        int i = 0;
        int loop_count = 5;

        printf("Case1:\n");
        do {
                printf("%d\n",i);
                i++;
                } while (i<loop_count);

        printf("Case2:\n");
        i=20;
        do {
                printf("%d\n",i);
                i++;
        } while (0);

        printf("Case3:\n");
        i=0;
```

```
      do {
           printf("%d\n",i);
           } while (i++<5);

      printf("Case4:\n");
      i=3;
      do {
           printf("%d\n",i);
           i++;
           } while (i < 5 && i >=2);
      return 0;
}
```

Output:

```
# ./a.out
 Case1:
 0
 1
 2
 3
 4
 Case2:
 20
 Case3:
 0
 1
 2
 3
 4
 5
 Case4:
 3
 4
```

3. While Loop Examples

It is another loop like ‘do-while’ loop in C.
The ‘while’ loop allows execution of statements inside block of loop only if condition in loop succeeds.

Basic syntax to use ‘while’ loop is:

```
variable initialization;
 while (condition to control loop)
{
     statement 1;
     statement 2;
 iteration of variable;
 }
```

Basic C program covering usage of ‘while’ loop in several cases:

```
#include <stdio.h>

int main () {

    int i = 0;
    int loop_count = 5;

    printf("Case1:\n");
    while (i<loop_count)
    {
        printf("%d\n",i);
        i++;
    }

    printf("Case2:\n");
    i=20;
    while (0)
    {
        printf("%d\n",i);
        i++;
    }

    printf("Case3:\n");
    i=0;
    while (i++<5)
   {
        printf("%d\n",i);
    }
    printf("Case4:\n");
    i=3;
    while (i < 5 && i >=2)
   {
        printf("%d\n",i);
        i++;
    }

    return 0;
}
```

Output:

```
# ./a.out
Case1:
0
1
2
3
4
Case2:
```

```
Case3:
1
2
3
4
5
Case4:
3
4
```

Read more assembly instructions set for the conditional statements
1.Narrow & lesser case value
2.Distributed/ Wide and lesser value
3. Distributed/ Wide and more value

Break and continue

To exit a loop you can use the break statement at any time. This can be very useful if you want to stop running a loop because a condition has been met other than the loop end condition. Take a look at the following example:

```
#include<stdio.h>

    int main()
    {
        int i;

        i = 0;
        while ( i < 20 )
        {
            i++;
            if ( i == 10)
                break;
        }
        return 0;
    }
```

In the example above, the while loop will run, as long i is smaller then twenty. In the while loop there is an if statement that states that if i equals ten the while loop must stop (break).

With “continue;” it is possible to skip the rest of the commands in the current loop and start from the top again. (the loop variable must still be incremented). Take a look at the example below:

```
#include<stdio.h>

     int main()
     {
          int i;

          i = 0;
          while ( i < 20 )
          {
               i++;
               continue;
               printf("Nothing to see\n");
          }
          return 0;
     }
```

In the example above, the printf function is never called because of the “continue;”.

goto statement

A goto statement in C programming language provides an unconditional jump from the goto to a labeled statement in the same function.

NOTE: Use of goto statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten so that it doesn't need the goto.

syntax:

```
goto label;
..
.
label: statement;
```

```
#include <stdio.h>
 int main ()
{
   /* local variable definition */
   int a = 10;

   /* do loop execution */
   LOOP:do
   {
      if( a == 15)
      {
         /* skip the iteration */
         a = a + 1;
         goto LOOP;
      }
      printf("value of a: %d\n", a);
      a++;

   }while( a < 20 );

   return 0;
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Functions in C

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, function `strcat()` to concatenate two strings, function `memcpy()` to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure, etc.

Defining a Function:

The general form of a function definition in C programming language is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

A function definition in C programming language consists of a function header and a function body. Here are all the parts of a function:

Return Type: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body: The function body contains a collection of statements that define what the function does.

Example:

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function Declarations:
A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

Calling a Function:
While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main ()
{
  /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2)
{
   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

We kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:

```
Max value is : 200
```

Function Arguments:

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. While calling a function, there are two ways that arguments can be passed to a function:

Call by value

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Call by reference

This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Function Stack

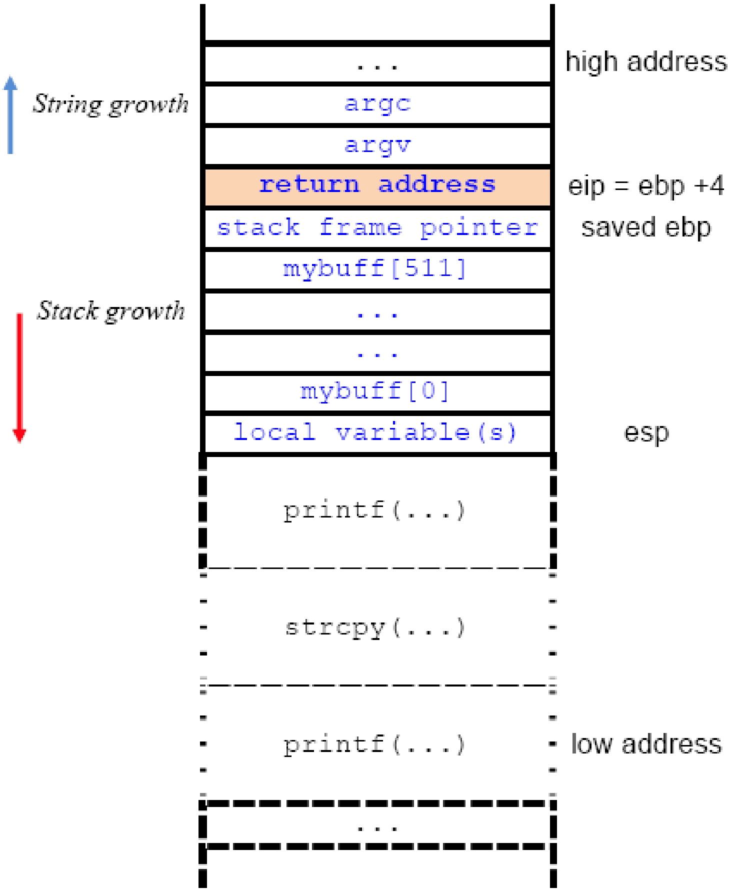


Fig 14: Function stack grow

Advance Stack View

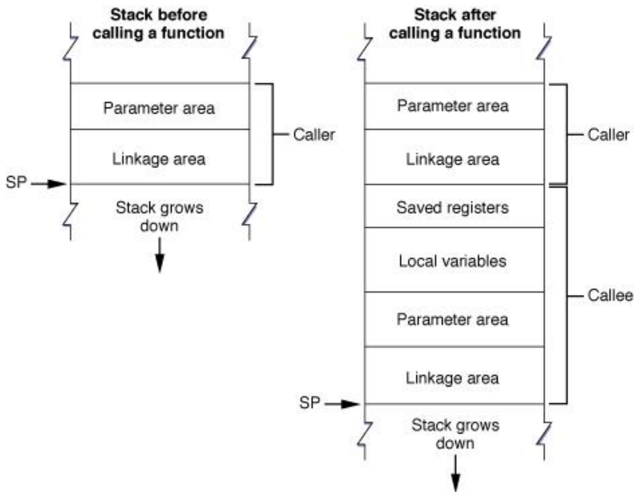


Fig 15: Advance stack view of RAM

Recursion: Beautiful concept in C just replica of loop concept

Recursion is a programming technique that allows the instructions is to "repeat the process" like loops. If a programming allows you to call a function inside the same function that is called recursive call of the function.

A simple example of recursion would be:

```
void recurse()
{
    recurse(); /* Function calls itself */
}

int main()
{
    recurse(); /* Sets off the recursion */
    return 0;
}
```

example:

```
void count_to_ten ( int count )
{
    /* we only keep counting if we have a value less than ten
       if ( count < 10 )
       {
           count_to_ten( count + 1 );
       }
}
int main()
{
  count_to_ten ( 0 );
}
```

Number Factorial

```
#include <stdio.h>

int factorial(unsigned int i)
{
   if(i <= 1)
   {
      return 1;
   }
   return i * factorial(i - 1);
}
int  main()
{
    int i = 15;
    printf("Factorial of %d is %d\n", i, factorial(i));
    return 0;
}
```

```
Output:

Factorial of 15 is 2004310016
```

Fibonacci Series

```
#include <stdio.h>

int fibonaci(int i)
{
   if(i == 0)
   {
      return 0;
   }
   if(i == 1)
   {
      return 1;
   }
   return fibonaci(i-1) + fibonaci(i-2);
}

int  main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
       printf("%d\t%n", fibonaci(i));
    }
    return 0;
}
```

```
Output:
0     1     1     2     3     5     8     13    21    34
```

Learn the recursion by tracing your result, very useful concept for industrial project programming, practice it well.

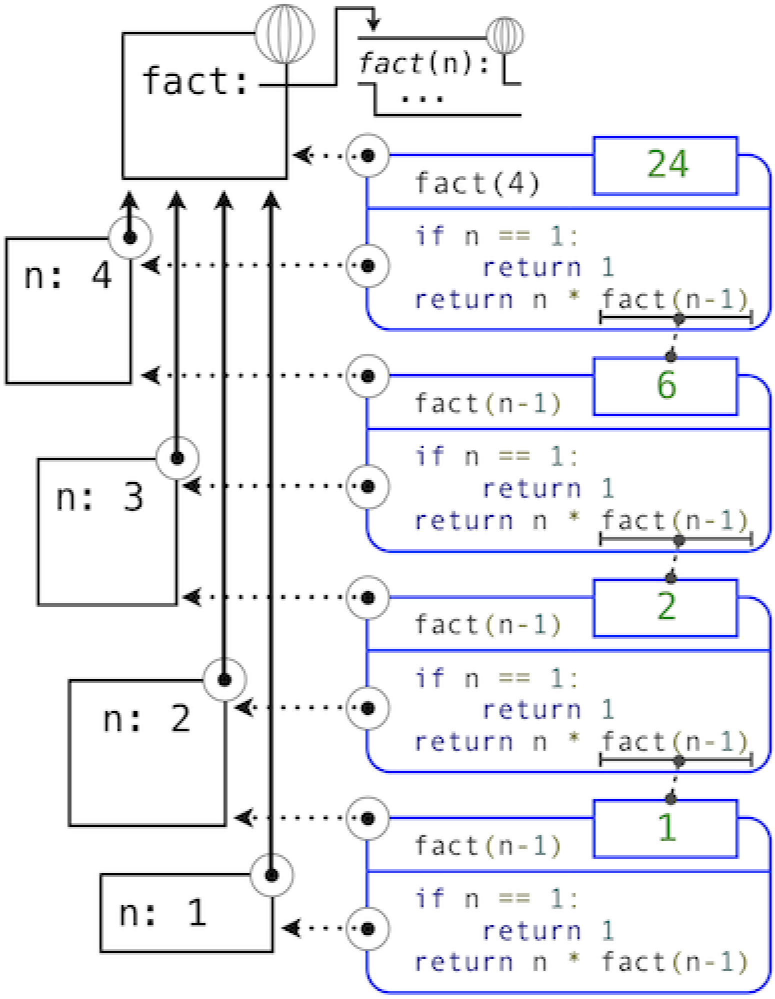


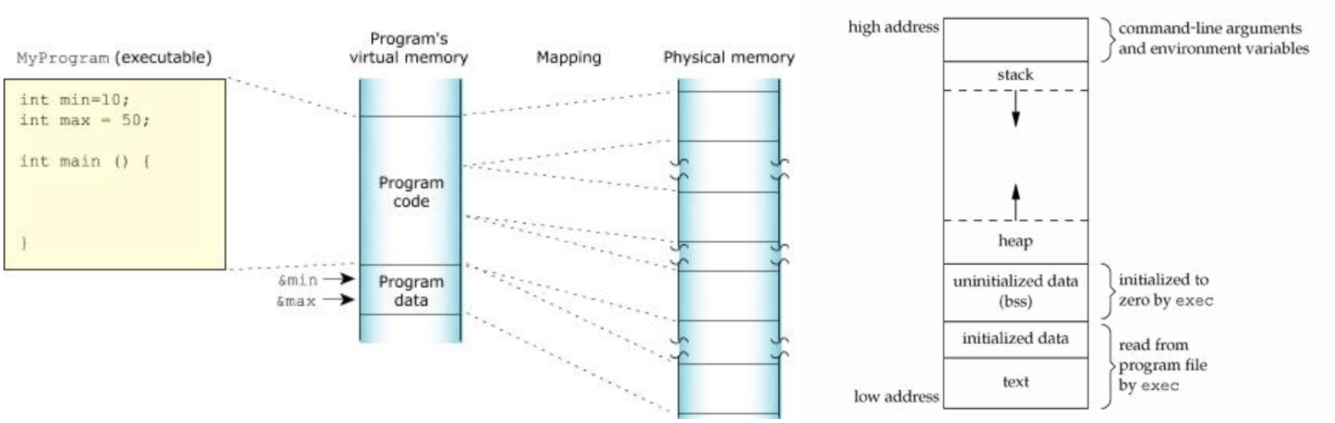
Fig 16: Recursive tracing for factorial program

Storage Class

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. These specifiers precede the type that they modify.

There are the following storage classes, which can be used in a C Program

- auto
- register
- static
- extern



```
int min=10;
int max = 50;

int main () {

}
```

Fig 17: Understanding program scope to memory layout

The auto Storage Class

The auto storage class is the default storage class for all local variables.

```
{
   int mount;
   auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
   register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main()
{
   while(count--)
   {
      func();
   }
   return 0;
}
/* function definition */
void func( void )
{
   static int i = 5; /* local static variable */
   i++;

   printf("i is %d and count is %d\n", i, count);
}
```

You may not understand this example at this time because I have used function and global variables, which I have not explained so far. So for now let us proceed even if you do not understand it completely. When the above code is compiled and executed, it produces the following result:

```
i is 6 and count is 4
i is 7 and count is 3
```

```
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.
When you have multiple files and you define a global variable or function, which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();

main()
{
   count = 5;
   write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void)
{
   printf("count is %d\n", count);
}
```

Here, extern keyword is being used to declare count in the second file where as it has its definition in the first file, main.c. Now, compile these two files as follows:

```
 $gcc main.c support.c
```

This will produce a.out executable program, when this program is executed, it produces the following result:

```
5
```

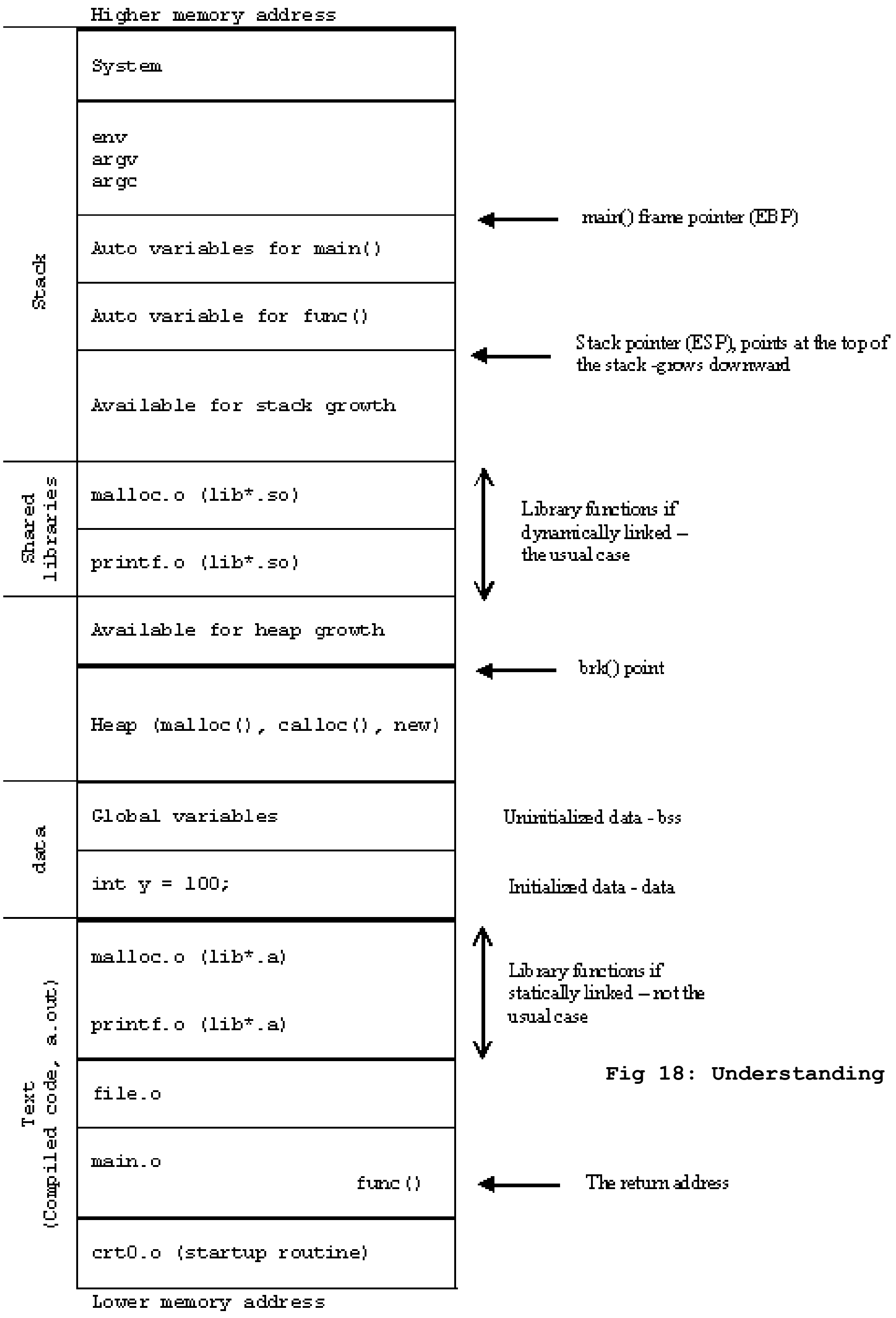
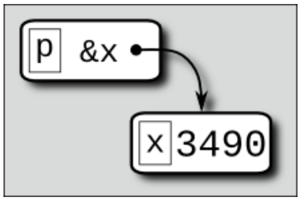

Higher memory address
Stack
System
env
argv
argc
Auto variables for main()
main() frame pointer (EBP)
Auto variable for func()
Stack pointer (ESP), points at the top of the stack -grows downward
Available for stack growth
Shared libraries
malloc.o (lib*.so)
printf.o (lib*.so)
Library functions if dynamically linked – the usual case
Available for heap growth
brk() point
Heap (malloc(), calloc(), new)
data
Global variables
Uninitialized data - bss
int y = 100;
Initialized data - data
Text (Compiled code, a.out)
malloc.o (lib*.a)
printf.o (lib*.a)
Library functions if statically linked – not the usual case
file.o
main.o
func()
The return address
crt0.o (startup routine)
Lower memory address

Fig 18: Understanding RAM over a C program function

	Auto	Register	Static	Global
Initial Value	Garbage	Garbage	Zero	Zero
Storage	RAM(Stack)	CPU Register	RAM(Data Seg.)	RAM(Data Seg.)
Scope	Within block	Within block	Within block	Across block
Life	Block ends	Block ends	Program ends	Program ends

Fig 19: Table to learn quick ! storage classes

Pointers



A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

The significance of pointers in C is the flexibility it offers in the programming. Pointers enable us to achieve parameter passing by reference, deal concisely and effectively either arrays, represent complex data structures, and work with dynamically allocated memory.

Although, a lot of programming can be done without the use of pointers, their usage enhances the Capability of language to manipulate data . Pointers are also used for accessing array elements, passing arrays and strings to functions, creating data structures such as linked lists, trees , graph, and so on.

What is a pointer variable?

Memory can be visualized as an ordered sequence of consecutively numbered storage locations. A data item is stored in memory in one or more adjacent storage locations depending upon its type. The address of a data item is the address of its first storage location. The address can be stored in another data item and manipulated in the program. The address of a data item is called a pointer to the data item and a variable that holds an address is called a pointer variable.

Uses of pointers

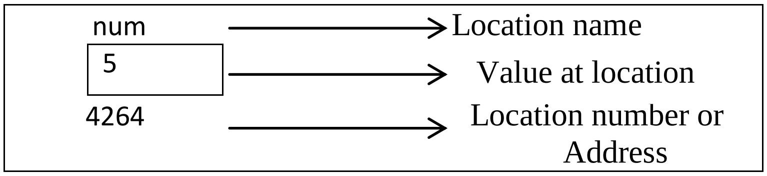
- Keep track of address of memory location.
- By changing the address in pointer type variable you can manipulate data in different memory allocation.
- Allocation of memory can be done dynamically.

Address and Dereferencing (& and *) Operators

Consider the declaration

```
  int num = 5;
```

The compiler will automatically assign memory for this data item. The data item can be accessed if we know the location (i.e, the address) of the first memory cell.



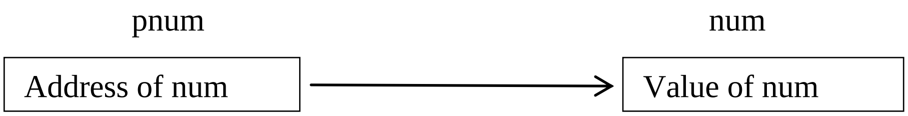
The address of num's memory location can be determined by the expression &num, where & is unary operator,called the 'address of' operator. It evaluates the address of its operand.
We can assign the address of num to another variable, pnum as:

```
pnum = &num;
```

The new variable pnum is called a pointer to num, since it points to the location where where num is stored in the memory. The pnum is referred to as a pointer variable.

The data item represented by num, can be accessed by the expression *pnum, where * is unary operator, called 'the value at the address' operator. It operates only a pointer variable.

It can be illustrated as below:



Relationship between pnum and num (where pnum =&num and num = *pnum).

Therefore, *pnum and num both represent the same data item.

According a data item through a pointer is called Dereferencing,and the operator asterisk (*) is called the 'dereferencing or indirection operator'.

Pointer type Declaration

Pointers are also variables and hence, must be defined in a program like any other variable. The Rules for declaring pointer variable names are the same as ordinary variables.

```
  type  *variable_name;
```

Where,

type	Data type of the variable pointed variable.
variable_name	Name of the pointer variable.
*(asterisk)	Signifies to the compiler that this variable has to be considered a pointer to the data type indicated by type.

For example,

int*int_ptr	int_ptr is a pointer to the data of type integer
char*ch_ptr	ch_ptr is a pointer to data of type character
double*db_ptr	db_ptr is a pointer to data of type double

Note: All the pointer variables will occupy 4 bytes of memory regardless of the type they point to.

Pointer Assignment

The address of (&) operator, when used as a prefix to the variable name, gives the address of that variable.
Thus,

```
   ptr = &i;
```

Assign address of variable i to ptr.

```
/*Example of '&'- address of operator*/
#include<stdio.h>
void main(void)
{
          int a=100;
          int b=200;
          int c=300;
          printf("Address: %u contains value :%d\n",&a,a);
          printf("Address: %u contains value :%d\n",&b.b);
          printf("Address: %u contains value :%d\n",&c,c);
 }
 Output
           Address : 65524 contains value : 100
           Address : 65520 contains value : 200
           Address : 65516 contains value : 300
```

Sample Code for '&' operator

For example, in the program below

```
int i=1, j, *ip;
ip=&i;
j=*ip;
*ip=0;
```

The first assignment assigns the address of variable i to ip.
The second assigns the value at address ip, that is, i tp j, and finally to the third assigns 0 to i
Since *ip is the same as i.

The two statements

```
 ip=&i;
j=*ip;
```

Are equivalent to the single assignment

```
j=*(&i);
```

or to the assignment

```
j=i;
```

i.e, the address of the operator & is the inverse of the dereferencing operator*. Consider the following segment of code

```
#include<stdio.h>
int main
{
     char*ch;
     char b='A';
     ch=&b;        /*assign address of b to ch*/
     printf("%c",*ch);
}
Output:                  A
```

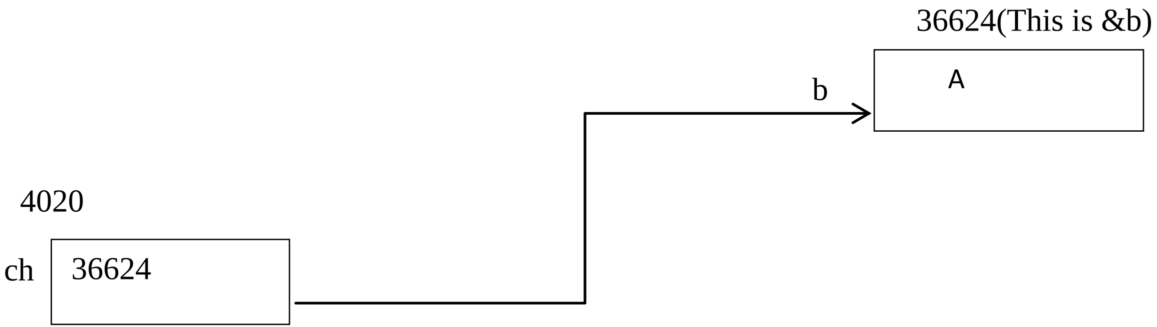


Fig. 6.2 Memory representation of pointer

In the above example,

b	value of b,which is 'A'
&b	address of b, i.e., 36624
ch	value of ch, which is 36624
&ch	Address of ch, i.e., 4020(arbitrary)
*ch	Contents of ch, =>value at 36624 i.e., A This is same as *(&b)

Pointer Initialisation

The declaration of pointer variable may be accompanied by an initializer. The form of an Initialization of a pointer variable is

```
   type * identifier=initializer;
```

The initializer must either evaluate to an address of previously defined data of appropriate type or it can be NULL pointer.

For example, the declaration

```
    float *fp=null;
```

The declarations

```
    Short s;
    short *sp;
    sp=&s;
```

Initialize sp to the address of s.
The declarations

```
     char c[10];
     char*cp=&c[4];
```

Initializes cp to the address of the fifth element of the array c.

```
char *cfp=&c[0];
```

Initialize cfp to the address of an array is also called as base address of array.

Following program illustrates declaration, assignment, and dereferencing of pointers.

```
/* Example : Usage of Pointers*/
#include<stdio.h>
int main(void)
{
    int i,j=1;
    int *jp1,*jp2=&j;      /*jp2 points to j*/
    jp1=jp2;               /*jp1 also points to j*/
    i=*jp1;                /*i gets the value of j*/
    *jp2=*jp1+I;           /*i is added to j*/
    printf("i=%d j=%d *jp1=%d *jp2=%d\n",i,j,*jp1,*jp2);
}
Output:
         i=1 j=2 *jp1=2 *jp2=2
```

Pointer Arithmetic

Arithmetic can be performed on pointers. However, in pointer arithmetic, a pointer is a valid operand only for the addition(+) and subtraction(-) operators.

An integral value n may be added to or subtracted from a pointer ptr. Assuming that the data item that pointer points to lies within an array of such data items. The result is a pointer to the data item that lays n data items after or before the one p points to respectively.

The value of ptr+-n is the storage location ptr+-n*sizeof(*ptr), where sizeof is an operator that yields the size in bytes of its operands.

Consider the following example

```
/*Example of Pointer arithmetic*/
#include<stdio.h>
int main(void)
{
     int i=3,*x;
     float j=1.5,*y;
     char k=’C’,*z;
     printf(“Value of i=%d\n”,i);
     printf(“Value of j=%f\n”,j);
     printf(“Value of k=%c\n”,k);
     x=&i;
     y=&j;
     z=&k;
     printf(“Original Value in x=%u\n”,x);
     printf(“Original Value in y=%u\n”,y);
     printf(“Original Value in z=%u\n”,z);
     x++;
     y++;
     z++;
     printf(“New Value in x=%u\n”,x);
     printf(“New Value in y=%u\n”,y);
     printf(“New Value in z=%u\n”,z);
}
Output:
     Value of i=3
     Value of j=1.500000
     Value of k=C
     Original Value in x=1002
     Original Value in y=2004
     Original Value in z=5006
     New Value in x=1006
     New Value in y=2008
     New Value in z=5007
```

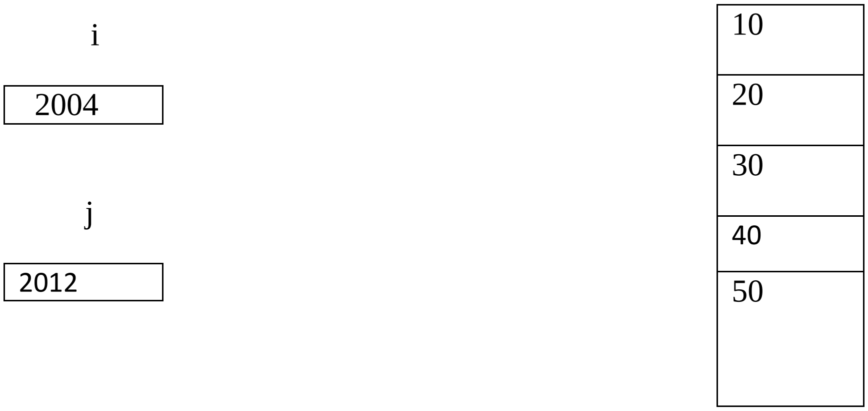
In the above example , New value in x is 1002(original value)+4, New value in y is 2004 (original value)+4, New value in z is 5006(original value)+1.

This happens because every time a pointer is incremented it points to the immediate next location of this type. That is why, when the integer pointer x is incremented, it points to an address four locations after the current location, since an int is always 4 bytes long. Similarly ,
y points to an address 4 locations after the current locations and z points 1 location after the current location.

Some valid pointer arithmetics are
Addition of a number to a pointer.
Subtraction of a number from a pointer.
For example, if p1 and p2 are properly declared pointers, then the following statements are valid.

```
    y=*p1**p2; /*same as (*p1)*(*p2)*/
    sum=sum+*p1;
    z=5*-*p2/*p1; /*same as (5*(-(*p2)))/(*p1)*/
    *p2=*p1+10;
```

C allows subtracting one pointer from another. The resulting value indicates the number of bytes separating the corresponding array elements. This is illustrated in the following example:



Memory Representation of Pointer Arithmetic

The result of expression (j-1) is not 8 as expected (2012-2004) but 2.
This is because when a pointer is decremented (or incremented) it is done so by the length of the data type it points to,called the scale factor.

```
   (j-1)=(2012-2004)/4=2
```

as size of int is 4.
This is called reference by address.
Some invalid pointer arithmetics are
Addition two pointers.
Multiplication of a number with a pointer.
Division of a pointer with a number.

Pointer Comparison

The relational comparison ==,!= are permitted between pointers of the same type.

The relational comparison <, <=, >, >= are permitted between pointers of the same type and the result depends on the relative location of the two data items pointed to.

For example,

```
   int a[10],*ap;
```

the expression

```
   ap==&a[9];
```

is true if ap is pointing to the last element of the array a, and the expression

```
   ap<&a[10];
```

is true as long as ap is pointing to one of the element of a.

Pointers and Functions

A function can take a pointer to any data type, as argument and can return a pointer to any data type. For example, the function definition

```
double*maxp(double*xp,double*yp)
{
   return *xp>=*yp?x;
}
```

specifies that the function maxp() return a pointer to a double variable, and expects two arguments, both of which are pointers to double variables. The function dereferences the two argument pointers to get the values of the corresponding variables, and returns the pointer to the variable that has the larger of the two values. Thus give that,

```
double u=1,v=2,*mp;
```

the statement

```
mp=maxp(&u,&v);
```

makes mp point to v.

Call by Value

In a call by value, values of the arguments are used to initialize parameters of the called function, but the addresses of the arguments are not provided to the called function. Therefore, any change in the value of a parameter in the called function is not reflected in the variable supplied as argument in the calling function.

```
/*Example: function parameters passed by Value*/
#include<stdio.h>
int main(void)
{
   int a=5,b=7;
   void swap(int,int);
   printf(“Before function call: a=%d b=%d”,a,b);
   swap(a,b); /*Variables a and b are passed by value*/
   printf(“After function call: a=%d b=%d”,a,b);
}
Void swap(int x, int y)
{
   int temp;
   temp=x;
   x=y;
   y=temp;
}
Output:
   Before function call: a=5 b=7
   After function call: a=7 b=7
```

Call by Reference

In contrast, in a call by reference, addresses of the variables are supplied to the called function and changes to the parameter value in the called function cause changes in the values of the variable in the calling function.

Call by reference can be implemented by passing pointers to the variables as arguments to the function. These pointers can then be used by the called function to access the argument variables and change them.

```
/*Example : Arguments as pointers*/
#include<stdio.h>
int main(void)
{
   int a=5,b=7;
   void swap(int*,int*);
   printf(“Before function call: a=%d b=%d”,a,b);
   swap(&a,&b); /*Address of variable a and b is passed*/
   printf(“After function call: a=%d b=%d”,a,b);
}
Void swap(int*x,int*y)
{
   int temp;
   /*The content of memory location are changed*/
   temp=*x;
   *x=*y;
   *y=temp;
}
```

```
Output:
      Before function call: a=5 b=7
      After function call: a=7 b=5
```

Steps involved for using pointers in a function are

1. Pass address of the variable (Using the ampersand(&) or direct pointer variables).
2. Declare the variable as pointers within the routine.
3. Refer to the values contained in a memory location via asterisk(*).

Using call by reference, we can make a function return more than one value at a time, as shown in the program below:

```
/*Returning more than one values from a function through arguments*/
#include<stdio.h>
int main(void)
{
   float radius;
   float area, peri;
   void areaperi(float,float*,float*);
   printf(“Enter radius:”);
   scanf(“%f”,&radius);
   areaperi(radius,&area,&peri);
   printf(“\nArea=%f.2f”,area);
   printf(“Perimeter=%.2f”,peri);
}
void areaperi(float r,float*a,float*p)
{
   *a=3.14*r*r;
   *p=2*3.14*r;
}
Output:
   Enter radius of a circle: 5
   Area= 78.50
   Perimeter= 31.40
```

Pointers to Functions

1. Functions have addresses just like data items. A pointer to a function can be defined as the address of the code executed when the function is called. A function’s address is the starting address of the machine language code of the function stored in the memory.
2. Pointers to functions are used in
3. writing memory resident programs.
4. writing viruses, or vaccines to remove the viruses

Address of a Function

The address of a function can be obtained by only specifying the name of the function without the trailing parantheses.

For example, if CalcArea is a function already defined, then CalcArea is the address of the function CalcArea().

Declaration of a Pointer to a Function

The declaration of a pointer to a function requires the function’s return type and the function’s argument list to be specified along with the pointer variable.

The general syntax for declaring a pointer to a function is as follows:

```
return-type(*pointer variable)(function’s argument list);
```

Thus , the declaration

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

declares fp to be a variable of type “pointer to a function that takes two integer arguments and return an integer as its value.” The identifiers i and j are written for descriptive purpose only,

The preceding declaration can, therefore also be written as

```
int(*fp)(int,int);
```

Thus declarations

`int i(void)`	declares i to a function with no parameters that return an int.
`int*pi(void)`	declares pi to a function with no parameters that returns a pointer to an int.
`int(*ip)(void)`	declares ip to be a pointer to a function that returns an integer value and takes no argument.

```
/*Example: Pointer to Function*/
#include<stdio.h>
int func1(int i)
{
   return(i);
}
float func2(float f)
{
   return(f);
}
```

```
int main(void)
{
   int(*p)(int);  /*declaring pointer to function*/
   float(*q)(float);
   int i=5;
   float f=1.5;
   p=func1;  /*assigning address of function func1 to p*/
   q=func2;  /*assigning address of function func2 to q*/
   printf(“i=%d f=%f\n”,p(i),q(f));
}
```

After declaring the function prototypes and two pointers p and q to the function; p is assigned the address of function func1 and q is assigned the address of function func2.

Invoking a Function by using Pointers

In the pointer declaration to functions, the pointer variable along with operator(*) plaays the role of the function name. Hence, while invoking function by using pointers, the function name is replaced by the pointer variable.

```
/*Example: Invoking function using pointers*/
#include<stdio.h>
int main(void)
{
   unsigned int fact(int);
   unsigned int ft,(*ptr)(int);
   int n;
   ptr=fact; /*assigning address of fact() to ptr*/
   printf(“Enter integer whose factorial is to be found:”);
   scanf(“%d”,&n);
   ft=ptr(n); /*call to function fact using pointer ptr*/
   printf(“Factorial of %d is %u \n”,n,ft);
}
unsigned int fact(int m)
{
   unsigned int I,ans;
   if(m==0)
       return(1);
   else
   {
       for(i=m,ans=1;i>1;ans*=i--)
       return(ans);
   }
}
Output:
    Enter integer whose factorial is to be found: 8
    Factorial of 8 is 40320
```

Functions returning Pointers

We have already learnt that a function can return an int, a double or any other data type. Similarly it can return a pointer. However, to make a function return a pointer it has to be explicitly mentioned in the calling function as well as in the function declaration.

While retaining pointer, return the pointer to global variables or static or dynamically allocated address. Do not return any address of local variable because stop to exit after function call.

```
/*Example: Function returning pointers*/
/*Program to accept two numbers and find greater number*/
#include<stdio.h>
int main(void)
{
   int a, b,*c;
   int*check(int,int);
   printf(“Enter two numbers:”);
   scanf(“%d%d”,&a,&b);
   c=check(&a,&b);
   printf(“\n Greater numbers: %d”,*c);
}
int*check(int*p,int*q)
{
   if(*p>=*q)
       return(p);
   else
       return(q);
}
```

The address of integer being passed to check() are collected in p and q.

Then in the next statement the conditional operators test the value of *p and *q and return either the address stored in p or the address stored in q.

This address gets collected in c in main().

Pointers and Arrays

In C, there is a close correspondence between arrays and pointers that results not only in notational convenience but also in code that uses less memory and runs faster. Any operation that can be achieved by array subscripting can also be done with pointers.

Pointer to Array

Arrays are internally stored as pointers. A pointer can efficiently access the elements of an array.

```
/*Program to access array elements using pointers*/
#include<stdio.h>
int main(void)
{
static int ar[5]={10,20,30,40,50};
int i,*ptr;
ptr=&ar[0]; /*same as ptr=ar*/
   for(i=0;i<5;i++)
   {
   printf("%d-%d\n",ptr,*ptr);
      ptr++;
   }
}
Output:
            5000-10
            5004-20
            5008-30
            5012-40
            5016-50
```

An integer pointer, ptr is explicitly declared and assigned the starting address. The memory representation of above declared array ar (assuming an integer takes 4 bytes of storage) is shown below:

5000	5004	5008	5012	5016
10	20	30	40	50
ar[0]	ar[1]	ar[2]	ar[3]	ar[4]

Recall that an array name is really a pointer to the first element in that array. Therefore, address of the first element can be expressed as either &ar[0] or simply ar.

i.e. ar=&ar[0]=5000

Hence,

*ar=*ar(&ar[0])
i.e. *ar=ar[0] or *(ar+0)=ar[0]

To take the above statement more general, we can write

*(ar+i)=ar[i];

where, i=0,1,2,3,...

Hence any array element can be accessed using pointer notation, and vice versa.

It will be clear from following table:

char c[10],int i;	
Array Notation	Pointer Notation
&c[10]	c
c[i]	*(c+i)
&c[i]	c+i

For example, given that

```
char c[5]={‘a’,’e’,’i’,’o’,’u’};
char*cp;
```

and

```
cp=c;
```

and

c[0]	‘a’	*cp	cp[0]
c[1]	‘b’	*(cp+1)	cp[1]
c[2]	‘c’	*(cp+2)	cp[2]
c[3]	‘d’	*(cp+3)	cp[3]
c[4]	‘e’	*(cp+4)	cp[4]

Using this concept, we can write the above program as shown below.

```
#include<stdio.h>
int main(void)
{
    static int ar[5]={10,20,30,40,50};
    int i;
    for(i=0;i<5;i++)
        printf(“%d-%d\n”,(ar+i),*(ar+i);
}
```

Note: C does not allow to assign an address to an array.

Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

(a) We define a pointer variable
(b) Assign the address of a variable to a pointer and
(c) Finally access the value at the address available in the pointer variable.

```
#include <stdio.h>

int main ()
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

Output:

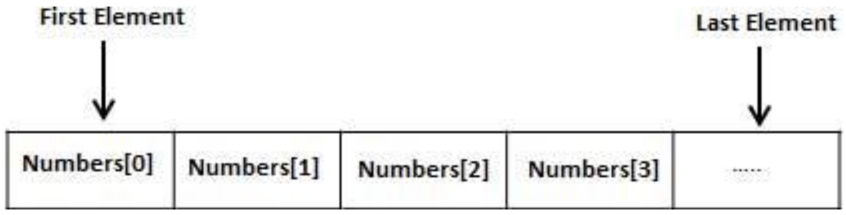
```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Null Pointers
Generic Pointers
Dangling Pointers
String Literlas

Array
C programming language provides a data structure called **the array**, which can store a fixed-size sequential collection of elements of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

1. Assignment
2. Initialization
3. Generic



Initializing Array

```
int a =7;

int a[] = {2,3,6,7};
```

Assignment Array

```
int a;
a=7;

int a[4];
int a[0] = 2, int a[0] = 3, int a[0] = 6, int a[0] = 7,
```

Generic Arrays

```
int a;
scanf(“%d”, &a);
printf(“%d”, a);
int a[5], int i;
for( i=0; i<5; i++)
      scanf(“%d”, &a[i]);
for( i=0; i<5; i++)
      printf(“%d”, a[i]);
```

a[0]	a[1]	a[2]	a[3]	a[4]
2	3	6	7	7
3000	3004	3008	3012	3016

This is called one-dimension array

Two Dimension array a[i][j]
Initializing Array

```
int a[2][2] = { {1,1}, {6, 6});
or
int a[2][2] = {1, 1, 6, 6};
```

Assignment Array

```
int a[2][2];
a[0][0] =1, a[0][1] = 1, a[1][0]= 6, a[1][1] = 6;
```

Generic Arrays

```
int a[2][2];
int i=0,j=0;
for( i=0; i<2; i++)
        for(j =0; j<2; j++)
        {
                scanf("%d", &a[i][j]);
        }
```

	Col[0]	col[1]	
Row a[0]	1	1	} 1-D Array
Row a[1]	2	2	} 1-D Array

Just have one hand-on exercise:

```
int a[4];                         Example

a[i]   ⇔   *(a + i)               a[3]  ⇔  *(a + 3)
&a[i]  ⇔   (a + i)                &a[3]  ⇔   (a + 3)

a[i][j]  ⇔  *(*(a+i)+j)           a[1][2]  ⇔  *(*(a+1)+2)
& a[i][j]  ⇔  (*(a+i)+j)          & a[1][2]  ⇔  (*(a+1)+2)


int a[4] ={ 1, 2, 3, 4};  //take some base address
int *p = a;
```

Do these are valid?

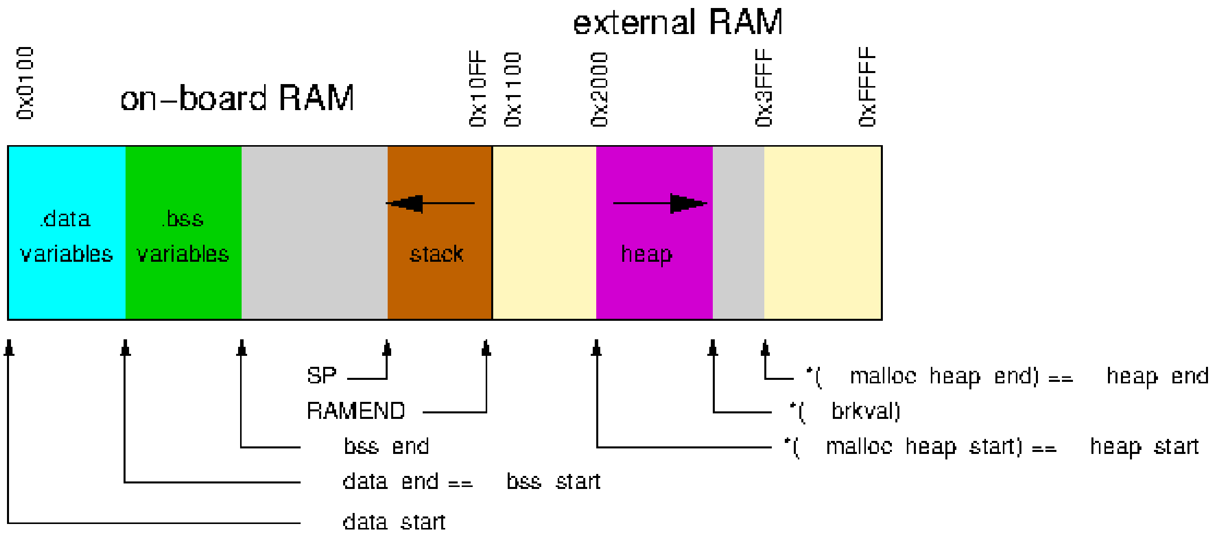
```
p++ ? p = p+1 ?  p-- ? p = p -1

int a[4] = { 1, 2, 3, 4}
int *p = (a + 0);
int *q = (a +3);
p = p + q ?    //invalid or valid
int result = q - p ? //invalid or valid
```

Dynamic Memory Allocation/De-allocation

Compiler please, gives me required space!

C dynamic memory allocation refers to performing manual memory management for dynamic memory allocation in the C programming language via a group of functions in the C standard library, namely

1. malloc, [for allocation]
2. realloc [for expand & shrink]
3. calloc [for allocation]
4. free. [for deallocation]



C library function - malloc()

```
void *malloc(size_t size)
int*p = (int*)malloc( 4*(sizeof(int));
```

C library function - calloc()

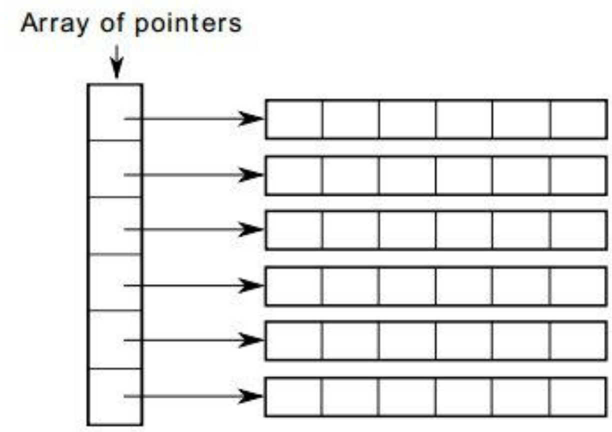
```
void *calloc(size_t nitems, size_t size)
```

C library function - free()

```
void free(void *ptr)
```

Dynamic Memory Allocation for 2D Array using Pointers

```
int **p = (int**)malloc (ROWS * sizeof( int *));
for (i = 0; i< ROW, i++)
p[i] = (int*) malloc ( COL * sizeof(int));
```



Take one more stub :-

```
int *p = a;
where int a[4] = {4, 5, 6, 7};

similarly
int (*ptr)[4] = a;
where
int a[4][4] = { {1,2,2,7}
                {5,5,6,3}
                {2,1,5,6}
                {1,2,3,4} };
```

Here we can say – ‘p’ is a pointer to int
& ptr is pointer to 1-D array

Let’s guess these-

```
int a[5] = { 1, 2, 3, 4, 5};
```

a[0]	a[1]	a[2]	a[3]	a[4]
1	2	3	4	5
3000	3004	3008	3012	3016

```
a =?, &a =?, a+1 =?, &a+1 =?
```

Now tryout

- WAP to pass int to a function & print its value there.
- WAP to pass 1-D array to function & print its value there.
- WAP to pass 2-D array to function & print its value there.

Arrays

An array has the following properties:

- The type of an array is the data type of its elements.
- The location of an array is the location of its first element.
- The length of an array is the number of data elements in the array.
- The storage required for an array is the length of the array times the size of an element.

Arrays, whose elements are specified by one subscript, are called one-dimensional arrays. These are commonly known as vectors .

Arrays, whose elements are specified by more than one subscript, are called multi-dimensional arrays. These are commonly known as matrix .

Declaration of Single Dimensional Array (Vectors)

Arrays, like simple variables, need to be declared before use.

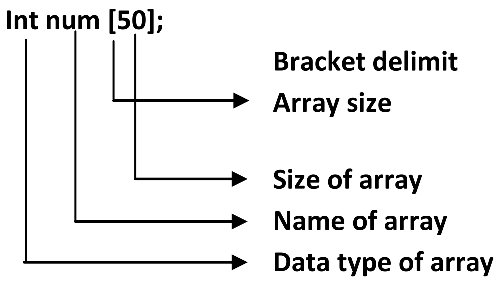
An array declaration is of the form:

```
[ storage class] data-type arrayname [size] ;
```

Where,

Storage class	Storage class of an array.
Data-type	The type of data stored in the array.
Arrayname	Name of the array.
Size	Maximum number of elements that the array can hold.

Hence, an array num of 50 integer elements can be declared as:



Int num [50];

Bracket delimit

Array size

Size of array

Name of array

Data type of array

Initialization of Dimensional array

Elements of an array can be assigned initial value by following the array definition with a list of initializers enclosed in braces and separated by commas.
For example, the declaration:

```
int Mark[5] = {40,97,91,88,100} ;
```

Declares an array mar to contain five integer elements and initializes the elements of array as given below:

Mark[0]	**40**
Mark[1]	**97**
Mark[2]	**91**
Mark[3]	**88**
Mark[4]	**100**

The declaration:

```
char Name [3] = {‘R’, ‘A, ‘J’} ;
```

Declares an array name to contain three character elements and initializes the elements of array as given below:

Name[0]	**‘R’**
Name[1]	**‘A’**
Name[2]	**‘J’**

The declaration:

```
float Price [7] = {0.25 , 15.5, 10.7, 26.8, 8.8, 2.8, 9.7};
```

Declares an array price to contain seven float elements and initializes the elements of array as given below:

Price[0]	0.25
Price[1]	15.5
Price[2]	10.7
Price[3]	26.8
Price[4]	8.8
Price[5]	2.8
Price [6]	9.7

Since any constant integral expression may be used to specify of elements in an array, symbolic constant or expressions involving symbolic constant may also apper in array declarations.

For example:

```
#define      UNIT_PRICE 80
#define      TOT_PRICE 100
int    s1_price [UNIT_PRICE] ;
 int nt_price [TOT_PRICE] ;
```

Declare s1_price and nt_price to be one-dimensional integer array of 80 and 100 elements respectively.

The array size may be omitted during declaration.
Thus, The declaration:

```
int mark[] = {40,97,91,88,100};
```

Is equivalent to the

```
int mark [5] = {40, 97, 91, 88, 100};
```

In such cases, the subscript is assumed to be equal to the number of elements in the array (5 in this case).
The elements, which are not explicitly initialized, are automatically set to zero,
E.g.:

```
int x[4]={1.2}; implies
    x[0]=1
    x[1]=2
    x[2]=0
    x[3]=0
```

Array elements in memory

Consider the following array declaration:

```
int num[100] ;
```

In the above declaration, 400 bytes get immediately reserved in memory , as each of the 100 integers would be of bytes long an array is a set of contiguous memory locations, first element starting at index zero. The allocation will be like this.

4000	4004	4008	4012	4020	4392	4396	
0	1	2	3	4	5	98	99
num[0]	num[0]	num[0]	num[0]	num[0]	num[0]	num[0]	num[0]

As seen above, array elements are always numbered (index) from 0 to (n-1) where n is the size of the array.

Array processing

The capability to represent a collection of related data items by a single array enables the development of concise and efficient programs.

An individual array element can be used in a similar manner that a simple variable is used. That is user can assign a value, display it's value of perform arithmetic operations on it.

To access a particular element in an array, specify the array name, followed by square braces enclosing an integer, which is called The Array index

For example, the assignment statement

```
num[5] =2;
```

Assigns 2 to 6th elements of num.

```
p = (net [1] + amount [9] /2) ;
```

Assigns the average value of 2nd element of net and 10th element of amount to p.

```
      -- num [8] ;
```

Decrements the content of 9th element of num by 1.

The assignment statements.

```
      i = 5 ;
      p = num[==i] ;
```

Assigns the value of num [6] to p.
Whereas the statements

```
      i = 5 ;
      p = num[==i] ;
```

Assigns the value of num [5] to p.
However, all operation involving entire array must be performed on an element-by-element basis. This done using loops. The number of loop iterations will hence equal to the number of array elements to be processed.

As an illustration of the use of arrays, consider the following program.

```
/* Program to find average marks obtained by 25 students in a test by accepting marks  of
each student*/

      #include<stdio.h>
      int main (void)
      {
            int i;
            float sum=0
            for(i=0;i<25;i++)
            {
            printf("enter marks : ");
            scanf("%f",&mark[i]);
            }
            printf("\n Average marks:%2f\n",sum/25);
      }
```

sample code using Arrays

Declaration of multi- dimension arrays

Declaration of multi dimension arrays

Syntax :

[strange class] data type array name [expr-1] [expr-2]… [expr-n];

Where,

Storage class	Storage class of an array.
Data-type	The type of data stored in the array.
Array name	Name of the array.
Expr-1	A constant integral expression specifying the name of elements in the 1st dimension of the array point.
Expr-n	A constant integral expression specifying the name of elements in the nth dimension of the array point.

The scope of this course will limit the discussion to 2-dimensional arrays only.

Thus a two-dimensional array will require two pair of square brackets. One subscript denotes the row and the other the column. All subscripts i.e. row and columns start with 0.

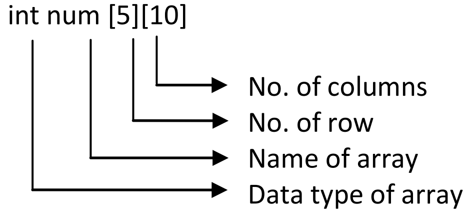
So, a two-dimensional array can be declared as

```
[storage class] data – type arrayname [expr1] [expr2] ;
```

Where,

Expr1	Maximum number of row that the array can hold
Expr2	Maximum number of columns that the array can hold

Hence, an array num of integer type holding 5 rows and 10 columns can be declared as



Initialization of two- dimensional arrays

Two-dimensional array are initialized analogously, with initializers list by rows. A pair of braces is used to separate the list of initializers for one row from the next, and commas are placed after each pair of brace except for the last row that row that close off a row.
e.g.

```
int no[3][4]={
              {1,2,3,4},
              {5,6,7,8},
              {9,10,11,12}
             };
```

Declares an array no of integer type to contain 3 row and 4 columns. The inner pairs of braces are optional. Thus the above declaration can equivalently be written as

```
int no[3][4]= {1,2,3,4,5,6,7,8,9,10,11,12};
```

Above array initializes the elements of array as given below:

no[0,0]=1	**no[0,1]=2**	**no[0,2]=3**	**no[0,3]=4**
no[1,0]=5	**no[1,1]=6**	**no[1,2]=4**	**no[1,3]=5**
no[2,0]=9	**no[2,1]=10**	**no[2,2]=11**	**no[2,3]=12**

Note: it is important to remember that while initializing a two-dimensional array it is necessary to mention the second (column) dimension, where as the first dimension (row) is optional.

```
int arr[2][3] ={12,34,56,78};      valid
int arr[ ][3] ={12,34,56,78};      valid
int arr[2][ ] ={12,34,56,78};      invalid
int arr[ ][ ] ={12,34,56,78};      invalid
```

Memory representation of Two-dimensional Arrays

A two-dimensional array a[i][j] can be visualized as a table or a matrix of I rows and j columns as shown below:

Row1	a[0][0]	a[0][1]	a[0][2]	----	a[0][j-2]	a[0][j-1]
Row 2	a[1][0]	a[1][1]	a[1][2]	----	a[1][j-2]	a[1][j-1]
Row i-1	a[i-2][0]	a[i-2][1]	a[i-2][2]	----	a[i-2][j-2]	a[i-2][j-1]
Row i	a[i-1][0]	a[[i-1][1]	a[i-1][2]	----	a[i-1][j-2]	a[i-1][j-1]

All the elements in a row are placed in contiguous memory location.
Consider the statement

```
        char OS [2][4] = {“DOS”,”ABC”};
```

Internally in memory, it is represented as:

D	O	S	‘\0’	A	B	C	‘\0’
OS[0,0]	OS[0,1]	OS[0,2]	OS[0,3]	OS[1,0]	OS[1,1]	OS[1,2]	OS[1,3]

Two-dimensional Array Processing

Processing of two-dimensional array is same as that of single dimensional array.
As an illustration of the use of two-dimensional arrays, consider the following program.

```
/* Program to find average marks obtained by a class of students in a test by
accepting roll number and marks of each student*/

   #include<stdio.h>
   int main (void)
   {
        int i;
        float sum=0
        int student[25][2]
        for(i=0;i<25;i++)
        {
              printf(“enter roll no and marks : “);
              scanf(“%d%d”,&student[i][0],&student[i][0]);
              /* Roll no will get stored in student[i][0] and marks
in students[i][1] */
              sum+=student[i][1];
        }
             printf(“\n Average marks:%2f\n”,sum/25);
   }
```

Fig5.1: sample code for Two-dimensional Arrays processing

What are string?

A string is one-dimensional array of character terminated by null (‘\0’) character.
Strings are used to store text information and to perform manipulations on them. Strings are declared in the same manner as other arrays.

For Example:

```
      char fruit[10];
```

Initializing Character Arrays

Character arrays can be initialized in two ways as individual characters or as a single string.

```
      char name[ ] =  {‘p’, ‘a’, ‘t’, ‘n’, ‘i’,’\0’;
```

Each character in the array occupies one byte of memory and the last character is always’0\0’, which is single character. The null character acts as a string terminator. Hence a sting of n elements can hold (n-1) characters.

```
      Char fruit[ ] = “Apple”
```

Note that, in this declaration ‘\0’ is not necessary, C interests the null character automatically, when the array is initialization with a double quoted string constant.

When initializing a character array, the length may be omitted. The compiler automatically allocates the storage depending on the length of the value given.

E.g.:

```
      char name[ ] =  “rapid”;
```

The above declaration automatically assigns storage equivalent to 6 character including ‘\0’ to the character array name.
Memory representation of above array is show in figure below:

‘r’	‘a’	‘p’	‘l’	d	\0

String Terminator

```
/* Program to accept and print a string*/

   void main (void)
   {
        char name[20];
        scanf(“%s”,name);
        printf(“%s”,name);
   }
```

The %s use in printf() is a format specification for printing out a string. The same specification can be used with scanf() also. In both cases we are supplying the base address to the functions. The scanf() function, after the enter is pressed automatically insert a ‘\0’ at the end of the string. The scanf() function is not capable of receiving multi-word strings separated by space. In that case the gets() and puts() functions.

/* Program to accept and print a string using gets and puts functions*/

```
#include<stdio.h>
#include<string.h>
main()
{
     char name[20];
     gets(name);
     puts(name);
}
```

Following are some example given using strings.

/* Program to complete the length of a given string*/

```
#include<stdio.h>
void main(void)
{
     char str[10];
     int len;
     printf(“\n enter string :”);
     scanf(“%[^\n]”, arr1);
     for(len=0;str[len]!= ‘\0’;len++)
     printf(“\n The length of the string is%d\n”,len);

}
```

Built-in String Function

The header file string.h provides useful set of string functions. These functions help in manipulating string. To use these functions, the header file string.h must be included in the program with the statement:

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

strcat (target, source)

The Strcat() function accepts two string as parameters and concatenates them. i.e. it appends the source string at the end of the target.

/* sample program using Strcat()*/

```
#include<stdio.h>
#include<string.h>
int main()
{
      char name[20];
      char name1[]=”Ash”;
      strcat(name1,name2);
      printf(“\n”);
      puts(name1);
}
Output:
      Ashwini
```

strcmp (string1, string

The function strcmp () is used to compare two strings. This function is useful while writing program for ordering or searching strings.

The function accepts two strings as parameters and return and integer value, depending up on the relative order of the two strings.

Return value	**Description**
Less than 0	If string1 is less than string2
Equal to 0	If string1 and string2 are identical
Greather than 0	If string1 is greater than string2

Table 5.1: strcmp() function return values

/* sample program to test equality of two strings using strcmp*/

```
#include<stdio.h>
#include<string.h>
int main()
{
      char str1[10];
      char str2[10];
      int result;
      print(“\n n*** Comparing two string ***\n”);
      fflush(stdin); /* flush the input buffer*/
      printf(“enter first string\n”);
      scanf(“%s”, str1);
      fflush(stdin);
      printf(“\nEnter second string\n”);
      scanf(“%s”, str2);
      result=strcmp(str1,str2);
     if (result<0)
            printf(“\nstring2 is greater than string1 …”);
     else if(result==0)
```

```
                     printf(“\nBoth the string are equal… ”)
                else
                    printf(“\nString 1 is greater than string2 …”)
         }
```

The function strcmp() compares the two strings, character by character, to decide the greater one. Whenever two characters in the string differ, the string that has the character with a higher ASCII value is greater.

e.g. consider the string hello and Hello!

The first character itself differs. The ASCII code for h is 104. While that H is 72. Since the ASCII code of h is grater, the string hello is greater than Hello!. Once a difference is found, there is no need to compare the other characters of the strings; hence, function returns the result.

strcpy (target, source)

The strcpy() function copies one string to another. This function accepts two string as parameters and copies the source string character into the target string, up to and including the null character of the source string.

```
/* sample program using strcpy() function*/

        #include<stdio.h>
        #include<string.h>
        int main()
        {
            Char name1[]=”Ash”;
            Char name2[]=”win”;
            printf(“\n**Before Copying two strings are **\v”);
            printf(“%s\t%s”,name1, name2);
            strcpy(name1,name2);
            printf(“\n**After Copying two strings are **\v”);
            printf(“%s\t%s\n”, name1, name2);
        }
   Output
            **Before copying two strings are**
                       Ash                 win
            **After copying two strings are**
                       win                 win
```

strlen (string)

The strlen() function returns an integer value, which corresponds, to the length of the string passed. The length of a string is the number of character present in it, excluding the terminating null character.

```
/* sample program using strlen() function()*/

        #include<stdio.h>
```

```
#include<string.h>
int main()
{
        char arr1[10];
        int i,len;
        printf(“\nEnter string:\n”);
        scanf(“%[^\n]”,arr1);
        printf(“\n The length of the string is %d”, strlen(arr1));
}
```

Two Dimensional Array of Character

```
     main()
    {
          char namelist[3][10]={
                                    “akshay”,
                                    “parag”,
                                    “raman”
                                    };
     }
```

Instead of initializing the, names had these names been supplied from the keyboard, the program segment would have look like this…

```
for ( i = 0; i < 3; i++ )
     scanf (“%s”, namelist [i]);
```

The memory representation of the above array is given below

1001	a	k	s	h	a	y	\0			
1011	p	a	r	a	g	\0				
1021	r	a	m	a	n	\0				

Even though 10 bytes are reserved for storting the name ‘akshay’. It occupies only7 bytes. Thus3 bytes go waste.

Standard Library String Function

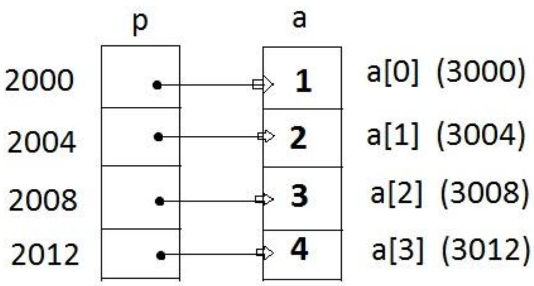
Function	Description
strlen	Find the length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at end another
strncat	Append first n character of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies a first n character of one string into another
strcmp	Compares two strings
strncmp	Compares first n character of two strings

Super! Let’s have a look at the code-

```
int a1[3]= {1,2,3};
int a2[2][3]= {{1,2,3},{1,4,5}};
int *ip = a1;
ip++;
printf(“%d\n”, *ip);
int (*ap)[3];
ap= &a1;
ap++; // it’ll move to next address not to next element, logical error
printf(“%d”, **ap);
```

Now look some more cases:

```
int a[4] = {1,2,3,4};
int *p[4] = {&a[0], &a[1], &a[2], &a[3], &a[4]};
```



‘p’ is an array of pointers.

```
int *a;
a = fun();  // int *fun(void);

int *fun(void)
{
      static int arr[] = {2,3,4};
      return arr;
}

int (*a)[4];
a = fun();   // int (*fun())[4];
int (*fun())[4]
{
      static int arr[2][4] = {{2,1,3,2}
                               {1,3,6,7}
                              };
      return arr;
}
```

```
void show(int, int);
int main()
```

```
{
      void (*fptr)(int, int);
      fptr = show();
      fptr(5,6);    // or same  (*fptr)(5,6);
      return 0;
}

void show(int a, int b)
{
      printf("%d%d",a,b);
}
```

Hopefully we come to know the difference between following-

`int *fun();   and     int(*fun)();`

Look at some twisting declarations & try to solve them your own.
Hint: know first how we can solve & distinguish `int(*fun[3])(); very well.`

int (*fptr) ();
int(*fptr)(int*, int *);
void (*fptr)();
char (*(*fptr())[3])();
int (*(* fptr)())[3][4];
int (*(*(*fptr)())[20])();
char(*(*fptr[3])())[20];
float*(*(*(*fptr)())[10])();

Similarly, declare an array of N Pointers to functions returning pointers to function returning pointers to functions returning pointers to character.

typedef

The C programming language provides a keyword called typedef, which you can use to give a type a new name. Following is an example to define a term BYTE for one-byte numbers:

```
typedef unsigned char BYTE;
```

After this type definitions, the identifier BYTE can be used as an abbreviation for the type unsigned char, for example:

```
BYTE  b1, b2;
```

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows:

```
typedef unsigned char byte;
```

typedef vs #define

The #define is a C-directive which is also used to define the aliases for various data types similar to typedef but with following differences:

The typedef is limited to giving symbolic names to types only where as #define can be used to define alias for values as well, like you can define 1 as ONE etc.

The typedef interpretation is performed by the compiler where as #define statements are processed by the pre-processor.

Try to make a plain declaration

```
typedef char*pc;
typedef pc fpc();
typedef fpc *pfpc;
typedef pfpc fpfpc();
typedef fpfpc *pfpfpc;
pfpfpc a[N];
```

Command Line Arguments

When we are passing some values from the command line to your C programs when they are executed. These values are called command line arguments and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program.

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with a single argument, it produces the following result.

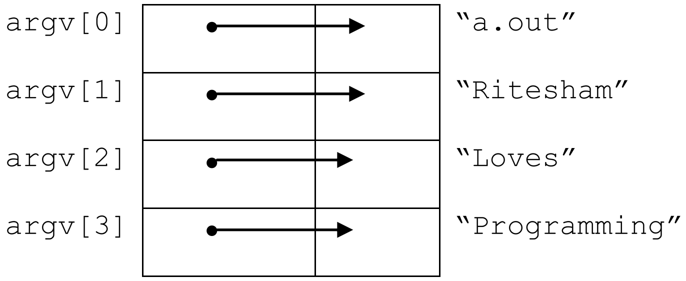
```
$./a.out testing
The argument supplied is testing
When the above code is compiled and executed with a two arguments, it produces
the following result.

$./a.out testing1 testing2
Too many arguments supplied.
When the above code is compiled and executed without passing any argument, it
produces the following result.

$./a.out
One argument expected
```

```
/* example 1 */

int
main( int number_of_args, char* list_of_args[] )
{
  if ( number_of_args != 1 )
    {
      printf("this program does not take any args!\n");
      exit(-1);
    }
}
```

```
// example 2

int
main( int number_of_args, char* list_of_args[] )
{
  char invoke_command[] = "./program_name #";
  int repeat;

  if ( number_of_args != 2 )
    {
      printf("Please use: %s\n", invoke_command);
      exit(-1);
    }

   // read the second arg and put the value in a number variable:
   sscanf(list_of_args[1], "%d", &repeat);

}
```

```
/*rit_test.c*/
int main (int argc, char* argv[])
{
      int i;
      printf(“%d\n”, argc);
      for(i=0; i<argc; i++)
      {
           printf(“%s\n”, argv[i]);
      }
}

$cc rit_test.c
$ ./a/out Ritesham Loves Programming
```

```
a.out
Ritesham
Loves
Programming
```



```
/* count lines, words, and characters in input */
#define IN  1 /* inside a word */
#define OUT 0 /* outside a word */


int main()
{
  int c, nl, nw, nc, state;

  state = OUT;
  nl = nw = nc = 0;
  while ((c = getchar()) != EOF)
      {
           ++nc;
            if (c == '\n')
             ++nl;
           if (c == ' ' || c == '\n' || c = '\t')
                state = OUT;
            else if (state == OUT)
           {
                 state = IN;
                ++nw;
           }
      }
  printf("%d %d %d\n", nl, nw, nc);
      return 0;
}
```

atoi() function

```
// A simple C program for implementation of atoi
#include <stdio.h>

// A simple atoi() function
int myAtoi(char *str)
{
    int res = 0; // Initialize result

    // Iterate through all characters of input string and update result
    for (int i = 0; str[i] != '\0'; ++i)
        res = res*10 + str[i] - '0';

    // return result.
    return res;
}

// Driver program to test above function
int main()
{
    char str[] = "89789";
    int val = myAtoi(str);
    printf ("%d ", val);
    return 0;
}
```

```
Output:

$ 89789
```

```
/* add two numbers using built-in atoi() */
int main( int argc, char *argv[])
{
      int sum =0;
      if (argc <3 || argc >3)
      {
           fprintf(stderr, "No proper input");
           exit(1);
      }
      sum = atoi(argv[1]) + atoi(argv[2]);
      printf("sum = %d\n", sum);
      return 0;
}
```

```
$ cc add.c
$ ./a.out 25 25
$ sum = 50
```

Try to find the value of EOF !

Structure

To represent the real time entity, C offers Structures. Example Library- book, pages, price, author etc.

Arrays provide the facility for grouping related data items of the same type into a single object. However, sometimes we need to group related data items of different types. An example is the inventory record of a stock items that groups together its item number, price, quantity in stock, reorder level etc. In order to handle such situations, C provides a data type, called structures, that allows a fixed number of data items, possibly of different types to be treated as a single object. It is used to group all related information into one variable.

Basics of Structures

`Structure` is a collection of logically related data items grouped together under a single name, called a structure tag.

The data items that make up a structure are called its `members` or `fields`, and can be of different types.

The general format for defining a structure is:

```
struct tag_name
{
     data_type member1;
     data_type member2;
     ....
};
```

Fig 7.1 Format for defining a structure

where,

struct	A keyword that introduces a structure definition.
Tag_name	The name of the structure.
member1, mamber2	Set of type of declarations for the member data items that make up the structure.

For example, the structure for the inventory record of a stock item may be defined as

```
struct item
{
    int itemno;
    float price;
    float quantity;
    int reorderlevel;
};
```

Consider another example, of a book database consisting of book name, author, number of pages and price.

To hold the book information, the structure can be defined as follows

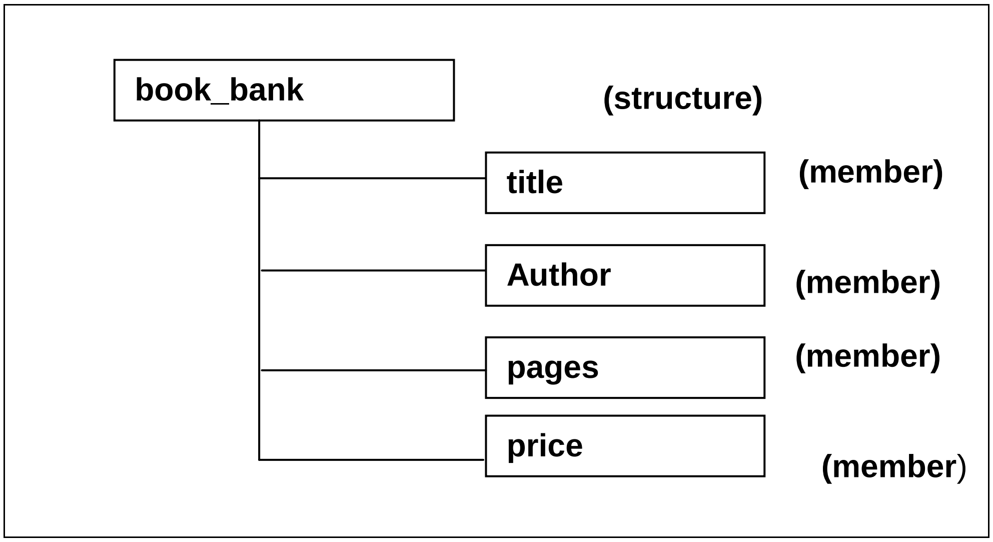
```
struct book_bank
{
     char title[15];
     char author[10];
     int pages;
     float price;
};
```

The above declaration does not declare any variables. It simply describes a format called template t o represent information as shown below:

struct book_bank

title	array of 15 characters
author	array of 10 characters
pages	integer
price	float

Following figure illustrates the composition of this book database schematically.



Structure for a book

All the members of the structure cab be of the same type, as in the following definition of the structure date.

```
struct date
{
    int day,month,year;
};
```

Declaration of Individual Members of a Structure

The individual members of a structure may be any of the common data types (such as int, float, etc.), pointers, arrays or even other structures.

All members names within a particular structure must be different. However, member names may be the same as those of the variables declared outside the structure..

Individual members cannot be initialized inside the structure declaration.

Structure Variables

A structure definition defines a new type, and variables of this type can be declared in the following ways

In the structure declaration: By including a list of variable names between the right brace and the termination semicolon in the structure definition.

For example, the declaration

```
struct student
{
    int rollno;
    char subject[10];
    float marks;
} student1, student2
```

declares `student1,student2` to be variables of type struct `student.`

If other variables of the structure are not required, the tag name student can be omitted as shown below

```
Struct
{
    int rollno;
    char name[10];
    float marks;
} student1, student2;
```

Using the structure tag

The structure tag can be thought of as the name of the type introduced by the structure definition and variables can also be declared to be of a particular structure type by a declaration of the form:

```
struct tag variable-list;
```

For example,

```
struct student student1,student2
```

declares `student1` and `student2` to be variables of type struct `student.`

Structure Initialization

A variable of particular structure type can be initialized by following its definition with an initializer for the corresponding structure type. Initializer contains intial values for components of the structure, placed within curley braces and separated by commas.

Thus, the declaration

```
struct date
{
    int day,month,year;
}independence={15,8,1947};
```

Initializes the member variables day, month and year of the structure variable independence to 15, 8 and 1947 respectively.

The declaration

```
                    struct date republic={26,1,1950}
```

initializes the member variables day, month and year of the structure variable republic to 26, 1 and 1950 respectively.

Considering the structure definition student (defined in 8.1.2), the declaration

```
        struct student student1={1,"Ashwini",98.5};
```

initializes the member variables rollno, name, and marks of the structure variable student1 to 1,"Ashwini" and 98.5 respectively.

If there are fewer initializers than that of member variables in the structure, the remaining member variables are initialized to zero.

Thus the initialization

```
struct date newyear={1,1};
```

is same as

```
struct date newyear={1,1,0};
```

Accessing Structure Members

With the help of dot operator (.) , individual elements of a structure can be accessed and the syntax is of the form

```
        structure-variable.member-name;
```

Thus to refer to name of the structure student, we can use

```
        student1.name;
```

The statements,

```
        struct date emp;            (date is defined in 8.1.3)
        emp.day=28;
        emp.month=7;
        emp.year=1969;
```

set the values of the member variables day, month and year within the variable emp to 28, 7 and 1969 respectively and the statement

```
struct date today;
 if(today.day==1&&today.month==1)
    printf(“Happy New Year”);
```

tests the values of day and month to check if both are 1 and if so, prints the message.

The elements of a structure are always stored in contiguous memory locations. It is shown below

emp.day	emp.month	emp.year
28	7	1969

Following are some example given using structures

```
/*Program to print the data using structure variable*/
#include<stdio.h>
void main(void)
{
struct date
{
     char month[15];
     int day, year;
};
struct date today;
today.day=11;


printf(“Enter Month:”);
 scanf(“%[^\n]”,today.month);
 today.year=1998;
 printf(“\nToday’s date is %d-%s-%d\n”,
    today.day, today.month,today.year);
}
```

structure definition

defining a structure variable

accessing and initializing structure

member

```
              ***str.h***
struct date
{
int month, day, year;
};
```

```
***_prog.c***
/*Program prompts the user for today’s date and prints tomorrow’s
date*/
#include<stdio.h>
#include”str.h”
void main(void)
{
     struct date today;
     struct date tomorrow;
     static int day_month[12]=
        {31,28,31,30,31,31,30,31,30,31};
     printf(“Enter Today’s date (dd:mm:yy):”);
     scaanf(“%d%d%d”,&today.day,&today.month,&today.year);
     if(today.day > day_month[today.month-1])
     {
         printf(“\n Invalid Date \n”);
         exit(0);
     }
     if(today.day!=day_month[today.month-1])
     {
          tomorrow.day=today.day+1;
          tomorrow.month=today.month;
          tomorrow.year=today.year;
     }
     else if(today.month==12)
     {
          tomorrow.day=1;
          tomorrow.month=1;
          tomorrow.year=today.year+1;
     }
     else
     {
          tomorrow.day=1;
          tomorrow.month=today.month+1;
          tomorrow.year=today.year;
     }
     printf(“\n Tomorrow’s date is %d-%d-%d\n ”,
     tomorrow.day,tomorrow.month,tomorrow.year);
```

One structure can be copied to another structure of same type directly using the assignment operator as well as element by element basis like arrays.

In this case, the values of members of a structure variable get assigned to members of another structure variable of the same type.

It is illustrated in the following example.

```
*** strdef.h ***
struct date
{
    char month[5];
    int day,year;
};
/*Example -To copy a structure to another structure*/
#include<stdio.h>
#include<string.h>
#include “strdef.h”
void main(void)
{
     struct date today={“March”,1,98};
     struct date day1, day2;
/* copying element by element basis*/
     strcpy(day1.month, today.month);
     day1.day=today.day;
     day1.year=today.year;
/* copying entire structure to another structure */
     day2=day1;
     printf(“\n Date is %d %s %d \n”,
           today.day, today.month, today.year);
     printf(“\nDate is %d %s %d\n”,
           day1.day,day1.month,day1.year);
     printf(“\n Date is %d %s %d \n”,
           day2.day,day2.month,day2.year);
}
```

accessing structure

defined in strdef.h

Nested Structure

The individual member of a structure can be other structure as well. It is termed as `Nested Structures`. We will include a new member date which itself is a structure. It can be done in two ways.

The first way is by declaring

```
struct date
{
     int day,month,year;
};
struct emp
{
     char name[15];
     struct date birthday;
     float salary;
};
```

The embedded structure date must be declared before its use within the containing structure.

The second way is by declaring

```
struct emp
{
    char name[15];
    struct date
    {
        int day,month,year;
    }birthday;
    float salary;
};
```

In this method , we combine the two structure declarations. The embedded structure `date` is defined within enclosing structure definition.

In the first case, where the `date` structure is declared outside the `emp` structure, it can be used directly in other places, as an ordinary structure. This is not possible insecond case.

Variables of a nested structure type can be defined as usual. They may also be initialized at that time of declaration as

```
struct emp
{
    char name[15];
    struct date
    {
        int day,month,year;
    }birthday;
    float salary;
}person = {“Ashwini”,{28,7,1969},5000.65};
```

The inner pair of braces is optional.

A particular member inside a nested structure can be accessed by repeatedly applying the dot operator. Thus the statement

```
          person.birthday.day=28;
```

sets the day variable in the birthday structure within person to 28.

The statement

```
printf(“%d%d%d”,person.birthday.day,person.birthday.month,
Person.birthday.year);
```

prints date of birth of a person.

However, a structure cannot be nested within itself.

Structures and Arrays

Arrays and structure can be freely intermixed to create arrays of structure, structures containg arrays.

Arrays of Structures

In the array of structures array contains individual structures as its elements. These are commonly used when a large number of similar records are required to be processed together.

For example, the data of motor containing 1000 parts can be organized in an array of structure as

```
struct item motor[1000];
```

This statement declares `motor` to be an array containing 1000 items of the type struct `item`.

An array of structures can be declared in two ways as illustrated below.

The first way is by declaring

```
struct person
{
    char name[10];
    struct date birthday;
    float salary;
}emprec[15];
```

In this case, `emprec` is an array of 15 person structures. Each element of the array emprec will contain the structure of type person. The person structure consists of 3 individual members : an array name, salary and another structure date.

The embedded structure date must be declared before its use within the containing structure.

The second approach to the same problem involves the use of the structure tag as below.

```
struct person
{
     char name [10];
     struct date birthday;
     float salary;
};
struct person emprec[15]
```

Following program explains how to use an array of structures.

```
/*Example - An array of structures*/
#include<stdio.h>
void main(void)
{
   struct book
   {
      char name[15];
      int pages;
      float price;
   };
   struct book b[10];
   int i;
   printf(“\n Enter name,pages and price of the book\n”);
/*accessing elements of array of structures*/
    for(i=0;i<9;i++)
    {
         scanf(“%s%d%f”,b[i].name,&b[i].pages,&b[i].price);
         printf(“\n”);
    }
    printf(“\n Name,Pages and Price of the book:\n”);
    for(i=0;i<=9;i++)
    {
         printf(“%s%d%f”,b[i].name,b[i].pages,b[i].price);
    }
}
```

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

Packing several objects into a machine word. e.g. 1 bit flags can be compacted.Reading external file formats -- non-standard file formats could be read in. E.g. 9 bit integers.

C allows us do this in a structure definition by putting :bit length after the variable. For example:

```
struct packed_struct {
  unsigned int f1:1;
  unsigned int f2:1;
  unsigned int f3:1;
  unsigned int f4:1;
  unsigned int type:4;
  unsigned int my_int:9;
} pack;
```

Here, the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4 bit type and a 9 bit my_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word.

Structure Padding

n order to align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called structure padding.
Architecture of a computer processor is such a way that it can read 1 word (4 byte in 32 bit processor) from memory at a time.
To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member's address.
Because of this structure padding concept in C, size of the structure is always not same as what we think.
For example, please consider below structure that has 5 members.

```
..
struct student
{
       int id1;
       int id2;
       char a;
       char b;
       float percentage;
};
..
```

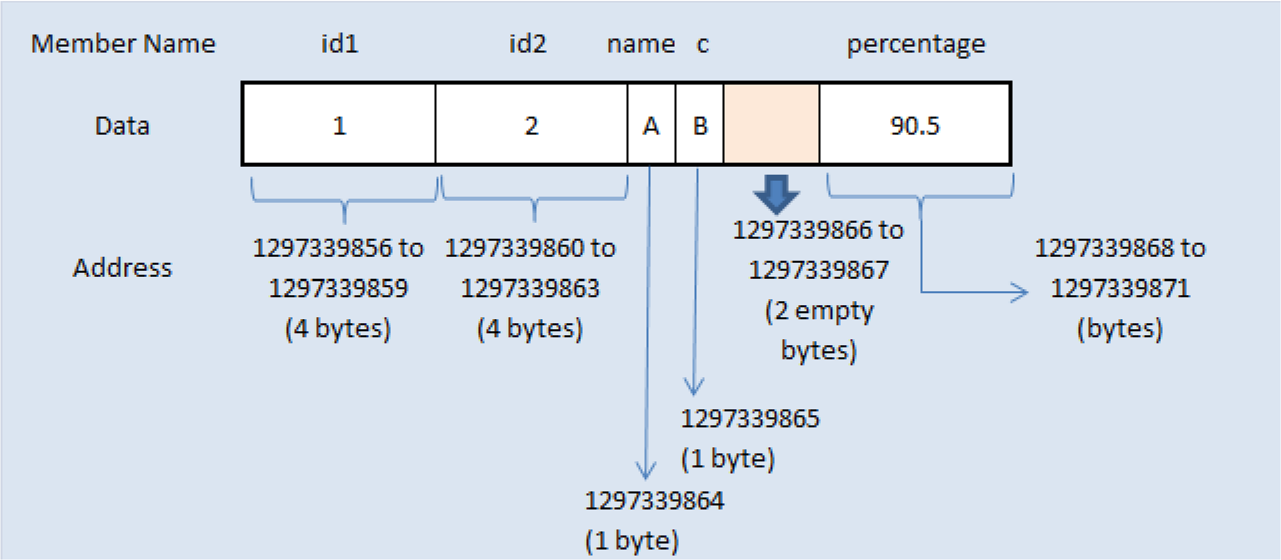
As per C concepts, int and float datatypes occupy 4 bytes each and char datatype occupies 1 byte for 32 bit processor. So, only 14 bytes (4+4+1+1+4) should be allocated for above structure.
But, this is wrong. Do you know why?
Architecture of a computer processor is such a way that it can read 1 word from memory at a time.
1 word is equal to 4 bytes for 32 bit processor and 8 bytes for 64 bit processor. So, 32 bit processor always reads 4 bytes at a time and 64 bit processor always reads 8 bytes at a time.
This concept is very useful to increase the processor speed.
To make use of this advantage, memory is arranged as a group of 4 bytes in 32 bit processor and 8 bytes in 64 bit processor.
Below C program is compiled and executed in 32 bit compiler. Please check memory allocated for structure1 and structure2 in below program.
Example program for structure padding in C:

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

/*  Below structure1 and structure2 are same.
    They differ only in member's allignment */

struct structure1
{
       int id1;
       int id2;
       char name;
       char c;
       float percentage;
};
struct structure2
{
       int id1;
       char name;
       int id2;
       char c;
       float percentage;
};

int main()
{
    struct structure1 a;
    struct structure2 b;

    printf("size of structure1 in bytes : %d\n",
            sizeof(a));
    printf ( "\n   Address of id1        = %u", &a.id1 );
    printf ( "\n   Address of id2        = %u", &a.id2 );
    printf ( "\n   Address of name       = %u", &a.name );
    printf ( "\n   Address of c          = %u", &a.c );
    printf ( "\n   Address of percentage = %u",
                   &a.percentage );

    printf("   \n\nsize of structure2 in bytes : %d\n",
                   sizeof(b));
    printf ( "\n   Address of id1        = %u", &b.id1 );
    printf ( "\n   Address of name       = %u", &b.name );
    printf ( "\n   Address of id2        = %u", &b.id2 );
    printf ( "\n   Address of c          = %u", &b.c );
    printf ( "\n   Address of percentage = %u",
                   &b.percentage );
    getchar();
    return 0;
}
```

```
Output:

size of structure1 in bytes : 16
Address of id1 = 1297339856
Address of id2 = 1297339860
Address of name = 1297339864
Address of c = 1297339865
Address of percentage = 1297339868

size of structure2 in bytes : 20

Address of id1 = 1297339824
Address of name = 1297339828
Address of id2 = 1297339832
Address of c = 1297339836
Address of percentage = 1297339840
```

Structure padding analysis for above C program:

Memory allocation for structure1:


Member Name
id1
id2
name
c
percentage
Data
1
2
A
B
90.5
Address
1297339856 to 1297339859 (4 bytes)
1297339860 to 1297339863 (4 bytes)
1297339866 to 1297339867 (2 empty bytes)
1297339868 to 1297339871 (bytes)
1297339865 (1 byte)
1297339864 (1 byte)

In above program, memory for structure1 is allocated sequentially for first 4 members. Whereas, memory for 5th member "percentage" is not allocated immediate next to the end of member "c".

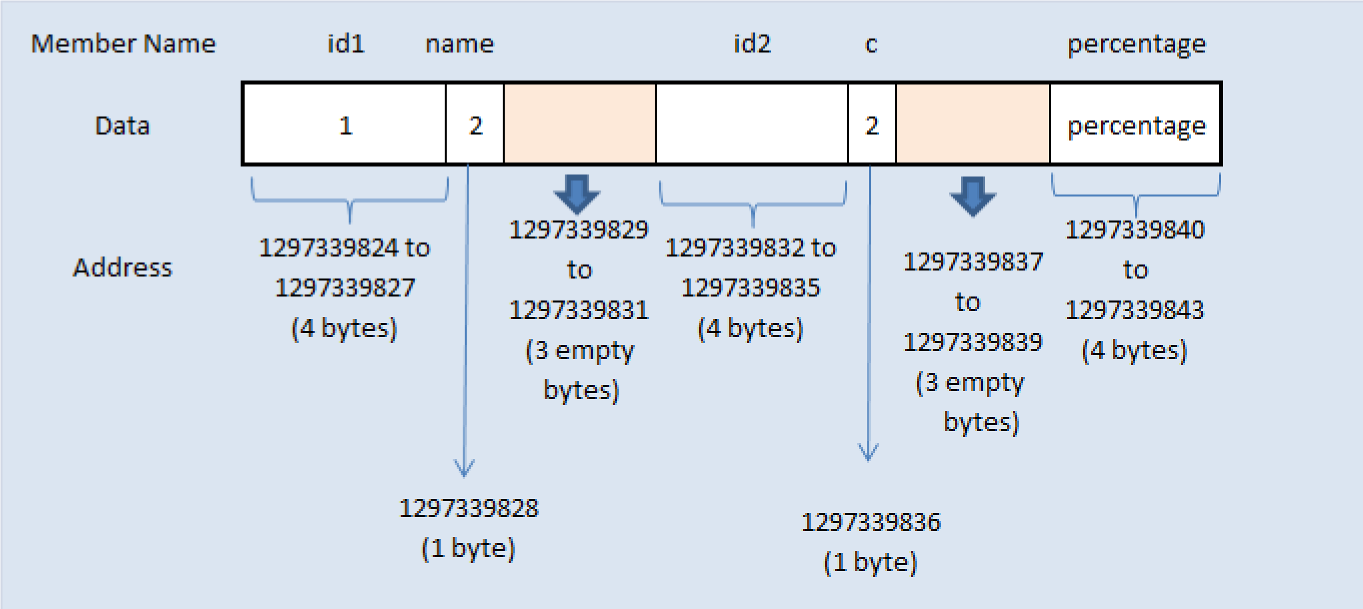
There are only 2 bytes remaining in the package of 4 bytes after memory allocated to member "c". Range of this 4 byte package is from 1297339864 to 1297339867.

Addresses 1297339864 and 1297339865 are used for members "name and c". Addresses 1297339866 and 1297339867 only is available in this package.

But, member "percentage" is datatype of float and requires 4 bytes. It can't be stored in the same memory package as it requires 4 bytes. Only 2 bytes are free in that package.
So, next 4 byte of memory package is chosen to store percentage data which is from 1297339868 to 1297339871.
Because of this, memory 1297339866 and 1297339867 are not used by the program and those 2 bytes are left empty.

So, size of structure1 is 16 bytes which is 2 bytes extra than what we think. Because, 2 bytes are left empty.

Memory allocation for structure2:


Member Name
id1
name
id2
c
percentage
Data
1
2
2
percentage
Address
1297339824 to 1297339827 (4 bytes)
1297339829 to 1297339831 (3 empty bytes)
1297339832 to 1297339835 (4 bytes)
1297339837 to 1297339839 (3 empty bytes)
1297339840 to 1297339843 (4 bytes)
1297339828 (1 byte)
1297339836 (1 byte)

Memory for structure2 is also allocated as same as above concept. Please note that structure1 and structure2 are same. But, they differ only in the order of the members declared inside the structure.
4 bytes of memory is allocated for 1st structure member “id1″ which occupies whole 4 byte of memory package.

Then, 2nd structure member “name” occupies only 1 byte of memory in next 4 byte package and remaining 3 bytes are left empty. Because, 3rd structure member “id2″ of datatype integer requires whole 4 byte of memory in the package. But, this is not possible as only 3 bytes available in the package.

So, next whole 4 byte package is used for structure member “id2″.
Again, 4th structure member “c” occupies only 1 byte of memory in next 4 byte package and remaining 3 bytes are left empty.

Because, 5th structure member “percentage” of datatype float requires whole 4 byte of memory in the package.

But, this is also not possible as only 3 bytes available in the package. So, next whole 4 byte package is used for structure member “percentage”.
S
o, size of structure2 is 20 bytes which is 6 bytes extra than what we think. Because, 6 bytes are left empty.

How to avoid structure padding in C?

#pragma pack (1) directive can be used for arranging memory for structure members very next to the end of other structure members.
VC++ supports this feature. But, some compilers such as Turbo C/C++ does not support this feature.
Please check the below program where there will be no addresses (bytes) left empty because of structure padding.
Example program to avoid structure padding in C:

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

/* Below structure1 and structure2 are same.
    They differ only in member's allignment */

#pragma pack(1)
struct structure1
{
       int id1;
       int id2;
       char name;
       char c;
       float percentage;
};

struct structure2
{
       int id1;
       char name;
       int id2;
       char c;
       float percentage;
};

int main()
{
    struct structure1 a;
    struct structure2 b;

    printf("size of structure1 in bytes : %d\n",
                   sizeof(a));
    printf ( "\n   Address of id1        = %u", &a.id1 );
    printf ( "\n   Address of id2        = %u", &a.id2 );
    printf ( "\n   Address of name       = %u", &a.name );
    printf ( "\n   Address of c          = %u", &a.c );
    printf ( "\n   Address of percentage = %u",
                   &a.percentage );

    printf("   \n\nsize of structure2 in bytes : %d\n",
                   sizeof(b));
    printf ( "\n   Address of id1         = %u", &b.id1 );
    printf ( "\n   Address of name        = %u", &b.name );
    printf ( "\n   Address of id2         = %u", &b.id2 );
    printf ( "\n   Address of c           = %u", &b.c );
    printf ( "\n   Address of percentage = %u",
                   &b.percentage );
    getchar();
    return 0;
}
```

```
Output:

size of structure1 in bytes : 14
Address of id1 = 3438103088
Address of id2 = 3438103092
Address of name = 3438103096
Address of c = 3438103097
Address of percentage = 3438103098

size of structure2 in bytes : 14

Address of id1 = 3438103072
Address of name = 3438103076
Address of id2 = 3438103077
Address of c = 3438103081
Address of percentage = 3438103082
```

C Programming Structure and Function

In C, structure can be passed to functions by two methods:

Passing by value (passing actual value as argument)
Passing by reference (passing address of an argument)

Passing structure by value

```
#include <stdio.h>
struct student{
    char name[50];
    int roll;
};
void Display(struct student stu);
/* function prototype should be below to the structure declaration otherwise
compiler shows error */
int main(){
    struct student s1;
    printf("Enter student's name: ");
    scanf("%s",&s1.name);
    printf("Enter roll number:");
    scanf("%d",&s1.roll);
    Display(s1);   // passing structure variable s1 as argument
    return 0;
}
void Display(struct student stu){
  printf("Output\nName: %s",stu.name);
  printf("\nRoll: %d",stu.roll);
}
```

Passing structure by reference

```
/* C program to add two distances(feet-inch system) entered by user. To solve
this program, make a structure. Pass two structure variable (containing distance
in feet and inch) to add function by reference and display the result in main
function without returning it.
*/

#include <stdio.h>
struct distance{
    int feet;
    float inch;
};
void Add(struct distance d1,struct distance d2, struct distance *d3);
int main()
{
    struct distance dist1, dist2, dist3;
    printf("First distance\n");
    printf("Enter feet: ");
    scanf("%d",&dist1.feet);
    printf("Enter inch: ");
    scanf("%f",&dist1.inch);
    printf("Second distance\n");
    printf("Enter feet: ");
    scanf("%d",&dist2.feet);
    printf("Enter inch: ");
    scanf("%f",&dist2.inch);
    Add(dist1, dist2, &dist3);

/*passing structure variables dist1 and dist2 by value whereas passing structure
variable dist3 by reference */
    printf("\nSum of distances = %d\'-%.1f\"",dist3.feet, dist3.inch);
    return 0;
}
void Add(struct distance d1,struct distance d2, struct distance *d3)
{
/* Adding distances d1 and d2 and storing it in d3 */
     d3->feet=d1.feet+d2.feet;
     d3->inch=d1.inch+d2.inch;
     if (d3->inch>=12) {     /* if inch is greater or equal to 12, converting it
to feet. */
         d3->inch-=12;
         ++d3->feet;
    }
}
```

Unions

Unions are quite similar to the structures in C. Union is also a derived type as structure. Union can be defined in same manner as structures just the keyword used in defining union in union where keyword used in defining structure was struct.

```
union car
{
  char name[50];
  int price;
};

union car
{
  char name[50];
  int price;
}c1, c2, *c3;

OR;

union car
{
  char name[50];
  int price;
};

-------Inside Function-----------
union car c1, c2, *c3;

Difference between union and structure
#include <stdio.h>
union job
{          //defining a union
   char name[32];
   float salary;
   int worker_no;
}u;
struct job1
{
   char name[32];
   float salary;
   int worker_no;
}s;
int main()
{
   printf("size of union = %d",sizeof(u));
   printf("\nsize of structure = %d", sizeof(s));
   return 0;
}
size of union = 32
size of structure = 40
```

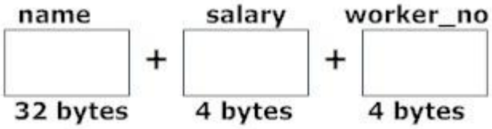


Fig: Memory allocation in case of structure

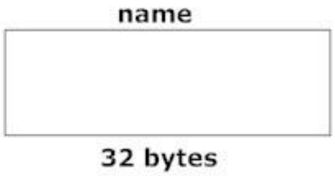


Fig: Memory allocation in case of union

Checking Little Endien or Big Endien Machine

```
int x = 1;
        if(*(char *)&x == 1)
                printf("little-endian\n");
        else    printf("big-endian\n");
```

or a union:

```
        union {
                int i;
                char c[sizeof(int)];
        } x;
        x.i = 1;
        if(x.c[0] == 1)
                printf("little-endian\n");
        else    printf("big-endian\n");
```

Type Qualifiers

C – type qualifiers : The keywords which are used to modify the properties of a variable are called type qualifiers.

Types of C type qualifiers:

There are two types of qualifiers available in C language. They are,

const
volatile

1. const keyword:
[Stored in ROM]
Constants are also like normal variables. But, only difference is, their values can’t be modified by the program once they are defined. They refer to fixed values. They are also called as literals.

They may be belonging to any of the data type.
Syntax:

```
const data_type variable_name; (or) const data_type *variable_name;
```

Types of C constant:

Integer constants
Real or Floating point constants
Octal & Hexadecimal constants
Character constants
String constants
Backslash character constants

Rules for constructing C constant:

1. Integer Constants in C:

An integer constant must have at least one digit.
It must not have a decimal point.
It can either be positive or negative.
No commas or blanks are allowed within an integer constant.
If no sign precedes an integer constant, it is assumed to be positive.
The allowable range for integer constants is -32768 to 32767.

2. Real constants in C:

A real constant must have at least one digit
It must have a decimal point
It could be either positive or negative
If no sign precedes an integer constant, it is assumed to be positive.
No commas or blanks are allowed within a real constant.

3. Character and string constants in C:

A character constant is a single alphabet, a single digit or a single special symbol enclosed within single quotes.
The maximum length of a character constant is 1 character.
String constants are enclosed within double quotes.
4. Backslash Character Constants in C:

There are some characters which have special meaning in C language.
They should be preceded by backslash symbol to make use of special function of them.
Given below is the list of special characters and their purpose.

Backslash_character	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\”	Double quote
\’	Single quote
\\	Backslash
\v	Vertical tab
\a	Alert or bell
\?	Question mark
\N	Octal constant (N is an octal constant)
\XN	Hexadecimal constant (N – hex.dcml cnst)

How to use constants in a C program?

We can define constants in a C program in the following ways.
By “const” keyword
By “#define” preprocessor directive
Please note that when you try to change constant values after defining in C program, it will through error.

Example program using const keyword in C:

```
#include <stdio.h>

void main()
{
   const int  height = 100;                /*int constant*/
   const float number = 3.14;              /*Real constant*/
   const char letter = 'A';                /*char constant*/
   const char letter_sequence[10] = "ABC"; /*string constant*/
   const char backslash_char = '\?';      /*special char cnst*/

   printf("value of height    : %d \n", height );
   printf("value of number : %f \n", number );
   printf("value of letter : %c \n", letter );
   printf("value of letter_sequence : %s \n", letter_sequence);
   printf("value of backslash_char  : %c \n", backslash_char);
}
```

```
Output:
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

2. Example program using #define preprocessor directive in C:

```
#include <stdio.h>

#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'

void main()
{

   printf("value of height    : %d \n", height );
   printf("value of number : %f \n", number );
   printf("value of letter : %c \n", letter );
   printf("value of letter_sequence : %s \n", letter_sequence);
   printf("value of backslash_char  : %c \n", backslash_char);

}
```

```
Output:
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

2. **Volatile keyword:**

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time. The system always reads the current value of a volatile object from the memory location rather than keeping its value in temporary register at the point it is requested, even if a previous instruction asked for a value from the same object. So the simple question is, how can value of a variable change in such a way that compiler cannot predict. Consider the following cases for answer to this question.

1) Global variables modified by an interrupt service routine outside the scope: For example, a global variable can represent a data port (usually global pointer referred as memory mapped IO) which will be updated dynamically. The code reading data port must be declared as volatile in order to fetch latest data available at the port. Failing to declare variable as volatile, the compiler will optimize the code in such a way that it will read the port only once and keeps using the same value in a temporary register to speed up the program (speed optimization). In general, an ISR used to update these data port when there is an interrupt due to availability of new data

2) Global variables within a multi-threaded application: There are multiple ways for threads communication, viz, message passing, shared memory, mail boxes, etc. A global variable is weak form of shared memory. When two threads sharing information via global variable, they need to be qualified with volatile. Since threads run asynchronously, any update of global variable due to one thread should be fetched freshly by another consumer thread. Compiler can read the global variable and can place them in temporary variable of current thread context. To nullify the effect of compiler optimizations, such global variables to be qualified as volatile

If we do not use volatile qualifier, the following problems may arise
1) Code may not work as expected when optimization is turned on.
2) Code may not work as expected when interrupts are enabled and used.

Let us see an example to understand how compilers interpret volatile keyword. Consider below code, we are changing value of const object using pointer and we are compiling code without optimization option. Hence compiler won't do any optimization and will change value of const object.

```
/* Compile code without optimization option */
#include <stdio.h>
int main(void)
{
    const int local = 10;
    int *ptr = (int*) &local;
    printf("Initial value of local : %d \n", local);
    *ptr = 100;
    printf("Modified value of local: %d \n", local);
    return 0;
}
```

When we compile code with “–save-temps” option of gcc it generates 3 output files

1) preprocessed code (having .i extention)
2) assembly code (having .s extention) and
3) object code (having .o option).

We compile code without optimization, that’s why the size of assembly code will be larger (which is highlighted in red color below).

Output:

```
$ gcc volatile.c -o volatile -save-temps
$ ./volatile
Initial value of local : 10
Modified value of local: 100
$ ls -l volatile.s
-rw-r-r- 1 rapidcode rapidcode 731 2016-11-19 16:19 volatile.s
$
```

Let us compile same code with optimization option (i.e. -O option). In thr below code, “local” is declared as const (and non-volatile), GCC compiler does optimization and ignores the instructions which try to change value of const object. Hence value of const object remains same.

```
/* Compile code with optimization option */
#include <stdio.h>

int main(void)
{
    const int local = 10;
    int *ptr = (int*) &local;

    printf("Initial value of local : %d \n", local);

    *ptr = 100;

    printf("Modified value of local: %d \n", local);

    return 0;
}
```

For above code, compiler does optimization, that’s why the size of assembly code will reduce.

```
Output:
```

```
$ gcc -O3 volatile.c -o volatile -save-temps
$ ./volatile
Initial value of local : 10
Modified value of local: 10
$ ls -l volatile.s
-rw-r-r- 1 rapidcode rapidcode 626 2016-11-19 16:21 volatile.s
```

Let us declare const object as volatile and compile code with optimization option. Although we compile code with optimization option, value of const object will change, because variable is declared as volatile that means don’t do any optimization.

```
/* Compile code with optimization option */
#include <stdio.h>

int main(void)
{
    const volatile int local = 10;
    int *ptr = (int*) &local;

    printf("Initial value of local : %d \n", local);

    *ptr = 100;

    printf("Modified value of local: %d \n", local);

    return 0;
}
```

```
Output:

  $ gcc -O3 volatile.c -o volatile –save-temp
  $ ./volatile
  Initial value of local : 10
  Modified value of local: 100
  $ ls -l volatile.s
  -rw-r–r– 1 rapidcode rapidcode 711 2016-11-19 16:22 volatile.s
  $
```

The above example may not be a good practical example, the purpose was to explain how compilers interpret volatile keyword. As a practical example, think of touch sensor on mobile phones. The driver abstracting touch sensor will read the location of touch and send it to higher level applications. The driver itself should not modify (const-ness) the read location, and make sure it reads the touch input every time fresh (volatile-ness). Such driver must read the touch sensor input in const volatile manner.

Typedef

Typedef is a keyword that is used to give a new symbolic name for the existing name in a C program. This is same like defining alias for the commands.
Consider the below structure.

```
struct student
{
         int mark [2];
         char name [10];
         float average;
}
```

Variable for the above structure can be declared in two ways.

1st way :

```
struct student record;       /* for normal variable */
struct student *record;     /* for pointer variable */
```

2nd way :

```
typedef struct student status;
```

When we use “typedef” keyword before struct <tag_name> like above, after that we can simply use type definition “status” in the C program to declare structure variable.
Now, structure variable declaration will be, “status record”.
This is equal to “struct student record”. Type definition for “struct student” is status. i.e. status = “struct student”
An alternative way for structure declaration using typedef in C:

```
typedef struct student
{
         int mark [2];
         char name [10];
         float average;
} status;

To declare structure variable, we can use the below statements.
status record1;                 /* record 1 is structure variable */
status record2;                 /* record 2 is structure variable */
```

Example program for C typedef:

```
// Structure using typedef:

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

typedef struct student
{
  int id;
  char name[20];
  float percentage;
} status;

int main()
{
  status record;
  record.id=1;
  strcpy(record.name, "Rapidcode");
  record.percentage = 86.5;
  printf(" Id is: %d \n", record.id);
  printf(" Name is: %s \n", record.name);
  printf(" Percentage is: %f \n", record.percentage);
  return 0;
}
```

Output:

```
Id is: 1
Name is: Rapidcode
Percentage is: 86.500000
```

Typedef can be used to simplify the real commands as per our need.
For example, consider below statement.

```
typedef long long int LLI;
```

In above statement, LLI is the type definition for the real C command “long long int”. We can use type definition LLI instead of using full command “long long int” in a C program once it is defined.

Another example program for C typedef:

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

int main()
{
   typedef long long int LLI;

   printf("Storage size for long long int data " \
          "type  : %ld \n", sizeof(LLI));

   return 0;
}
```

```
Output:

Storage size for long long int data type : 8
```

Writing a ``varargs'' Function

In ANSI C, the head of a varargs function looks just like its prototype. We will illustrate by writing our own, stripped-down version of printf. The bare outline of the function definition will look like this:

```
      void myprintf(const char *fmt, ...)
      {
      }
```

printf's job, of course, is to print its format string while looking for % characters and treating them specially. So the main loop of printf will look like this:

```
#include <stdio.h>

      void myprintf(const char *fmt, ...)
      {
      const char *p;

      for(p = fmt; *p != '\0'; p++)
            {
            if(*p != '%')
                  putchar(*p);
            else  {
                  handle it specially
                  }
            }
      }
```

In this stripped-down version, we won't worry about width and precision specifiers and other modifiers; we'll always look at the very next character after the % and assume that it's the primary format character. Continuing to flesh out our outline, we get this:

```
#include <stdio.h>

      void myprintf(const char *fmt, ...)
      {
      const char *p;

      for(p = fmt; *p != '\0'; p++)
            {
            if(*p != '%')
                  {
                  putchar(*p);
                  continue;
                  }

            switch(*++p)
                  {
                  case 'c':
                        fetch and print a character
                        break;
                  case 'd':
                        fetch and print an integer
                        break;

                  case 's':
                        fetch and print a string
                        break;

                  case 'x':
                        print an integer, in hexadecimal
                        break;

                  case '%':
                        print a single %
                        break;
                  }
            }
      }
```

The famous `“int printf(const char*fmt, …);”` uses the VARARGS
There are certain macros defined in `<stdagg.h>`

```
va_list // declare variable to refer to each argument
va_start  // initialize the va_list variable to the first unnamed argument
va_arg  // returns one argument; increment va_list variable to the next
av_end // cleans up everything
```

```
#include<stdard.h>
#include<stdio.h>
int main()
{
      Var_Arg (1,2,3,4,5,5);
      Var_Arg (3,4,5,5);
      Var_Arg (1,2,3,4,5,5,6,7,8);
      return 0;
}
void Va_Arg( int Tot, …)
{
      int i;
      Va_List args;
      Va_start (args, Tot);
      int Num = 0;
      for (i = 0; i<Tot; i++)
            printf(“%d\n”, Va_arg(args, int));
      Va_End(args);

}
```

Input & Output

When we are saying Input that means to feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

When we are saying Output that means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output the data on the computer screen as well as you can save that data in text or binary files.

The Standard Files

C programming language treats all the devices as files. So devices such as the display are addressed in the same way as files and following three file are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

The file points are the means to access the file for reading and writing purpose. This section will explain you how to read values from the screen and how to print the result on the screen.

The getchar() & putchar() functions

The int getchar(void) function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one characters from the screen.

The int putchar(int c) function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen. Check the following example:

```
#include <stdio.h>
int main( )
{
   int c;

   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );

   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text when you enter a text and press enter then program proceeds and reads only a single character and displays it as follows:

```
$./a.out
Enter a value : this is test
You entered: t
```

The gets() & puts() functions

The char *gets(char *s) function reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF.

The int puts(const char *s) function writes the string s and a trailing newline to stdout.

```
#include <stdio.h>
int main( )
{
   char str[100];

   printf( "Enter a value :");
   gets( str );

   printf( "\nYou entered: ");
   puts( str );

   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text when you enter a text and press enter then program proceeds and reads the complete line till end and displays it as follows:

```
$./a.out
Enter a value : this is test
You entered: This is test
```

The scanf() and printf() functions

The int scanf(const char *format, ...) function reads input from the standard input stream stdin and scans that input according to format provided.

The int printf(const char *format, ...) function writes output to the standard output stream stdout and produces output according to a format provided.

The format can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. For a complete detail you can refer to a man page for these function. For now let us proceed with a simple example which makes things clear:

```
#include <stdio.h>
int main( )
{
   char str[100];
   int i;

   printf( "Enter a value :");
   scanf("%s %d", str, &i);

   printf( "\nYou entered: %s %d ", str, i);

   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text when you enter a text and press enter then program proceeds and reads the input and displays it as follows:

```
$./a.out
Enter a value : seven 7
You entered: seven 7
```

Here, it should be noted that scanf() expect input in the same format as you provided %s and %d, which means you have to provide valid input like "string integer", if you provide "string string" or "integer integer" then it will be assumed as wrong input. Second, while reading a string scanf() stops reading as soon as it encounters a space so "this is test" are three strings for scanf().

A file represents a sequence of bytes, does not matter if it is a text file or binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through important calls for the file management.

Opening Files

You can use the fopen( ) function to create a new file or to open an existing file, this call will initialize an object of the type FILE, which contains all the information necessary to control the stream. Following is the prototype of this function call:

```
FILE *fopen( const char * filename, const char * mode );
```

Here, filename is string literal, which you will use to name your file and access mode can have one of the following values:

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for reading and writing both.
w+	Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
a+	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is:

```
int fclose( FILE *fp );
```

The fclose( ) function returns zero on success, or EOF if there is an error in closing the file. This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file stdio.h.

There are various functions provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.

Writing a File

Following is the simplest function to write individual characters to a stream:

```
int fputc( int c, FILE *fp );
```

The function fputc() writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise EOF if there is an error. You can use the following functions to write a null-terminated string to a stream:

```
int fputs( const char *s, FILE *fp );
```

The function fputs() writes the string s to the output stream referenced by fp. It returns a non-negative value on success, otherwise EOF is returned in case of any error. You can use int fprintf(FILE *fp,const char *format, ...) function as well to write a string into a file. Try the following example:

Make sure you have /tmp directory available, if its not then before proceeding, you must create this directory on your machine.

```
#include <stdio.h>

main()
{
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file test.txt in /tmp directory and writes two lines using two different functions. Let us read this file in next section.

Reading a File

Following is the simplest function to read a single character from a file:

```
int fgetc( FILE * fp );
```

The fgetc() function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error it returns EOF. The following functions allow you to read a string from a stream:

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions fgets() reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer buf, appending a null character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character. You can also use int fscanf(FILE *fp, const char *format, ...) function to read strings from a file but it stops reading after the first space character encounters.

```
#include <stdio.h>

int main()
{
   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);
}
```

When the above code is compiled and executed, it reads the file created in previous section and produces the following result:

```
1 : This
2: is testing for fprintf...
3: This is testing for fputs...
```

Let's see a little more detail about what happened here. First fscanf() method read just This because after that it encountered a space, second call is for fgets() which read the remaining line till it encountered end of line. Finally last call fgets() read second line completely.

Binary I/O Functions

There are following two functions, which can be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

Some General Example Programs

Writing to a file

```
#include <stdio.h>
int main()
{
   int n;
   FILE *fptr;
   fptr=fopen("C:\\program.txt","w");
   if(fptr==NULL){
      printf("Error!");
      exit(1);
   }
   printf("Enter n: ");
   scanf("%d",&n);
   fprintf(fptr,"%d",n);
   fclose(fptr);
   return 0;
}
```

Reading from file

```
#include <stdio.h>
int main()
{
   int n;
   FILE *fptr;
   if ((fptr=fopen("C:\\program.txt","r"))==NULL){
       printf("Error! opening file");
       exit(1);         /* Program exits if file pointer returns NULL. */
   }
   fscanf(fptr,"%d",&n);
   printf("Value of n=%d",n);
   fclose(fptr);
   return 0;
}
```

```
/*C program to read name and marks of n number of students from user and store
them in a file*/

#include <stdio.h>
int main(){
   char name[50];
   int marks,i,n;
   printf("Enter number of students: ");
   scanf("%d",&n);
   FILE *fptr;
   fptr=(fopen("C:\\student.txt","w"));
   if(fptr==NULL){
       printf("Error!");
       exit(1);
   }
   for(i=0;i<n;++i)
   {
      printf("For student%d\nEnter name: ",i+1);
      scanf("%s",name);
      printf("Enter marks: ");
      scanf("%d",&marks);
      fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
   }
   fclose(fptr);
   return 0;
}
```

```
/*C program to read name and marks of n number of students from user and store
them in a file. If the file previously exits, add the information of n
students.*/

#include <stdio.h>
int main(){
   char name[50];
   int marks,i,n;
   printf("Enter number of students: ");
   scanf("%d",&n);
   FILE *fptr;
   fptr=(fopen("C:\\student.txt","a"));
   if(fptr==NULL){
       printf("Error!");
       exit(1);
   }
   for(i=0;i<n;++i)
   {
      printf("For student%d\nEnter name: ",i+1);
      scanf("%s",name);
      printf("Enter marks: ");
      scanf("%d",&marks);
      fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
   }
   fclose(fptr);
   return 0;
}
```

```
/*C program to write all the members of an array of strcures to a file using
fwrite(). Read the array from the file and display on the screen.*/

#include <stdio.h>
struct s
{
char name[50];
int height;
};
int main(){
    struct s a[5],b[5];
    FILE *fptr;
    int i;
    fptr=fopen("file.txt","wb");
    for(i=0;i<5;++i)
    {
        fflush(stdin);
        printf("Enter name: ");
        gets(a[i].name);
        printf("Enter height: ");
        scanf("%d",&a[i].height);
    }
    fwrite(a,sizeof(a),1,fptr);
    fclose(fptr);
    fptr=fopen("file.txt","rb");
    fread(b,sizeof(b),1,fptr);
    for(i=0;i<5;++i)
    {
        printf("Name: %s\nHeight: %d",b[i].name,b[i].height);
    }
    fclose(fptr);
}
```

Preprocessors

C Preprocessor is not part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP.

All preprocessor commands begin with a pound symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column. Following section lists down all important preprocessor directives:

Directive	Description
`#define`	`Substitutes a preprocessor macro`
`#include`	`Inserts a particular header from another file`
`#undef`	`Undefines a preprocessor macro`
`#ifdef`	`Returns true if this macro is defined`
`#ifndef`	`Returns true if this macro is not defined`
`#if`	`Tests if a compile time condition is true`
`#else`	`The alternative for #if`
`#elif`	`#else an #if in one statement`
`#endif`	`Ends preprocessor conditional`
`#error`	`Prints error message on stderr`
`#pragma`	`Issues special commands to the compiler, using a standardized method`

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use #define for constants to increase readability.

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

These directives tell the CPP to get stdio.h from System Libraries and add the text to the current source file. The next line tells CPP to get myheader.h from the local directory and add the content to the current source file.

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

This tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
```

This tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
   /* Your debugging statements here */
#endif
```

This tells the CPP to do the process the statements enclosed if DEBUG is defined. This is useful if you pass the -DDEBUG flag to gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Predefined Macros

ANSI C defines a number of macros. Although each one is available for your use in programming, the predefined macros should not be directly modified.

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format
__TIME__	The current time as a character literal in "HH:MM:SS" format
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.
__STDC__	Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example:

```
#include <stdio.h>
int main()
{
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );

}
```

When the above code in a file test.c is compiled and executed, it produces the following result:

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Preprocessor Operators

The C preprocessor offers following operators to help you in creating macros:

Macro Continuation (\)

A macro usually must be contained on a single line. The macro continuation operator is used to continue a macro that is too long for a single line. For example:

```
#define  message_for(a, b)  \
    printf(#a " and " #b ": We love you!\n")
```

Stringize (#)

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro that has a specified argument or parameter list. For example:

```
#include <stdio.h>

#define  message_for(a, b)  \
     printf(#a " and " #b ": We love you!\n")

int main(void)
{
   message_for(Carole, Debra);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Carole and Debra: We love you!
```

Token Pasting (##)

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example:

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void)
{
   int token34 = 40;

   tokenpaster(34);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

token34 = 40

How it happened, because this example results in the following actual output from the preprocessor:

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both stringize and token-pasting.

The defined() Operator

The preprocessor defined operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows:

```
#include <stdio.h>

#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif

int main(void)
{
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Here is the message: You wish!
```

Parameterized Macros

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows:

```
int square(int x)
{
   return x * x;
}
```

We can rewrite above code using a macro as follows:

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the #define directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between and macro name and open parenthesis.

For example:

```
#include <stdio.h>

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

int main(void)
{
   printf("Max between 20 and 10 is %d\n", MAX(10, 20));
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Max between 20 and 10 is 20
```

Header Files

A header file is a file with extension .h which contains C function declarations and macro definitions and to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that come with your compiler.

You request the use of a header file in your program by including it, with the C preprocessing directive #include like you have seen inclusion of stdio.h header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be very much error-prone and it is not a good idea to copy the content of header file in the source files, specially if we have multiple source file comprising our program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in header files and include that header file wherever it is required.

Include Syntax

Both user and system header files are included using the preprocessing directive #include. It has following two forms:

```
#include <file>
```

This form is used for system header files. It searches for a file named file in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

```
#include "file"
```

This form is used for header files of your own program. It searches for a file named file in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

Include Operation

The #include directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the #include directive. For example, if you have a header file header.h as follows:

```
char *test (void);
```

and a main program called program.c that uses the header file, like this:

```
int x;
#include "header.h"

int main (void)
{
   puts (test ());
}
```

the compiler will see the same token stream as it would if program.c read

```
int x;
char *test (void);

int main (void)
{
   puts (test ());
}
```

Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice and will result an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef HEADER_FILE
#define HEADER_FILE
```

the entire header file file

```
#endif
```

This construct is commonly known as a wrapper `#ifndef`. When the header is included again, the conditional will be false, because HEADER_FILE is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Computed Includes

Sometimes it is necessary to select one of several different header files to be included into your program. They might specify configuration parameters to be used on different sorts of operating systems, for instance. You could do this with a series of conditionals as follows:

```
#if SYSTEM_1
   # include "system_1.h"
#elif SYSTEM_2
   # include "system_2.h"
#elif SYSTEM_3
   ...
#endif
```

But as it grows, it becomes tedious, instead the preprocessor offers the ability to use a macro for the header name. This is called a computed include. Instead of writing a header name as the direct argument of #include, you simply put a macro name there instead:

```
 #define SYSTEM_H "system_1.h"
 ...
 #include SYSTEM_H
```

SYSTEM_H will be expanded, and the preprocessor will look for system_1.h as if the #include had been written that way originally. SYSTEM_H could be defined by your Makefile with a -D option.

Error Handling

As such C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and sets an error code errno is set which is global variable and indicates an error occurred during any function call. You can find various error codes defined in <error.h> header file.

So a C programmer can check the returned values and can take appropriate action depending on the return value. As a good practice, developer should set errno to 0 at the time of initialization of the program. A value of 0 indicates that there is no error in the program.

The errno, perror() and strerror()
The C programming language provides perror() and strerror() functions which can be used to display the text message associated with errno.

The perror() function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.

The strerror() function, which returns a pointer to the textual representation of the current errno value.

Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use stderr file stream to output all the errors.

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

extern int errno ;
int main ()
{
   FILE * pf;
   int errnum;
   pf = fopen ("unexist.txt", "rb");
   if (pf == NULL)
   {
      errnum = errno;
      fprintf(stderr, "Value of errno: %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
   }
   else
   {
      fclose (pf);
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

Divide by zero errors

It is a common problem that at the time of dividing any number, programmers do not check if a divisor is zero and finally it creates a runtime error.

The code below fixes this by checking if the divisor is zero before dividing:

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

int main()
{
   int dividend = 20;
   int divisor = 0;
   int quotient;

   if( divisor == 0){
      fprintf(stderr, "Division by zero! Exiting...\n");
      exit(-1);
   }
   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient );

   exit(0);
}
```

When the above code is compiled and executed, it produces the following result:

```
Division by zero! Exiting...
```

Program Exit Status

It is a common practice to exit with a value of EXIT_SUCCESS in case of programming is coming out after a successful operation. Here, EXIT_SUCCESS is a macro and it is defined as 0.

If you have an error condition in your program and you are coming out then you should exit with a status EXIT_FAILURE which is defined as -1. So let's write above program as follows:

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

main()
{
   int dividend = 20;
   int divisor = 5;
   int quotient;

   if( divisor == 0){
      fprintf(stderr, "Division by zero! Exiting...\n");
      exit(EXIT_FAILURE);
   }
   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient );

   exit(EXIT_SUCCESS);
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of quotient : 4
```

A Simple optimization in C Programs

Let’s check the following stub-

```
for( i=0; i<n; i++)
{
      y[i] = i;
}
```

Check above for n=5 i.e. 5 tests

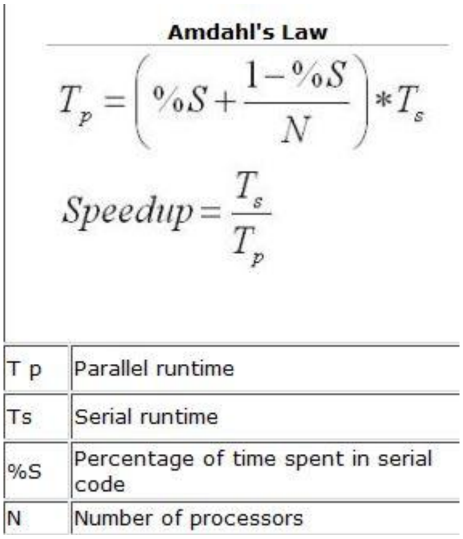
Now check below-

```
for( i=0; i<n%2; i++)
      y[i] = i;
for(; i+1<n; i+=2)
{
      y[i] = i;
      y[i+1] = i+1;
}
```

It’ll take 3 tests i.e. test reduced by fraction of 2.
At some places we’ve to look for space & some place speed.

Be perfect at C-Coding problems: Optimization Techniques

1. Remember Ahmdal’s Law:



Amdahl's Law

$$T_p = \left(\%S + \frac{1-\%S}{N} \right) * T_s$$

$$Speedup = \frac{T_s}{T_p}$$

T p	Parallel runtime
Ts	Serial runtime
%S	Percentage of time spent in serial code
N	Number of processors

This is often phrased as: “make the common case fast and the rare case correct.”

2. Code for correctness first, then optimize!
- This does not mean write a fully functional ray tracer for 8 weeks, then optimize for 8 weeks!
- Perform optimizations on your ray tracer in multiple steps.

- Write for correctness, then if you know the function will be called frequently, perform obvious optimizations.
- Then profile to find bottlenecks, and remove the bottlenecks (by optimization or by improving the algorithm).

Often improving the algorithm drastically changes the bottleneck – perhaps to a function you might not expect. This is a good reason to perform obvious optimizations on all functions you know will be frequently used.

3. People I know who write very efficient code say they spend at least twice as long optimizing code as they spend writing code.

4. Jumps/branches are expensive. Minimize their use whenever possible.
- Function calls require two jumps, in addition to stack memory manipulation.
- Prefer iteration over recursion.
- Use inline functions for short functions to eliminate function overhead.
- Move loops inside function calls (e.g., change for(i=0;i<100;i++) DoSomething(); into DoSomething() { for(i=0;i<100;i++) { ... } }).
- Long if...else if...else if...else if... chains require lots of jumps for cases near the end of the chain (in addition to testing each condition). If possible, convert to a switch statement, which the compiler sometimes optimizes into a table lookup with a single jump. If a switch statement is not possible, put the most common clauses at the beginning of the if chain.

5. Think about the order of array indices.
- Two and higher dimensional arrays are still stored in one dimensional memory. This means (for C/C++ arrays) array[i][j] and array[i][j+1] are adjacent to each other, whereas array[i][j] and array[i+1][j] may be arbitrarily far apart.
- Accessing data in a more-or-less sequential fashion, as stored in physical memory, can dramatically speed up your code (sometimes by an order of magnitude, or more)!
- When modern CPUs load data from main memory into processor cache, they fetch more than a single value. Instead they fetch a block of memory containing the requested data and adjacent data (a cache line). This means after array[i][j] is in the CPU cache, array[i][j+1] has a good chance of already being in cache, whereas array[i+1][j] is likely to still be in main memory.

6. Think about instruction-level-parallelism.
- Even though many applications still rely on single threaded execution, modern CPUs already have a significant amount of parallelism inside a single core. This means a single CPU might be simultaneously executing 4 floating point multiplies, waiting for 4 memory requests, and performing a comparison for an upcoming branch.
- To make the most of this parallelism, blocks of code (i.e., between jumps) need to have enough independent instructions to allow the CPU to be fully utilized.
- Think about unrolling loops to improve this.
- This is also a good reason to use inline functions.

7. Avoid/reduce the number of local variables.
- Local variables are normally stored on the stack. However if there are few enough, they can instead be stored in registers. In this case, the function not only gets the benefit of the faster memory access of data stored in registers, but the function avoids the overhead of setting up a stack frame.

- (Do not, however, switch wholesale to global variables!)

8. Reduce the number of function parameters.
- For the same reason as reducing local variables – they are also stored on the stack.

9. Pass structures by reference, not by value.
- I know of no case in a ray tracer where structures should be passed by value (even simple ones like Vectors, Points, and Colors).

10. If you do not need a return value from a function, do not define one.

11. Try to avoid casting where possible.
- Integer and floating point instructions often operate on different registers, so a cast requires a copy.
- Shorter integer types (char and short) still require the use of a full-sized register, and they need to be padded to 32/64-bits and then converted back to the smaller size before storing back in memory. (However, this cost must be weighed against the additional memory cost of a larger data type.)

12. Be very careful when declaring C++ object variables.
- Use initialization instead of assignment (Color c(black); is faster than Color c; c = black;).

13. Make default class constructors as lightweight as possible.
- Particularly for simple, frequently used classes (e.g., color, vector, point, etc.) that are manipulated frequently.
- These default constructors are often called behind your back, where you are not expecting it.
- Use constructor initializer lists. (Use Color::Color() : r(0), g(0), b(0) {} rather than Color::Color() { r = g = b = 0; } .)

14. Use shift operations >> and << instead of integer multiplication and division, where possible.

15. Be careful using table-lookup functions.
- Many people encourage using tables of precomputed values for complex functions (e.g., trigonometric functions). For ray tracing, this is often unnecessary. Memory lookups are exceedingly (and increasingly) expensive, and it is often as fast to recompute a trigonometric function as it is to retrieve the value from memory (especially when you consider the trig lookup pollutes the CPU cache).
- In other instances, lookup tables may be quite useful. For GPU programming, table lookups are often preferred for complex functions.

16. For most classes, use the operators += , -= , *= , and /= , instead of the operators + , - , * , and / .
- The simple operations need to create an unnamed, temporary intermediate object.
- For instance: Vector v = Vector(1,0,0) + Vector(0,1,0) + Vector(0,0,1); creates five unnamed, temporary Vectors: Vector(1,0,0), Vector(0,1,0), Vector(0,0,1), Vector(1,0,0) + Vector(0,1,0), and Vector(1,0,0) + Vector(0,1,0) + Vector(0,0,1).
- The slightly more verbose code: Vector v(1,0,0); v+= Vector(0,1,0); v+= Vector(0,0,1); only creates two temporary Vectors: Vector(0,1,0) and Vector(0,0,1). This saves 6 functions calls (3 constructors and 3 destructors).

17. For basic data types, use the operators + , - , * , and / instead of the operators += , -= , *= , and /= .

18. Delay declaring local variables.
- Declaring object variable always involves a function call (to the constructor).
- If a variable is only needed sometimes (e.g., inside an if statement) only declare when necessary, so the constructor is only called if the variable will be used.

19. For objects, use the prefix operator (++obj) instead of the postfix operator (obj++).
- This probably will not be an issue in your ray tracer.
- A copy of the object must be made with the postfix operator (which thus involves an extra call the constructor and destructor), whereas the prefix operator does not need a temporary copy.

20. Be careful using templates.
- Optimizations for various instantiations may need to be different!
- The standard template library is reasonably well optimized, but I would avoid using it if you plan to implement an interactive ray tracer.
- Why? By implementing it yourself, you'll know the algorithms it uses, so you will know the most efficient
way to use the code.
- More importantly, my experience is that debug compiles of STL libraries are slow. Normally this isn't a problem, except you will be using debug versions for profiling. You'll find STL constructors, iterators, etc. use 15+% of your run time, which can make reading the profile output more confusing.

21. Avoid dynamic memory allocation during computation.
- Dynamic memory is great for storing the scene and other data that does not change during computation.
- However, on many (most) systems dynamic memory allocation requires the use of locks to control a access to the allocator. For multi-threaded applications that use dynamic memory, you may actually get a slowdown by adding additional processors, due to the wait to allocate and free memory!
- Even for single threaded applications, allocating memory on the heap is more expensive than adding it on the stack. The operating system needs to perform some computation to find a memory block of the requisite size.

22. Find and utilize information about your system's memory cache.
- If a data structure fits in a single cache line, only a single fetch from main memory is required to process the entire class.
- Make sure all data structures are aligned to cache line boundaries. (If both your data structure and a cache line is 128 bytes, you will still have poor performance if 1 byte of your structure is in once cache line and the other 127 bytes are in a second cache line).

23. Avoid unnecessary data initialization.
- If you must initialize a large chunk of memory, consider using memset().

24. Try to early loop termination and early function returns.
- Consider intersecting a ray and a triangle. The "common case" is that the ray will miss the triangle. Thus, this should be optimized for.

- If you decide to intersect the ray with the triangle plane, you can immediately return if the t value ray-plane intersection is negative. This allows you to skip the barycentric coordinate computation in roughly half of the ray-triangle intersections. A big win! As soon as you know no intersection occurs, the intersection function should quit.
- Similarly, some loops can be terminated early. For instance, when shooting shadow rays, the location of the nearest intersection is unnecessary. As soon as any occluding intersection is found, the intersection routine can return.

25. Simplify your equations on paper!

- In many equations, terms cancel out... either always or in some special cases.
- The compiler cannot find these simplifications, but you can. Eliminating a few expensive operations inside an inner loop can speed your program more than days working on other parts.

26. The difference between math on integers, fixed points, 32-bit floats, and 64-bit doubles is not as big as you might think.

- On modern CPUs, floating-point operations have essentially the same throughput as integer operations. In compute-intensive programs like ray tracing, this leads to a negligible difference between integer and floating-point costs. This means, you should not go out of your way to use integer operations.
- Double precision floating-point operations may not be slower than single precision floats, particularly on 64-bit machines. I have seen ray tracers run faster using all doubles than all floats on the same machine. I have also seen the reverse.

27. Consider ways of rephrasing your math to eliminate expensive operations.

- sqrt() can often be avoided, especially in comparisons where comparing the value squared gives the same result.
- If you repeatedly divide by x, consider computing 1 x and multiplying by the result. This used to be a big win for vector normalizations (3 divides), but I've recently found it's now a toss-up. However, it should still be beneficial if you do more than 3 divides.
- If you perform a loop, make sure computations that do not change between iterations are pulled out of the loop.
- Consider if you can compute values in a loop incrementally (instead of computing from scratch each iteration).

Common Q & A

Variables & Control Flow

1. What is the difference between declaring a variable and defining a variable?
2. What is a static variable?
3. What is a register variable?
4. Where is an auto variable stored?
5. What is scope & storage allocation of extern and global variables?
6. What is scope & storage allocation of register, static and local variables?
7. What are storage memory, default value, scope and life of Automatic and Register storage class?
8. What are storage memory, default value, scope and life of Static and External storage class?
9. What is the difference between 'break' and 'continue' statements?
10. What is the difference between 'for' and 'while' loops?

Operators, Constants & Structures

1. Which bitwise operator is suitable for checking whether a particular bit is ON or OFF?
2. Which bitwise operator is suitable for turning OFF a particular bit in a number?
3. What is equivalent of multiplying an unsigned int by 2: left shift of number by 1 or right shift of number by1?
4. What is an Enumeration Constant?
5. What is a structure?
6. What are the differences between a structure and a union?
7. What are the advantages of unions?
8. How can `typedef` be to define a type of structure?
9. Write a program that returns 3 numbers from a function using a structure.
10. In code snippet below:

```
struct Date
{
int yr;
int day;
int month;
}date1,date2;

date1.yr = 2004;
date1.day = 4;
date1.month = 12;
```

Write a function that assigns values to date2. Arguments to the function must be pointers to the structure, Date and integer variables date, month, year.

.

Functions

1. What is the purpose of main() function?
2. Explain command line arguments of main function?
3. What are header files? Are functions declared or defined in header files ?
4. What are the differences between formal arguments and actual arguments of a function?
5. What is pass by value in functions?
6. What is pass by reference in functions?
7. What are the differences between getchar() and scanf() functions for reading strings?

8. Out of the functions fgets() and gets(), which one is safer to use and why?
9. What is the difference between the functions strdup() and strcpy()?

Pointers

1. What is a pointer in C?
2. What are the advantages of using pointers?
3. What are the differences between `malloc() and calloc()`?
4. How to use `realloc()` to dynamically increase size of an already allocated array?
5. What is the equivalent pointer expression for referring an element a[i][j][k][l], in a four dimensional array?
6. Declare an array of three function pointers where each function receives two integers and returns float.
7. Explain the variable assignment in the declaration

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

8. What is the value of `sizeof(a) /sizeof(char *)`in a code snippet:

```
char *a[4]={"sridhar","raghava","shashi","srikanth"};
```

9. (i) What are the differences between the C statements below:

```
char *str = "Hello";
char arr[] = "Hello";
```

(ii) Whether following statements get complied or not? Explain each statement.

```
arr++;
*(arr + 1) = 's';
printf("%s",arr);
```

Programs

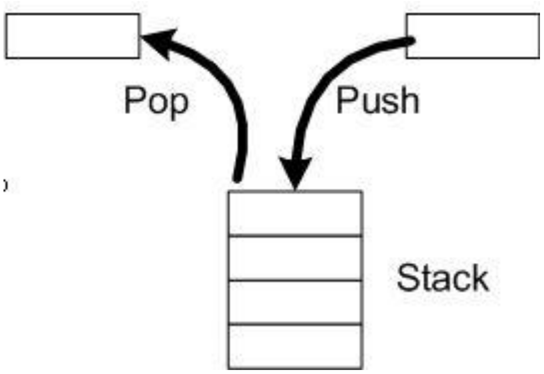
1. Write a program to find factorial of the given number.
2. Write a program to check whether the given number is even or odd.
3. Write a program to swap two numbers using a temporary variable.
4. Write a program to swap two numbers without using a temporary variable.
5. Write a program to swap two numbers using bitwise operators.
6. Write a program to find the greatest of three numbers.
7. Write a program to find the greatest among ten numbers.
8. Write a program to check whether the given number is a prime.
9. Write a program to check whether the given number is a palindromic number.
10.Write a program to check whether the given string is a palindrome.
11.Write a program to generate the Fibonacci series.
12.Write a program to print "Hello World" without using semicolon anywhere in the code.
13.Write a program to print a semicolon without using a semicolon anywhere in the code.
14.Write a program to compare two strings without using strcmp() function.
15.Write a program to concatenate two strings without using strcat() function.
16.Write a program to delete a specified line from a text file.
17.Write a program to replace a specified line in a text file.
18.Write a program to find the number of lines in a text file.
19. Write a C program which asks the user for a number between 1 to 9 and shows the number. If the user inputs a number out of the specified range, the program should show an error and prompt the user for a valid input.
20. Write a program to display the multiplication table of a given number.

4

Concept to Advance Data Structure

STACK

Basic Concept of Stack



Definition :
Stack is first in last out Structure (LIFO) Structure .

Position of Top and Its Value :

Position of Top	**Status of Stack**
-1	**Stack is Empty**
0	**First Element is Just Added into Stack**
N-1	**Stack is said to Full**
N	**Stack is said to be Overflow**

Stack can be represented as “Array” .

/* Simple Stack Program*/

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 5
void push( int a[],int*);
void pop (int a[], int*);
void display(int a[], int*);

int main()
{
     int ch; int t= -1;
     int a[SIZE], *p;
     p = &t;
     //printf(" press 1: push, 2: pop, 3: display \n");
     //scanf("%d", &ch);
     for(;;)
     {

           printf(" press 1: push, 2: pop, 3: display \n");
             scanf("%d", &ch);

          switch (ch)
          {
               case 1: push(a,p);
                    break;
               case 2:pop(a,p);
                    break;
               case 3:display(a,p);
                    break;
               default:
                    exit(0);
          }}
     return 0;

}
void push(int a[], int *p)
{
     if(*p == SIZE-1)
     {
          printf("Stack over flow\n");
          return;
     }
     printf("Enter the element\n");
     scanf("%d",&a[++(*p)]);
     return;
}

void pop(int a[], int *p)
{
     if ( *p == (-1))
     {
          printf("stack is emplty\n");
          return;
     }
     printf("pop out element is %d \n ", a[(*p)--]);
}
void display (int a[], int *p)
{
     int i;
     if (*p == (-1))
     {
          printf("Stack is empty\n");
          return ;
     }
     for( i=0; i<= SIZE; i++)
     {
          printf("%d", a[i]);
     }

}
```

Advance Level Implementation of Stack

```
/* stack.h */
#ifndef STACK_H
#define STACK_H

#define STACK_SUCCESS 0
#define STACK_FULL    1
#define STACK_EMPTY   2
#define STACK_MEM_FAILURE  ((void *)0)

#define INIT_TOP_OF_STACK  -1

typedef struct Stack * STACK;
STACK CreateNewStack( int SizeOfData, int MaxElementsToStore );
void DisposeStack( STACK S);
int Push( void* Element, STACK S);
int Pop( STACK S, void* Element);
int IsStackFull ( STACK S);
int IsStackEmpty ( STACK S);
void InitTopOfStack (STACK S);

#endif
```

libStack.c

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

struct Stack
{
     int Capicity;
     char *StackPointer;
     int TopOfStack;
     int SizeOfData;
};
STACK CreateNewStack( int SizeOfData, int MaxElementsToStore )
{
     STACK S;
     if (( S= ( STACK ) malloc(sizeof(struct Stack)))== NULL )
     {
          return STACK_MEM_FAILURE;
     }
     if  (  (S->  StackPointer  =  (char  *)malloc  (MaxElementsToStore  *
SizeOfData)) == NULL)
     {
          return STACK_MEM_FAILURE;
     }
     S-> Capicity = MaxElementsToStore;
     S-> SizeOfData;
     InitTopOfStack (S);
     return S;
}
void InitTopOfStack (STACK S)
{
     S-> TopOfStack = INIT_TOP_OF_STACK;
}
void DisposeStack (STACK S)
{
     if( S != NULL)
     {
          if(S-> StackPointer != NULL)
          {
               free( S->StackPointer);
          }
     free(S);
     }
}
int Push (void *Element, STACK S)
{
     if (IsStackFull (S) )
          return STACK_FULL;

     memcpy (S -> StackPointer, Element, S-> SizeOfData);
     S-> StackPointer = S->StackPointer + S-> SizeOfData;
     ++S-> TopOfStack;

     return STACK_SUCCESS;
}

int Pop (STACK S, void *Element)
{
     if( IsStackEmpty(S))
          return STACK_EMPTY;
     S->StackPointer = S->StackPointer - S->SizeOfData;
     --S->TopOfStack;
     memcpy( Element, S->StackPointer, S->SizeOfData );
     return STACK_SUCCESS;
}

int IsStackFull (STACK S)
{
     return S-> TopOfStack == (S->Capicity -1);
}
```

```
int IsStackEmpty (STACK S)
{
     return S->TopOfStack == INIT_TOP_OF_STACK;
}
```

AppUsingStack.c

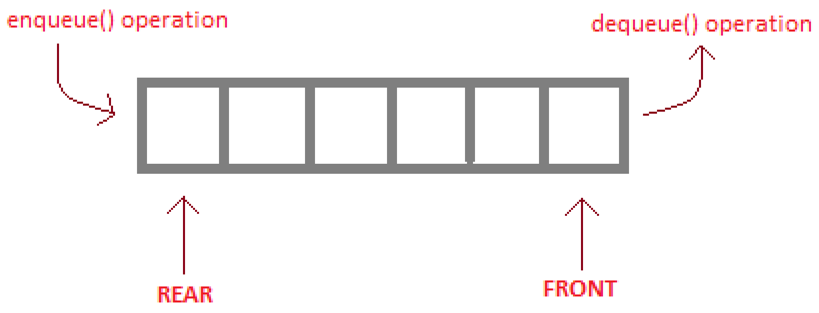
```
#include<stdio.h>
#include<stdlib.h>
#include"stack.h"

#define LENGTH 50

int main()
{
     char *Move;
     STACK S;
     char EnteredLine[LENGTH];
     int Number;

     if (( S= CreateNewStack(sizeof(int), 3)) == STACK_MEM_FAILURE )
     {
          fprintf (stderr, "cannot create new stack due to memory
issue\n");
          exit(1);
     }
     while(1)
     {
          if (IsStackEmpty(S))
          {
               printf("USAGE: <NUM> to PUSH, e or E to EXIT\n");
          }
          else if (IsStackFull(S))
          {
               printf("USAGE:p or P to POP, e or E to EXIT\n");
          }
          else
          {
               printf("USAGE: <NUM> to PUSH, p or P to POP, e or E to
EXIT\n");
          }
          if ( !fgets(EnteredLine, LENGTH, stdin) )
               exit(2);
          for( Move = EnteredLine; *Move == ' ' || *Move == '\t'; Move
++);

          if( isdigit (*Move))
          {
               Number = atoi(Move);
               if (Push ( &Number, S) == STACK_SUCCESS )
                    printf("PUSHING is sucess\n");
               else
                    printf("CANNOT PUSH: STACK FULL\n");
          }
          else if (*Move =='p' || *Move == 'P')
          {
               if( Pop (S, &Number ) == STACK_SUCCESS )
                    printf("POPPED NUMBER = %d\n\n", Number);
               else
                    printf("CANNOT POP: STACK EMPTY\n");
          }
          else if( *Move == 'e' || *Move == 'E')
          {
               while (Pop (S,&Number ) == STACK_SUCCESS )
               {
                    printf("POPPED NUMBER WHILE EXITING = %d\n",Number);
               }
                    break;
          }

     }
     DisposeStack(S);
     printf("Good Bye\n");
     return 0;
}
```

Queue

Queue is a specialized data storage structure (Abstract data type). Unlike, arrays access of elements in a Queue is restricted. It has two main operations enqueue and dequeue. Insertion in a queue is done using enqueue function and removal from a queue is done using dequeue function. An item can be inserted at the end ('rear') of the queue and removed from the front ('front') of the queue.

It is therefore, also called First-In-First-Out (FIFO) list. Queue has five properties - capacity stands for the maximum number of elements Queue can hold, size stands for the current size of the Queue, elements is the array of elements, front is the index of first element (the index at which we remove the element) and rear is the index of last element (the index at which we insert the element).

Basic Concept of Queue



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

/* Simple Queue Program*/

```
#include<stdio.h>
#define SIZE 5
void insert( int a[], int*);
void delete( int a[], int*, int*);
void display(int a[], int*, int*);
int main()
{
      int a[SIZE];
      int r= -1, f= 0, *p, *q, ch;
      p = &r;
      q = &f;

      for(;;)
      {
            printf("Enter choice 1:INSERT, 2: DELETE, 3: DISPLAY \n");
            scanf("%d",&ch);
            switch(ch)
            {
                  case 1: insert(a,p);
                        break;
                  case 2: delete(a,p,q);
                        break;
                  case 3: display(a,p,q);
                        break;
                  default:
                        exit(0);

            }

      }
      return 0;
}

void insert( int a[], int *p)
{
      if (*p == SIZE-1)
      {
            printf("Queve is over flow\n");
            return;
      }
      printf("Enter the element\n");
      scanf("%d",&a[++(*p)]);
}

void delete( int a[], int *p, int *q)
{
      if (*q > *p)
      {
            printf("Queue is empty\n");
            return;
```

```
        }
        printf("deleted element is %d \n", a[(*q)++]);
}

void display( int a[SIZE], int *p, int *q)
{
        int i;
        if (*q > *p)
        {
                printf("Queue is empty\n");
                return;
        }
        for(i = *q; i<= *p; i++)
        {
                printf("%d",a[i]);
        }
}
```

Advance Level Implementation of Queue

/*queue.h.h*/

```
#ifndef QUEUE_H
#define QUEUE_H
#define QUEUE_SUCCESS 0
#define INVALID_QUEUE ((void *)0)
#define QUEUE_MEM_FAILURE ((void *)0)
#define QUEUE_FULL 1
#define QUEUE_EMPTY 2
typedef struct Queue * QUEUE;
QUEUE CreateNewQueue(int SizeOfData, int capacity);
int Enqueue(QUEUE Q, void *data);
int Dequeue(QUEUE Q, void *data);
int DisposeQueue (QUEUE Q);
int IsQueueFull(QUEUE Q);
int IsQueueEmpty(QUEUE Q);
#endif
```

/* libQueue.c */

```
#include"queue.h"
#include<stdio.h>
#include<stdlib.h>
struct Queue
{
        int capacity;
        int SizeOfData;
        int CountOfElements;
        char *Last;
```

```
      char *Head;
      char *Tail;
      char *QPointer;
};

QUEUE CreteNewQueue(int SizeOfData, int capacity)
{
      QUEUE Q;
      if(capacity<1 || SizeOfData<1)
           return INVALID_QUEUE;
      if((Q=malloc(sizeof(struct Queue)))==NULL)
           return QUEUE_MEM_FAILURE;
      if((Q->QPointer=malloc(capacity * SizeOfData))==NULL)
           return QUEUE_MEM_FAILURE;
      Q->Head=Q->Tail=Q->QPointer;
      Q->Last=Q->QPointer+capacity * SizeOfData;
      Q->capacity=capacity;
      Q->SizeOfData=SizeOfData;
      Q->CountOfElements=0;
      return 0;
}
int Enqueue(QUEUE Q, void *item)
{
      if(IsQueueFull(Q))
           return QUEUE_FULL;
      memcpy(Q->Head,item,Q->SizeOfData);
      if((Q->Head+=Q->SizeOfData)==Q->Last)
           Q->Head=Q->QPointer;
      Q->CountOfElements++;
      return QUEUE_SUCCESS;
}
int Dequeue(QUEUE Q, void *item)
{
        if(IsQueueEmpty(Q))
                return QUEUE_EMPTY;
        memcpy(item,Q->Tail,Q->SizeOfData);
        if((Q->Tail+=Q->SizeOfData)==Q->Last)
                Q->Tail=Q->QPointer;
        Q->CountOfElements--;
        return QUEUE_SUCCESS;
}
int DisposeQueue(QUEUE Q)
{
      if(Q!=NULL)
      {
           if(Q->QPointer!=NULL)
           {
                free(Q->QPointer);
           }
           free(Q);
      }
      return QUEUE_SUCCESS;
```

```
}
int IsQueueFull(QUEUE Q)
{
      return Q->CountOfElements==Q->capacity;
}
int IsQueueEmpty(QUEUE Q)
{
      return Q->CountOfElements==0;
}
```

/* AppUsingQueue.c */

```
#include<stdio.h>
#include<stdlib.h>
#include"queue.h"
#define LENGTH 50
int main()
{
      char *Move;
      QUEUE Q;
      char EnteredLine[LENGTH];
      int Number;
      if((Q=CreateNewQueue(sizeof(int),3))==QUEUE_MEM_FAILURE)
      {
            ffprintf(stderr,"cannot cretae new Q due to memory issues\n");
            exit(1);
      }
      while(1)
      {
            if(IsQueueEmpty(Q))
            {
                  printf("USGE:<NUM>to ENQUEUE\n");
            }
            else if(IsQueueFull(Q))
            {
                  printf("USAGE: d to DEQUEUE");
            }
            else
            {
                  printf("USAGE: d or D to DEQUEUE,<NUM> to ENQUEUE\n");
            }
            if(!fgets(EnteredLine,LENGTH,stdin))
                  exit(2);
            for(Move=EnteredLine;*Move==' ' || *Move=='\t';Move++);
            if(isdigit(*Move))
            {
                  Number=atoi(Move);
                  if(Enqueue(Q,&Number)==QUEUE_SUCCESS)
                        printf("Enqueue is successful\n");
            else
                printf("cannot enqueue: QUEUE IS FULL\n");
        }
        else if(*Move=='d' || *Move =='D')
        {
            if(Dequeue(Q,&Number)==QUEUE_SUCCESS)
                printf("RETRIVED NUMBER=%d\n",Number);
            else
                printf("cannot dequeue: QUEUE IS EMPTY\n");
        }
    }
    DisposeQueue(Q);
    printf("Good Bye\n");
    return 0;
}
```

LinkedList

A list refers to a set of items organized sequentially. An array is an example of list. In an array, the sequential organization is provided implicitly by its index. The major problem with the arrays is that the size of an array must be specified precisely at the beginning, which is difficult in many practical applications.

Linked list is a linked list of structures (called nodes) for which memory is allotted dynamically. It is necessary to include an additional member that is pointer to structure.

Eg:

```
struct node
{
        data_type info;
        struct node * next;
};
```

Fig 8.1 Structure of a node in singly linked list

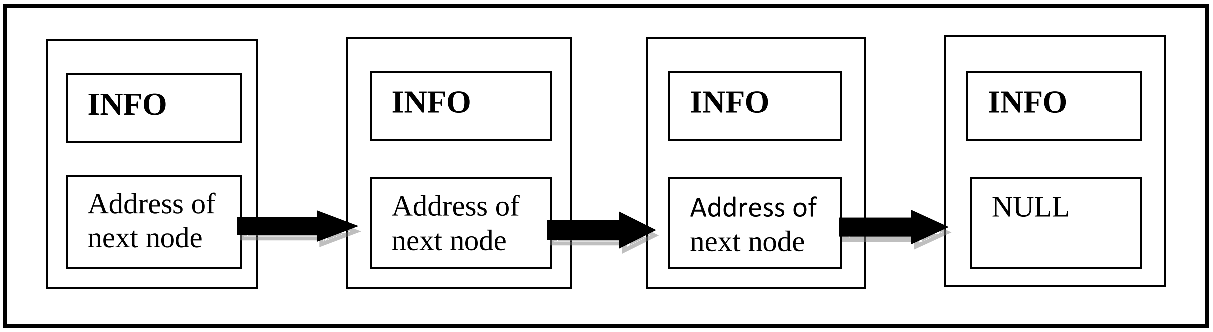


Fig 8.2 Pictorial representation of singly linked list in memory

The additional member pointer next, keeps the address of the next node in the linked list. The pointer next of the last node always points to NULL. One more pointer to structure must be declared to keep track of the first node of the linked list, which is not a member pointer.

Creating a linked list

Creation of a linked list requires the following three steps to be performed.

1. Define the structure of the node that will hold the data for each element in the list. Let us assume that the data we intend to store is the empid of the employee (which is unique), his name and salary.

The structure is defined as follows:

```
struct node
{
      int empid;
      char name[20];
      float salary;
      struct node *next;
};
```

Structure for employee node in the linked list

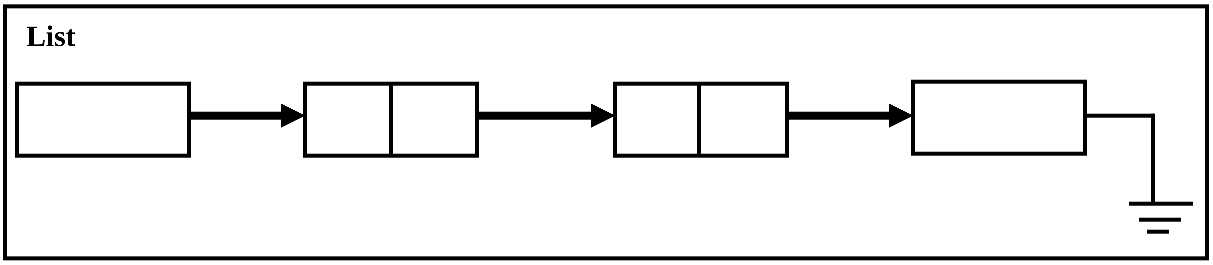
Note that the last element in the structure is a pointer to the next node in the list.

2. In a linked list nodes are created dynamically as and when required. So let us create a general purpose function that will return a node for which data has been entered by the user.

```
struct node *getnode()     /*creates a node and accepts data*/
{
      struct node *temp;
      temp=(struct node*)malloc(sizeof(struct node));
      printf(“enter the empid:”);
      scanf(“%d”,&temp->empid);
      fflush(stdin);
      printf(“enter the name:”);
      scanf(“%s”,temp->name);
      fflush(stdin);
      printf(“enter salary:”);
      scanf(“%f”,temp->salary);
      fflush(stdin);
      temp->next=NULL;
      return temp;
}
```

Code for creation of a node in the linked list

The list needs a pointer that will point to the beginning of the list. For this, we shall create as a global pointer that can be directly accessed by all the functions.

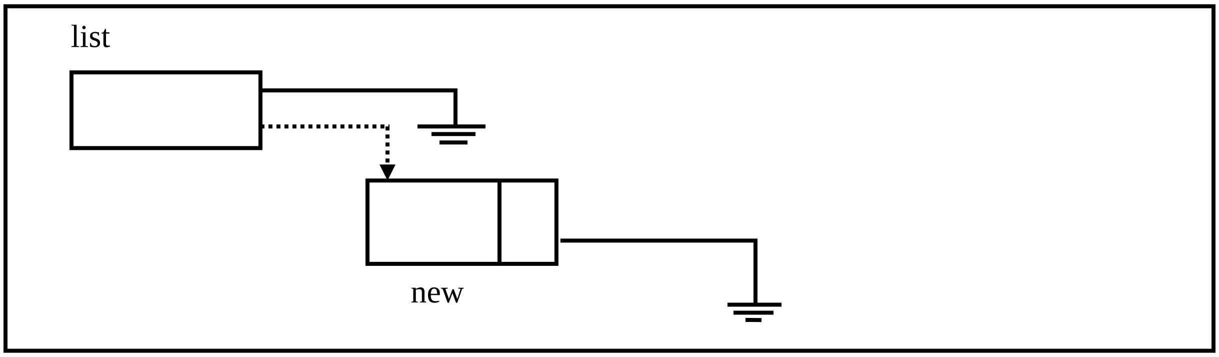


Header (list) for the linked list

The pointer will be initialized to NULL in the main functions.

3. Link the new node to the list.

Next, a function is required that will link a new node to the list. Let us call this function insert. This function will take the address of the node to be inserted as a parameter. If the list is empty, then the new node becomes the first node:

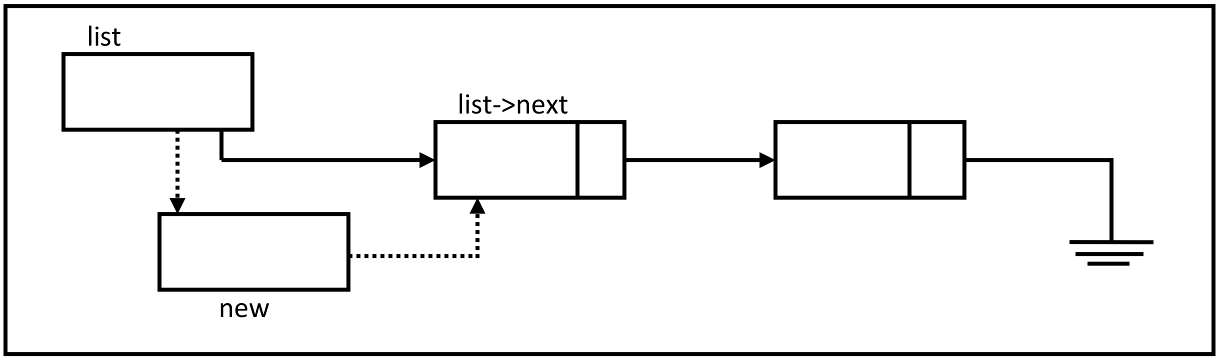


Making header list to point the first node of the linked list

Note that the dotted lines in the figure indicate the operation being performed in the list.

Assume that other nodes are required to be inserted in sorted order of empid. If the list is not empty, then the new node has to be added either to the beginning of the list or to the middle or end of list:

To add to the beginning of the list



Insertion of a node at the beginning of the linked list

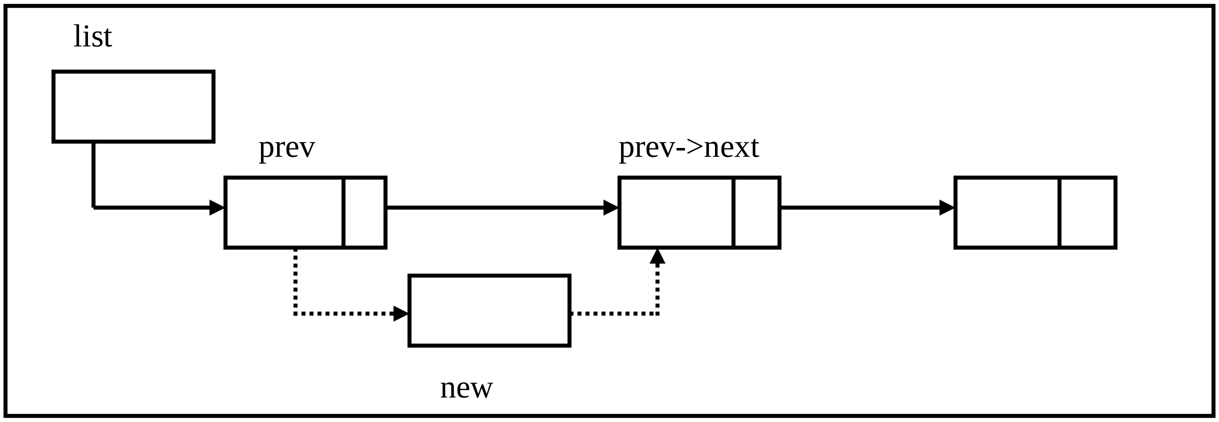
The new node has to point to existing of the list. This is done by the statement

```
new->next=list;
```

And the list pointer must point to the new node:

```
      list=new;
```

To add to the middle of the list

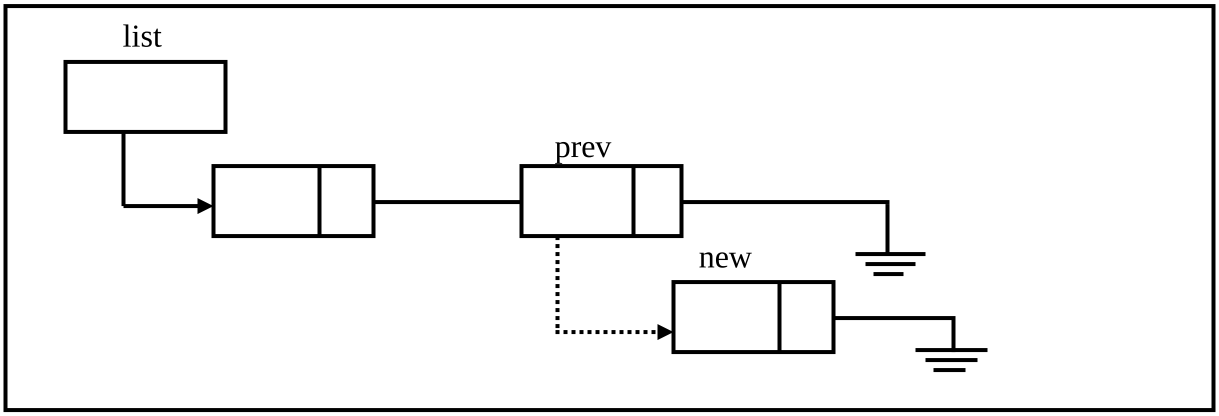


Insertion of a node at the middle of the linked list

The new node has to be inserted after the node pointed by prev node then previous node should point to new and new will point to prev.

```
        new->next=prev->next;
        prev->next=prev;
```

To add to the end of the list



Insertion of a node at the end of the linked list

The last node should point to new node and new node should point to NULL to become last node.

```
        new->next=NULL;
```

Which is equivalent to

```
        new->next=prev->next;
```

then

```
        prev->next=new;
```

Insertion of new node in the list

Note that in a singly linked list, nodes can only be inserted or deleted after a given node. So a general-purpose search function will be needed which will return the address of the node after which the new node is to be inserted/deleted. For the moment, let us assume a search function that takes the empid as a parameter and returns the address of the last node whose empid value is less than the given node. We can now code the insert function as-

```
int insert(struct node *new)
{
        struct node *prev;
        int flag;
        if (list==NULL)  /* list empty */
        {
                list=new;
                return 0;
        }
        prev=search(new->empid,&flag);
        if(flag==1)      /* duplicate empid */
                return -1;
        if(prev==NULL)   /* insert at beginning */
        {
                new->next=list;
                list=new;
        }
        else             /* insert at middle or end */
        {
                new->next=prev->next;
                prev->next=new;
        }
        return 0;
}
```

Code for insertion of a node in the linked list

Searching a node in the list

The search function traverses the list to find the first node whose empid value is greater than or equal to the empid value received as a parameter. Flag is set to 1 if a node is found with the same empid value else it is set to 0. Thus the search function can also be used to find a node with a given empid value. The search function can thus be used to display the details of a given node (identified by empid).

We display the details of the next node to that returned by search if flag is set to 1. During insert however, if a node with the same empid is found the insert operation fails.

To code the search function:

```
struct node * search(int id,int *flag)
{
     struct node *prev,*cur;
     *flag=0;
     if(list==NULL)   /* list empty */
           return NULL;
     for(prev=NULL,cur=list;((cur)&&(cur->empid)<id));
                                 prev=cur,cur=cur->next);
     if((cur)&&(cur->empid==id))
     /* node with given empid exists */
           *flag=1;
     else
           *flag=0;
     return prev;
}
```

Code for creation of a node in the linked list

Displaying the linked list

To display all the nodes in a linked list, we need to traverse the list sequentially and print the details.

```
void displayall()
{
     struct node *cur;
     system(“clear”);
     if(list==NULL)
     {
           printf(“list is empty\n”);
     }
     printf(“empid, name, salary\n”);
     for(cur=list;cur;cur=cur->next)
           printf(“%4d%-22s%8.2f\n”,cur->empid,cur-name,
                                            cur->salary);
}
```

Code for displaying all nodes in the linked list

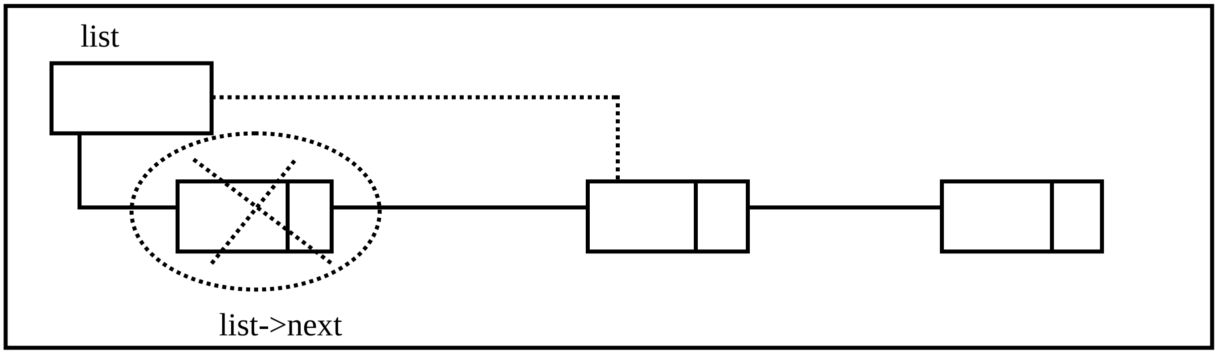
Deletion of existing node from linked list

To modify a node search for the node and accept the details again. To delete a node links have to be reformulated to exclude the deleted node. The memory allocated for the deleted node must also be freed.

The node to be freed

- may not be existing in the list
- may be the first node(in which case the list pointer must be reinitialised)
- may be any other node or the list may be empty

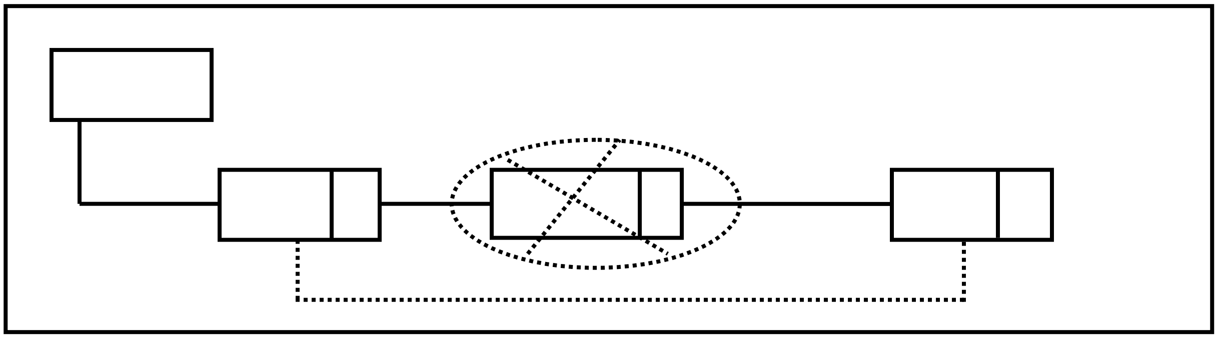
- To delete the first node



Deletion of the first node from the linked list

```
        temp=list;       /* where temp is defined as struct node */
        list=list->next;
        free(temp);
```

- If the first node is also the last node in the list, list automatically becomes NULL.
- To delete other nodes



Deletion of the middle node from the linked list

```
     temp=prev->next;
     prev->next=temp->next;
     free(temp);
```

- So the delete function can be coded as

```
int delete(int id)
{
    struct node *prev,*temp;
    int flag;
    if(list==NULL)    /* list empty */
          return -1;
    prev=search(id,&flag);
    if(flag==0)       /* empid not found */
          return -1;
    if(prev==NULL)
    /* node to delete is first node(as flag is 1) */
    {
          temp=list;
          list=list->next;
          free(temp);
    }
    else
    {
          temp=prev->next;
          prev->next=temp->next;
          free(temp);
    }
    return 0;
}
```

Deletion of the first node from the linked list

Complete program for the operations of linked list

Let us put the above modules into a complete program that will insert, delete or display information from a singly linked list

```
#include<stdio.h>
#include<alloc.h>
struct node
{
     int empid;
     char name[20];
     float salary;
     struct node *next;
};
struct node *list;    /* global pointer to beginning of list */
struct node * getcode() /*creates a node and accepts data*/
{
    struct node *temp;
    temp=(struct node *)malloc(sizeof(struct node));
    printf(“enter the empid:”);
    scanf(“%d”,&temp->empid);
    fflush(stdin);
    printf(“enter the name:”);
    scanf(“%s”,temp->name);
    fflush(stdin);
    printf(“enter the salary:”);
    scanf(“%f”,&temp->salary);
    fflush(stdin);
    temp->next=NULL;
    return temp;
}
/* search returns address of previous node; current node is */
Struct node * search(int id,int *flag)
{
    struct node *prev,*cur;
    *flag=0;
    if(list==NULL)    /* list empty */
           return NULL;
    for(prev=NULL,cur=list;((cur)&&((cur->empid)<id));
                                 prev=cur,cur=cur->next);
    if((cur)&&(cur->empid==id))
    /* node with given empid exists */
           *flag=1;
    else
           *flag=0;
    return prev;
}
int insert(struct node *new)
{
    struct node *prev;
    int flag;
    if(list==NULL)    /*list empty */
    {
           list=new;
           return 0;
    }
    prev = search(new->empid,&flag);
    if(flag==1)       /*duplicate empid */
           return -1;
    if(prev==NULL)    /*insert at beginning */
    {
           new->next=list;
           list=new;
    }

    else   /* insert at middle or end */
```

```
     {
           new->next=prev->next;
           prev->next=new;
     }
     return 0;
}
void displayall()
{
    struct node *cur;
    system(“clear”);
    if(list==NULL)
    {
           printf(“list is empty\n”);
           return;
    }
    printf(“empid name salary\n”);
    for(cur=list;cur;cur=cur->next)
           printf(%4d%-22s%8.2f\n”,cur->empid,cur->name,
                                       cur->salary);
}
int delete(int id)
{
    struct node *prev,*temp;
    int flag;
    if(list==NULL)    /*list empty*/
           return -1;
    prev=search(id,&flag);
    if(flag==0)       /*empid not found*/
           return -1;
    if(prev=NULL)
    /* node to delete is first node (as flag is 1) */
    {
           temp=list;
           list=list->next;
           free(temp);
    }
    else
    {
           temp=prev->next;
           prev->next=temp->next;
           free(temp);
    }
    return 0;
}


void main(void)
{
    struct node *new;
    int choice=1,id;
    list=NULL;
```

```
     do
     {
           printf(“\n\n\n\n\n\t\t\t\tMenu\n\n”);
           printf(“\t\t\t\t1. Insert\n”);
           printf(“\t\t\t\t2.Delete\n”);
           printf(“\t\t\t\t3.Display list\n”);
           printf(“\t\t\t\t0.Exit\n”);
           printf(“\n\n\t\t\t\t......enter choice:”);
           scanf(“%d”,&choice);
           fflush(stdin);
           system(“clear”);
           switch(choice)
           {
                case 1 : new =getnode();
                         if(insert(new)==-1)
                              printf(“error:cannot insert\n”);
                         else
                              printf(“node insert\n”);
                         getchar();
                         break;
                case 2 :
                          printf(“enter the empid to delete\n”);
                          scanf(“%d”,&id);
                          fflush(stdin);
                          if(delete(id)==-1)
                               printf(“delection failed\n”);
                          else
                               printf(“node deleted\n”);
                          getchar();
                          break;
                case 3 :
                         displayall();
                         getchar();
                         break;
                case 0 :
                         exit();

           }
     }while(choice !=0);
}
```

Complete code for linked list operations

Doubly linked list

A doubly linked list is very similar to the normal linked list, except that it has two links: One to the next node and the other to the previous node. So, the structure now has two additional member pointers for each link.

The advantage of a doubly linked list is that you can traverse in both directions using directions using the doubly linked list. The structure now looks as follows:

E.g.

```
struct node
{
            data_type info;
            struct node *next;
            struct node *prev;
};
```

The linked program generally remains the same, except that now you need to handle the additional link. So when a new node is to be added to an existing linked list, you need to assign to the pointer prev, the address of the previous node.

Advance Level List Implementation

/*list.h*/

```
#ifndef LIST_H
#define LIST_H

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

#define EMPTY 1
#define NOEMPTY 0

typedef struct Node* NODE;

int size;
void Init(int);
NODE I_F(void* , NODE);
NODE I_R(void* , NODE);
NODE D_F(NODE);
NODE D_R(NODE);
int Extract(void* , NODE*);
NODE GetNewNode();
void FreeNode(NODE);

#endif
```

/* libList.c */

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

struct Node
{
     char *Data;
     struct Node *ptr;
};

void Init (int Size)
{
      size = Size;
}
NODE I_R(void* item, NODE Head)
{
     NODE Last = Head;
     NODE Temp = GetNewNode();
     memcpy( Temp->Data, item, size);
     Temp ->ptr = NULL;
     if (Last == NULL)
     {
          Head = Temp;
          return Head;
     }
     while (Last ->ptr)
     {
          Last = Last->ptr;
     }
     Last ->ptr = Temp;
     return Head;
}
NODE D_R(NODE Head)
{
     NODE Last = Head;
     NODE Prev = NULL;
     if (Head == NULL)
     {
          printf("List is empty \n");
     }
     else if( !Head->ptr)
     {
          FreeNode(Head);
          Head = NULL;
          return Head;
     }
     else
     {
          while (Last ->ptr)
          {
               Prev = Last;
               Last = Last->ptr;
          }
          Prev ->ptr = NULL;
          FreeNode(Last);
     }
     return Head;
}

NODE I_F( void *item , NODE Head)
{
     NODE Temp;
     Temp = GetNewNode();
     memcpy( Temp ->Data , item , size);
     Temp->ptr = Head;
     Head = Temp;
     return Head;
}
NODE D_F( NODE Head)
{
     if( Head != NULL)
     {
          NODE Temp = Head;
          Head = Head-> ptr;
          free( Temp);
     }
     else
     {
          printf("List is empty\n");
     }
     return Head;
}
int Extract( void*item, NODE *Head)
{
          static int flag = 0;
          static struct Node *Saved;
          if( ! flag)
          {
               Saved = *Head;
               flag ++;
          }
          if ( *Head == NULL)
          {
               *Head = Saved;
               flag =0;
               return 0;
          }
          if ( *Head != NULL)
          {
               memcpy( item, (*Head)->Data, size);
               *Head = (*Head)->ptr;
          }
          return 1;
}
NODE GetNewNode()
{
     NODE Temp = (NODE) malloc(sizeof(struct Node));
     if ( Temp != NULL)
     {
          Temp -> Data = (char*)malloc(sizeof(size));
          return Temp;
     }
     else
     {
          printf("GetNewNode Failed: Could not allocate mem");
          exit(0);
     }
}
void FreeNode( NODE Temp)
{
     if(Temp != NULL)
     {
          if(Temp->Data != NULL)
               free(Temp->Data);
          free(Temp);
     }
}
int IsListEmpty( NODE Head)
{
     if( Head == NULL)
     {
          return EMPTY;
     }
     else
     {
          return NOEMPTY;
     }
}
```

/*AppUsingListChar.c */

```
/* AppUsingListChar */
#include<stdio.h>
#include<stdlib.h>
#include"LinkedList.h"

#define LENGTH 50

int main()
{
     NODE Head = NULL;
     char item, choice, CheckForEmptyList = NOEMPTY;
     char * Move;
     char EnteredLine[LENGTH];

     Init(sizeof(char));
     while(1)
     {
          if( IsListEmpty(Head))
          {
               CheckForEmptyList = EMPTY;
               printf("Usage Option:\n1: Insert Front\n 2: Insert 
Rare\n");
          }
          else
          {
               CheckForEmptyList = NOEMPTY;
               printf("Usage Option:\n1: Insert Front\n2: Insert Rear\n3: 
Delete Front\n4: Delete Rear\n5: Display\n");
          }
          if(! fgets(EnteredLine, LENGTH, stdin))
          {
               printf("Unable to retrive the input\n");
          }
          for (Move = EnteredLine; *Move == ' ' || *Move == '\t'; Move++
);
          if( isdigit (*Move) )
          {
          choice = atoi(Move);
          for ( Move = Move +1; *Move ==' ' || *Move == '\t' ;Move++);
          if (*Move != '\n')
          {
               printf("Enter Valid Option\n");
               continue;
          }
          if( choice >= 1 && choice <=5)
          {
               if(CheckForEmptyList && ((choice <1) || (choice >2)  ))
               {
               printf("Enter Valid Option Num\n");
               continue;
               }
               switch( choice )
               {
               case 1:printf("Enter an item to be inserted at the
front\n");
               while(1)
               {
                    if (! fgets (EnteredLine, LENGTH, stdin))
                    {
                         printf("Unable to retrive");
                         exit(2);
                    }
                    if(( item = *EnteredLine))
                    {
                         if( *(EnteredLine +1) != '\n')
                         {
                         printf("Entered Just a single character\n");
                         continue;
                         }
                         Head = I_F(&item , Head);
                    }
                    break;
               }
               break;

               case 2: printf("Enter an Item at rear\n");
               while(1)
               {
                    if( ! fgets( EnteredLine, LENGTH, stdin))
                    {
                         printf("Unable to retrive input\n");
                         exit(2);
                    }
                    if( item =* EnteredLine)
                    {
                         if( *(EnteredLine +1) != '\n')
                         {
                         printf("Entered Just a single character\n");
                         continue;
                         }
                         Head = I_R(&item , Head);
                    }
                    break;
               }
               break;

               case 3: Head = D_F( Head);
                    break;
               case 4: Head = D_R (Head);
                    break;
               case 5: while( Extract (&item, &Head))
                    {
                         printf("%c\t", item);
                    }
                    printf("\n");
                    break;
               }}
               else
               {
                    printf("Enter valid option\n");
               }
          }
          else
          {
               printf("Enter valid option\n");
          }

       }
}
```

5

Design Principles and Coding Techniques

Ask, “What Would the User Do?” (You Are Not the User)

We all tend to assume that other people think like us. But they don’t. Psychologists call this the *false consensus bias*. When people think or act differently from us, we’re quite likely to label them (subconsciously) as defective in some way.

This bias explains why programmers have such a hard time putting themselves in the users’ position. Users don’t think like programmers. For a start, they spend much less time using computers. They neither know nor care how a computer works. This means they can’t draw on any of the battery of problem-solving techniques so familiar to programmers. They don’t recognize the patterns and cues programmers use to work with, through, and around an interface. The best way to find out how a user thinks is to watch one. Ask a user to complete a task using a similar piece of software to what you’re developing. Make sure the task is a real one: “Add up a column of numbers” is OK; “Calculate your expenses for the last month” is better. Avoid tasks that are too specific, such as “Can you select these spreadsheet cells and enter a *SUM* formula below?”—there’s a big clue in that question. Get the user to talk through his or her progress. Don’t interrupt. Don’t try to help. Keep asking yourself, “Why is he doing that?” and “Why is she not doing that?” The first thing you’ll notice is that users do a core of things similarly. They try to complete tasks in the same order—and they make the same mistakes in the same places. You should design around that core behavior. This is different from design meetings, where people tend to listen when someone says, “What if the user wants to…?”

This leads to elaborate features and confusion over what users want. Watching users eliminates this confusion. You’ll see users getting stuck. When you get stuck, you look around. When users get stuck, they narrow their focus. It becomes harder for them to see solutions elsewhere on the screen. It’s one reason why help text is a poor solution to poor user interface design. If you must have instructions or help text, make sure to locate it right next to your problem areas. A user’s narrow focus of attention is why tool tips are more useful than help menus. Users tend to muddle through. They’ll find a way that works and stick with it, no matter how convoluted. It’s better to provide one really obvious way of doing things than two or three shortcuts. You’ll also find that there’s a gap between what users say they want and what they actually do. That’s worrying, as the normal way of gathering user requirements is to ask them. It’s why the best way to capture requirements is to watch users. Spending an hour watching users is more informative than spending a day guessing what they want.

Beauty Is in Simplicity! Don’t Just Learn the Language, Understand Its Culture

There is one quote , from Plato, that I think is particularly good for all software developers to know and keep close to their hearts: Beauty of style and harmony and grace and good rhythm depends on simplicity. In one sentence, this sums up the values that we as software developers should aspire to.

There are a number of things we strive for in our code:

- Readability
- Maintainability
- Speed of development
- The elusive quality of beauty

Plato is telling us that the enabling factor for all of these qualities is simplicity. What is beautiful code? This is potentially a very subjective question. Perception of beauty depends heavily on individual background, just as much of our perception of anything depends on our background. People educated in the arts have a different perception of (or at least approach to) beauty than people educated in the sciences. Arts majors tend to approach beauty in software by comparing software to works of art, while science majors tend to talk about symmetry and the golden ratio, trying to reduce things to formulae. In my experience, simplicity is the foundation of most of the arguments from both sides.

Think about source code that you have studied. If you haven’t spent time studying other people’s code, stop reading this right now and find some open source code to study. Seriously! I mean it! Go search the Web for some code in your language of choice, written by some well-known, acknowledged expert.

You’re back? Good. Where were we? Ah, yes…I have found that code that resonates with me, and that I consider beautiful, has a number of properties in common. Chief among these is simplicity. I find that no matter how complex the total application or system is, the individual parts have to be kept simple: simple objects with a single responsibility containing similarly simple, focused methods with descriptive names. Some people think the idea of having short methods of 5–10 lines of code is extreme, and some languages make it very hard to do, but I think that such brevity is a desirable goal nonetheless. The bottom line is that beautiful code is simple code. Each individual part is kept simple with simple responsibilities and simple relationships with the other parts of the system. This is the way we can keep our systems maintainable over time, with clean, simple, testable code, ensuring a high speed of development throughout the lifetime of the system. Beauty is born of and found in simplicity.

Domain-Specific Languages

Whenever you listen to a discuss ion by experts in any domain, be it chess players, kindergarten teachers, or insurance agents, you'll notice that their vocabulary is quite different from everyday language. That's part of what domain-specific languages (DSLs) are about: a specific domain has a specialized vocabulary to describe the things that are particular to that domain.

In the world of software, DSLs are about executable expressions in a language specific to a domain, employing a limited vocabulary and grammar that is readable, understandable, and—hopefully—writable by domain experts. DSLs targeted at software developers or scientists have been around for a long time. The Unix "little languages" found in configuration files and the languages created with the power of LISP macros are some of the older examples. DSLs are commonly classified as either internal or external:

Internal DSLs

Internal DSLs are written in a general-purpose programming language whose syntax has been bent to look much more like natural language. This is easier for languages that offer more syntactic sugar and formatting possibilities (e.g., Ruby and Scala) than it is for others that do not (e.g., Java). Most internal DSLs wrap existing APIs, libraries, or business code and provide a wrapper for less mind-bending access to the functionality.

They are directly executable by just running them. Depending on the implementation and the domain, they are used to build data structures, define dependencies,\ run processes or tasks, communicate with other systems, or validate user input. The syntax of an internal DSL is constrained by the host language. There are many patterns—e.g., expression builder, method chaining, and annotation—that can help you to bend the host language to your DSL.

If the host language doesn't require recompilation, an internal DSL can be developed quite quickly working side by side with a domain expert.

External DSLs

External DSLs are textual or graphical expressions of the language—although textual DSLs tend to be more common than graphical ones. Textual expressions can be processed by a toolchain that includes lexer, parser, model transformer, generators, and any other type of post-processing. External DSLs are mostly read into internal models that form the basis for further processing. It is helpful to define a grammar (e.g., in EBNF).

A grammar provides the starting point for generating parts of the toolchain (e.g., editor, visualizer, parser generator). For simple DSLs, a handmade parser may be sufficient—using, for instance, regular expressions. Custom parsers can become unwieldy if too much is asked of them, so it makes sense to look at tools designed specifically for working with language grammars and DSLs—e.g., openArchitectureWare, ANTLR, SableCC, AndroMDA.

Defining external DSLs as XML dialects is also quite common, although readability is often an issue—especially for nontechnical readers.

You must always take the target audience of your DSL into account. Are they developers, managers, business customers, or end users? You have to adapt the technical level of the language, the available tools, syntax help (e.g., IntelliSense), early validation, visualization, and representation to the intended audience.

By hiding technical details, DSLs can empower users by giving them the ability to adapt systems to their needs without requiring the help of developers. It can also speed up development because of the potential distribution of work after the initial language framework is in place. The language can be evolved gradually. There are also different migration paths for existing expressions and grammars available.

Bugs and Fixes

Check Your Code First Before Looking to Blame Others

Developers —all of us !—often have trouble believing our own code is broken. It is just so improbable that, for once, it must be the compiler that’s broken. Yet, in truth, it is very (very) unusual that code is broken by a bug in the compiler, interpreter, OS, app server, database, memory manager, or any other piece of system software. Yes, these bugs exist, but they are far less common than we might like to believe. I once had a genuine problem with a compiler bug optimizing away a loop variable, but I have imagined my compiler or OS had a bug many more times. I have wasted a lot of my time, support time, and management time in the process, only to feel a little foolish each time it turned out to be my mistake after all. Assuming that the tools are widely used, mature, and employed in various technology stacks, there is little reason to doubt the quality. Of course, if the tool is an early release, or used by only a few people worldwide, or a piece of seldom downloaded, version 0.1, open source software, there may be good reason to suspect the software. (Equally, an alpha version of commercial software might be suspect.)

Given how rare compiler bugs are, you are far better putting your time and energy into finding the error in your code than into proving that the compiler is wrong. All the usual debugging advice applies, so isolate the problem, stub out calls, and surround it with tests; check calling conventions, shared libraries, and version numbers; explain it to someone else; look out for stack corruption and variable type mismatches; and try the code on different machines and different build configurations, such as debug and release. Question your own assumptions and the assumptions of others. Tools from different vendors might have different assumptions built into them—so too might different tools from the same vendor. When someone else is reporting a problem you cannot duplicate, go and see what they are doing. They may be doing something you never thought of or are doing something in a different order. My personal rule is that if I have a bug I can’t pin down, and I’m starting to think it’s the compiler, then it’s time to look for stack corruption. This is especially true if adding trace code makes the problem move around.

Multithreaded problems are another source of bugs that turn hair gray and induce screaming at the machine. All the recommendations to favor simple code are multiplied when a system is multithreaded. Debugging and unit tests cannot be relied on to find such bugs with any consistency, so simplicity of design is paramount. So, before you rush to blame the compiler, remember Sherlock Holmes’s advice, “Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth,” and opt for it over Dirk Gently’s, “Once you eliminate the improbable, whatever remains, no matter how impossible, must be the truth.”

Reinvent the Wheel Often

Just use something that exists—it's silly to reinvent the wheel….

Have you ever heard this or some var iat ion thereof? Sure you have! Every developer and student probably hears comments like this frequently. Why, though? Why is reinventing the wheel so frowned upon? Because, more often than not, existing code is working code. It has already gone through some sort of quality control and rigorous testing, and is being used successfully. Additionally, the time and effort invested in reinvention are unlikely to pay off as well as using an existing product or codebase. Should you bother reinventing the wheel? Why? When? Perhaps you have seen publications about patterns in software development, or books on software design. These books can be sleepers, regardless of how wonderful the information contained in them is. The same way that watching a movie about sailing is very different from going sailing, so too is using existing code versus designing your own software from the ground up, testing it, breaking it, repairing it, and improving it along the way. Reinventing the wheel is not just an exercise in where to place code constructs: it is about how to get an intimate knowledge of the inner workings of various components that already exist. Do you know how memory managers work? Virtual paging? Could you implement these yourself? How about double-linked lists? Dynamic array classes? ODBC clients? Could you write a graphical user interface that works like a popular one you know and like? Can you create your own web-browser widgets? Do you know when to write a multiplexed system versus a multithreaded one? How to decide between a file- or a memory-based database?

Most developers simply have never created these types of core software implementations themselves and therefore do not have an intimate knowledge of how they work. The consequence is that all these kinds of software are viewed as mysterious black boxes that just work. Understanding only the surface of the water is not enough to reveal the hidden dangers beneath. Not knowing the deeper things in software development will limit your ability to create stellar work. Reinventing the wheel and getting it wrong is more valuable than nailing it first time. There are lessons learned from trial and error that have an emotional component to them that reading a technical book alone just cannot deliver!

Learned facts and book smarts are crucial, but becoming a great programmer is as much about acquiring experience as it is about collecting facts. Reinventing the wheel is as important to a developer's education and skill as weightlifting is to a body builder.

6

Know your customers Your Customers
Do Not Mean What They Say

I've never met a customer yet _that wasn't all too happy to tell me whatnthey wanted—usually in great detail. The problem is that customers don't always tell you the whole truth. They generally don't lie, but they speak in customer speak, not developer speak. They use their terms and their contexts. They leave out significant details. They make assumptions that you've been at their company for 20 years, just like they have. This is compounded by the fact that many customers don't actually know what they want in the first place! Some may have a grasp of the "big picture," but they are rarely able to communicate the details of their vision effectively. Others might be a little lighter on the complete vision, but they know what they don't want. So, how can you possibly deliver a software project to someone who isn't telling you the whole truth about what they want? It's fairly simple. Just interact with them more.

Challenge your customers early, and challenge them often. Don't simply restate what they told you they wanted in their words. Remember: they didn't mean what they told you. I often implement this advice by swapping out the customer's words in conversation with them and judging their reaction. You'd be amazed how many times the term *customer* has a completely different meaning from the term *client*. Yet the guy telling you what he wants in his software project will use the terms interchangeably and expect you to keep track as to which one he's talking about. You'll get confused, and the software you write will suffer.

Discuss topics numerous times with your customers before you decide that you understand what they need. Try restating the problem two or three times with them. Talk to them about the things that happen just before or just after the topic you're talking about to get better context. If at all possible, have multiple people tell you about the same topic in separate conversations. They will almost always tell you different stories, which will uncover separate yet related facts. Two people telling you about the same topic will often contradict each other. Your best chance for success is to hash out the differences before you start your ultra-complex software crafting. Use visual aids in your conversations. This could be as simple as using a whiteboard in a meeting, as easy as creating a visual mockup early in the design phase, or as complex as crafting a functional prototype. It is generally known that using visual aids during a conversation helps lengthen our attention span and increases the retention rate of the information. Take advantage of this fact and set your project up for success. In a past life, I was a "multimedia programmer" on a team that produced glitzy projects. A client of ours described her thoughts on the look and feel of the project in great detail. The general color scheme discussed in the design meetings indicated a black background for the presentation. We thought we had it nailed. Teams of graphic designers began churning out hundreds of layered graphics files. Loads of time was spent molding the end product. On the day we showed the client the fruits of our labor, we got some startling news. When she saw the product, her exact words about the background color were, "When I said black, I meant white." So, you see, it is never as clear as black and white.

Know Your Next Commit

I tapped thre programers on their shoulders_ and asked what they were doing. “I am refactoring these methods,” the first answered. “I am adding some parameters to this web action,” the second answered. The third answered, “I am working on this user story.”

It might seem that the first two were engrossed in the details of their work, while only the third could see the bigger picture, and that he had the better focus. However, when I asked when and what they would commit, the picture changed dramatically. The first two were pretty clear about what files would be involved, and would be finished within an hour or so. The third programmer answered, “Oh, I guess I will be ready within a few days. I will probably add a few classes and might change those services in some way.” The first two did not lack a vision of the overall goal. They had selected tasks they thought led in a productive direction, and could be finished within a couple of hours. Once they had finished those tasks, they would select a new feature or refactoring to work on. All the code written was thus done with a clear purpose and a limited, achievable goal in mind. The third programmer had not been able to decompose the problem and was working on all aspects at once. He had no idea of what it would take, basically doing speculative programming, hoping to arrive at some point where he would be able to commit. Most probably, the code written at the start of this long session was poorly matched for the solution that came out in the end. What would the first two programmers do if their tasks took more than two hours? After realizing they had taken on too much, they would most likely throw away their changes, define smaller tasks, and start over. To keep working would have lacked focus and led to speculative code entering the repository.

Instead, changes would be thrown away, but the insights kept. The third programmer might keep on guessing and desperately try to patch together his changes into something that could be committed. After all, you cannot throw away code changes you have done—that would be wasted work, wouldn’t it? Unfortunately, not throwing the code away leads to slightly odd code that lacks a clear purpose entering the repository.

At some point, even the commit-focused programmers might fail to find something useful they thought could be finished in two hours. Then, they would go directly into speculative mode, playing around with the code and, of course, throwing away the changes whenever some insight led them back on track. Even these seemingly unstructured hacking sessions have purpose: to learn about the code to be able to define a task that would constitute a productive step.

Know your next commit. If you cannot finish, throw away your changes, then define a new task you believe in with the insights you have gained. Do speculative experimentation whenever needed, but do not let yourself slip into speculative mode without noticing. Do not commit guesswork into your repository.

7

Tools, Automation, and Development Environments

Know Your IDE

In the 1980s, our programming environments were typically nothing better than glorified text editors…if we were lucky. Syntax highlighting, which we take for granted nowadays, was a luxury that certainly was not available to everyone. Pretty printers to format our code nicely were usually external tools that had to be run to correct our spacing. Debuggers were also separate programs run to step through our code, but with a lot of cryptic keystrokes. During the 1990s, companies began to recognize the potential income that they could derive from equipping programmers with better and more useful tools. The Integrated Development Environment (IDE) combined the previous editing features with a compiler, debugger, pretty printer, and other tools. During that time, menus and the mouse also became popular, which meant that developers no longer needed to learn cryptic key combinations to use their editors. They could simply select their command from the menu.

In the 21st century, IDEs have become so commonplace that they are given away for free by companies wishing to gain market share in other areas. The modern IDE is equipped with an amazing array of features. My favorite is automated refactoring, particularly *Extract Method*, where I can select and convert a chunk of code into a method. The refactoring tool will pick up all the parameters that need to be passed into the method, which makes it extremely easy to modify code.

My IDE will even detect other chunks of code that could also be replaced by this method and ask me whether I would like to replace them, too. Another amazing feature of modern IDEs is the ability to enforce style rules within a company. For example, in Java, some programmers have started making all parameters final (which, in my opinion, is a waste of time). However, since they have such a style rule, all I would need to do to follow it is set it up in my IDE: I would get a warning for any non-final parameter. Style rules can also be used to find probable bugs, such as comparing autoboxed objects for reference equality, e.g., using == on primitive values that are autoboxed into reference objects. Unfortunately, modern IDEs do not require us to invest effort to learn how to use them. When I first programmed C on Unix, I had to spend quite a bit of time learning how the vi editor worked, due to its steep learning curve.

This time spent up front paid off handsomely over the years. I am even typing the draft of this article with vi. Modern IDEs have a very gradual learning curve, which can have the effect that we never progress beyond the most basic usage of the tool.\ My first step in learning an IDE is to memorize the keyboard shortcuts. Since my fingers are on the keyboard when I'm typing my code, pressing *Ctrl+Shift+I* to inline a variable prevents breaking the flow, whereas switching to navigate a menu with my mouse interrupts it. These interruptions lead to unnecessary context switches, making me much less productive if I try to do everything the lazy way. The same rule also applies to keyboard skills: learn to touch type; you won't regret the time invested up front.

Lastly, as programmers we have time-proven Unix streaming tools that can help us manipulate our code. For example, if during a code review, I noticed that the programmers had named lots of classes the same, I could find these very easily using the tools find, sed, sort, uniq, and grep, like this:
find . -name "*.java" | sed 's/.*\///' | sort | uniq -c | grep -v "^ *1 " | sort –r We expect a plumber coming to our house to be able to use his blowtorch. Let’s spend a bit of time to study how to become more effective with our IDE.

You Got to Care about the Code

It doesn't take Sherlock Holmes _to work out that good programmers write good code. Bad programmers…don't. They produce monstrosities that the rest of us have to clean up. You want to write the good stuff, right? You want to be a good programmer.

Good code doesn't pop out of thin air. It isn't something that happens by luck when the planets align. To get good code, you have to work at it. Hard. And you'll only get good code if you actually *care* about good code. Good programming is not born from mere technical competence. I've seen highly intellectual programmers who can produce intense and impressive algorithms, who know their language standard by heart, but who write the most awful code. It's painful to read, painful to use, and painful to modify. I've seen more humble programmers who stick to very simple code, but who write elegant and expressive programs that are a joy to work with. Based on my years of experience in the software factory, I've concluded that the real difference between adequate programmers and great programmers is this: *attitude*.

Good programming lies in taking a professional approach, and wanting to write the best software you can, within the real-world constraints and pressures of the software factory. *The code to hell is paved with good intentions*. To be an excellent programmer, you have to rise above good intentions, and actually *care* about the code—foster positive perspectives and develop healthy attitudes. Great code is carefully crafted by master artisans, not thoughtlessly hacked out by sloppy programmers or erected mysteriously by self-professed coding gurus. You want to write good code. You want to be a good programmer.

So, you care about the code:

- In any coding situation, you refuse to hack something that only *seems* to work. You strive to craft elegant code that is clearly correct (and has good tests to show that it is correct).

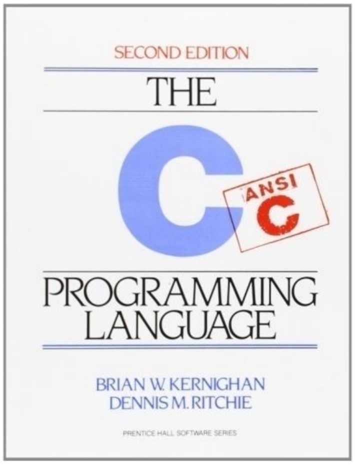
- You write code that is *discoverable* (that other programmers can easily pick up and understand), that is *maintainable* (that you, or other programmers, will be easily able to modify in the future), and that is *correct* (you take all steps possible to determine that you *have* solved the problem, not just made it look like the program works).

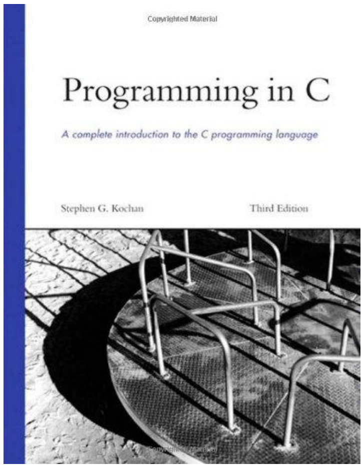
- You work well alongside other programmers. No programmer is an island. Few programmers work alone; most work in a team of programmers, either in a company environment or on an open source project. You consider other programmers and construct code that others can read. You want the team to write the best software possible, rather than to make yourself look clever.

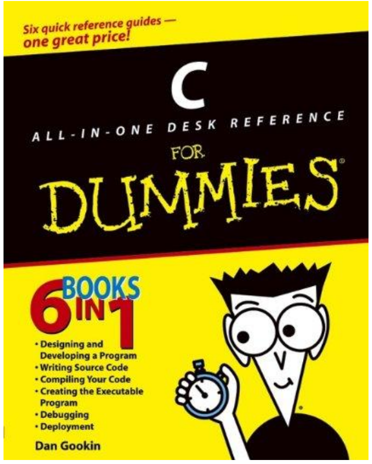
- Any time you touch a piece of code, you strive to leave it better than you found it (either better structured, better tested, more understandable…).

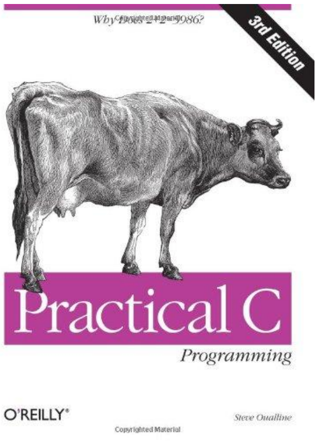
- You care about code and about programming, so you are constantly learning new languages, idioms, and techniques. But you apply them only when appropriate. Fortunately, you're reading this collection of advice because you do care about code. It interests you. It's your passion. Have fun programming. Enjoy cutting code to solve tricky problems. Produce software that makes you proud.

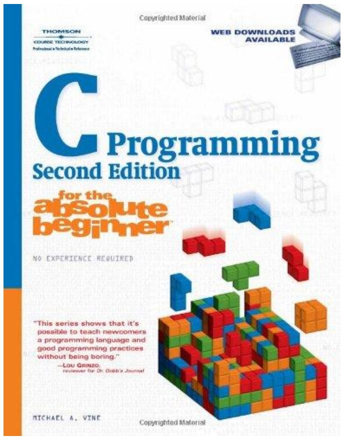
References

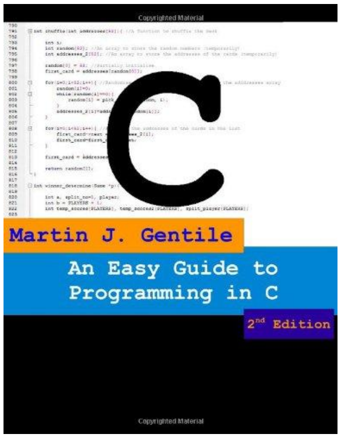


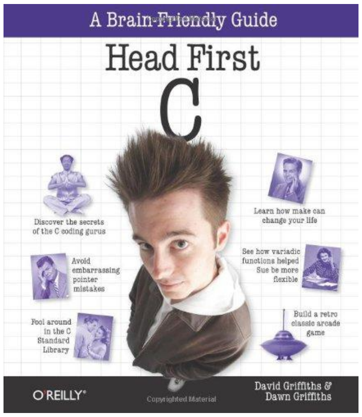


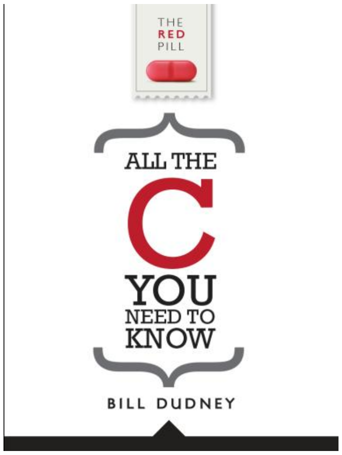

Six quick reference guides — one great price!
C
ALL-IN-ONE DESK REFERENCE
FOR DUMMIES
6 BOOKS IN 1
• Designing and Developing a Program
• Writing Source Code
• Compiling Your Code
• Creating the Executable Program
• Debugging
• Deployment
Dan Gookin

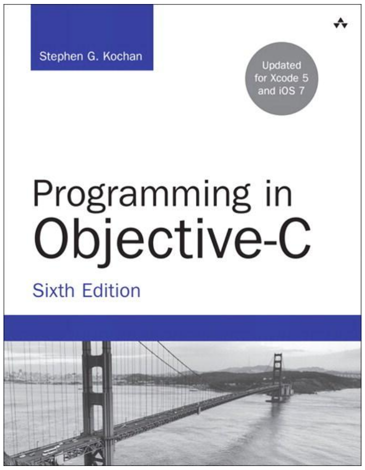



THOMSON COURSE TECHNOLOGY
Copyrighted Material
WEB DOWNLOADS AVAILABLE
C Programming
Second Edition
for the absolute beginner
NO EXPERIENCE REQUIRED
"This series shows that it's possible to teach newcomers a programming language and good programming practices without being boring."
—Lou Grinzo, reviewer for Dr. Dobb's Journal
MICHAEL A. VINE
Copyrighted Material




A Brain-Friendly Guide
Head First C
Discover the secrets of the C coding gurus
Learn how make can change your life
Avoid embarrassing pointer mistakes
See how variadic functions helped Sue be more flexible
Fool around in the C Standard Library
Build a retro classic arcade game
O'REILLY
Copyrighted Material
David Griffiths & Dawn Griffiths





http://cm.bell-labs.com/cm/cs/who/dmr/chist.html
http://www.eskimo.com/~scs/cclass/cclass.html

QUICK ACCESS NOTES

QUICK ACCESS NOTES