COMPUTER CONCEPTS AND PROGRAMMING IN C

MCA 103

SELF LEARNING MATERIAL

DIRECTORATE

OF DISTANCE EDUCATION

SWAMI VIVEKANAND SUBHARTI UNIVERSITY

MEERUT – 250 005,

UTTAR PRADESH (INDIA)

COMPUTER CONCEPTS AND PROGRAMMING IN C (MCA-103)

UNIT 1:

Introduction to any Operating System [Unix, Linux, Windows], Programming Environment, Write and Execute the first program, Introduction to the Digital Computer; Concept of an algorithm; termination and correctness. Algorithms to programs: specification, top-down development and stepwise refinement. Introduction to Programming, Use of high level programming language for the systematic development of programs. Introduction to the design and implementation of correct, efficient and maintainable programs, Structured Programming, Trace an algorithm to depict the logic, Number Systems and conversion methods

UNIT 2:

Standard I/O in "C", Fundamental Data Types and Storage Classes: Character types, Integer, short, long, unsigned, single and double-precision floating point, storage classes, automatic, register, static and external, Operators and Expressions: Using numeric and relational operators, mixed operands and type conversion, Logical operators, Bit operations, Operator precedence and associativity,

UNIT 3:

Conditional Program Execution: Applying if and switch statements, nesting if and else, restrictions on switch values, use of break and default with switch, Program Loops and Iteration: Uses of while, do and for loops, multiple loop variables, assignment operators, using break and continue, Modular Programming**:** Passing arguments by value, scope rules and global variables, separate compilation, and linkage, building your own modules.

UNIT 4:

Arrays: Array notation and representation, manipulating array elements, using multidimensional arrays, arrays of unknown or varying size, Structures: Purpose and usage of structures, declaring structures, assigning of structures, Pointers to Objects: Pointer and address arithmetic, pointer operations and declarations, using pointers as function arguments, Dynamic memory allocation, defining and using stacks and linked lists.

UNIT 5:

Sequential search, Sorting arrays, Strings, Text files, The Standard C Preprocessor: Defining and calling macros, utilizing conditional compilation, passing values to the compiler, The Standard C Library: Input/Output : fopen, fread, etc, string handling functions, Math functions : log, sin, alike Other Standard C functions.

UNIT 1:

Introduction to any Operating System [Unix, Linux, Windows]

Unix:- UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available, for example, in a telnet session.

Types of UNIX

There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, and MacOS X.

Here in the School, we use Solaris on our servers and workstations, and Fedora Linux on the servers and desktop PCs.

The UNIX operating system

The UNIX operating system is made up of three parts; the kernel, the shell and the programs.

The kernel

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the filestore and communications in response to system calls.

As an illustration of the way that the shell and the kernel work together, suppose a user types rm myfile (which has the effect of removing the file myfile). The shell searches the filestore for the file containing the program rm, and then requests the kernel, through system calls, to execute the program rm on myfile. When the process rm

myfile has finished running, the shell then returns the UNIX prompt % to the user, indicating that it is waiting for further commands.

The shell

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems).

The adept user can customise his/her own shell, and users can use different shells on the same machine. Staff and students in the school have the **tcsh shell** by default.

The tcsh shell has certain features to help the user inputting commands.

Filename Completion - By typing part of the name of a command, filename or directory and pressing the [Tab] key, the tcsh shell will complete the rest of the name automatically. If the shell finds more than one name beginning with those letters you have typed, it will beep, prompting you to type a few more letters before pressing the tab key again.

History - The shell keeps a list of the commands you have typed in. If you need to repeat a command, use the cursor keys to scroll up and down the list or type history for a list of previous commands.

Files and processes

Everything in UNIX is either a file or a process.

A process is an executing program identified by a unique PID (process identifier).

A file is a collection of data. They are created by users using text editors, running compilers etc.

Examples of files:

- a document (report, essay etc.)

- the text of a program written in some high-level programming language

- instructions comprehensible directly to the machine and incomprehensible to a casual user, for example, a collection of binary digits (an executable or binary file);

- a directory, containing information about its contents, which may be a mixture of other directories (subdirectories) and ordinary files.

The Directory Structure

All the files are grouped together in the directory structure. The file-system is arranged in a hierarchical structure, like an inverted tree. The top of the hierarchy is traditionally called root (written as a slash /)
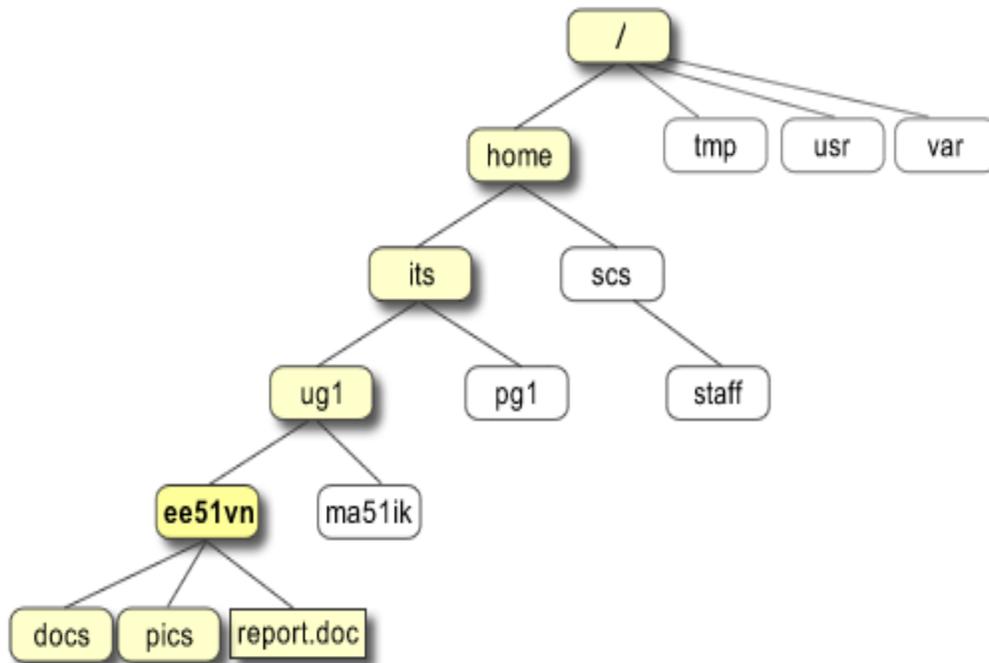
In the diagram above, we see that the home directory of the undergraduate student "ee51vn" contains two sub-directories (docs and pics) and a file called report.doc.

The full path to the file report.doc is "/home/its/ug1/ee51vn/report.doc"

Starting an UNIX terminal

To open an UNIX terminal window, click on the "Terminal" icon from Applications/Accessories menus.
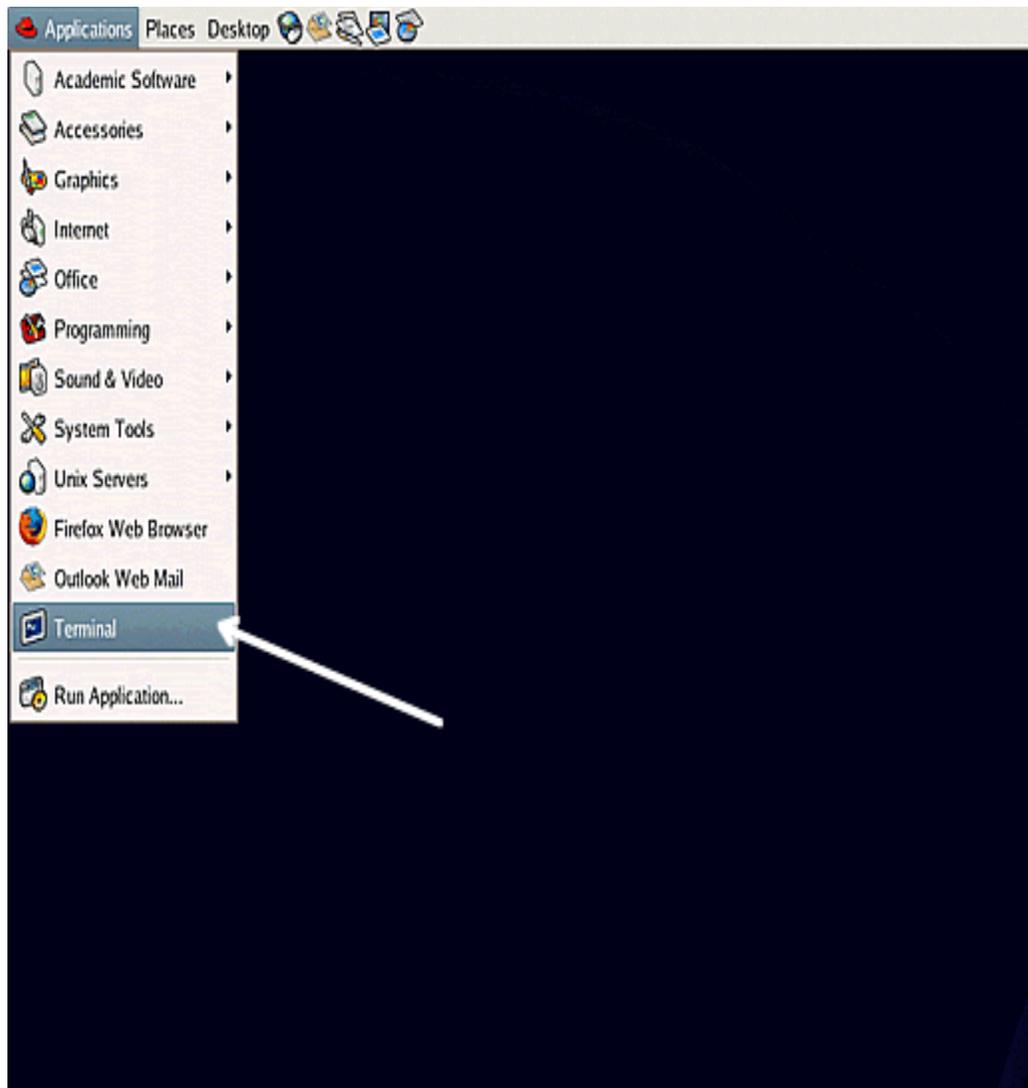
An UNIX Terminal window will then appear with a % prompt, waiting for you to start entering commands.
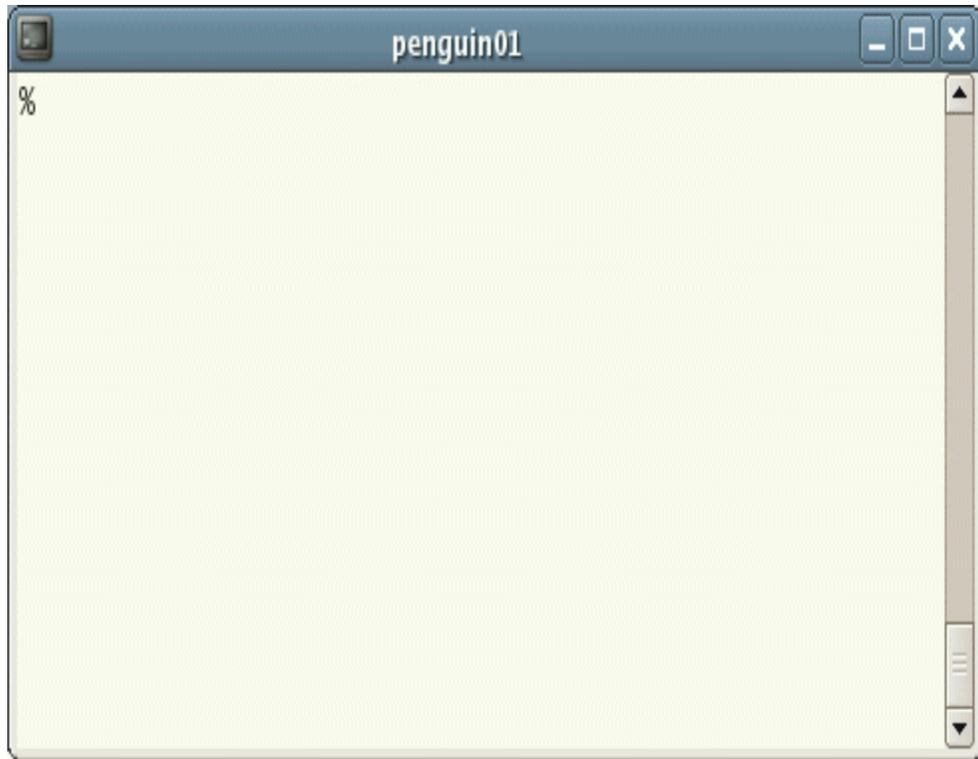
Linux

LINUX is an operating system or a kernel distributed under an open-source license. Its functionality list is quite like UNIX. The kernel is a program at the heart of the Linux operating system that takes care of fundamental stuff, like letting hardware communicate with software.
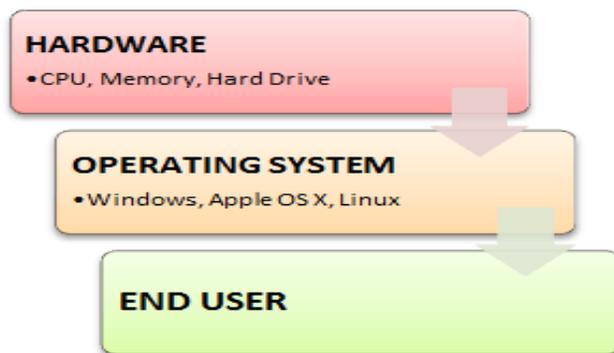
Why do you need an OS?

Every time you switch on your computer, you see a screen where you can perform different activities like write, browse the internet or watch a video. What is it that makes

the computer hardware work like that? How does the processor on your computer know that you are asking it to run a mp3 file?

Well, it is the operating system or the kernel which does this work. So, to work on your computer, you need an Operating System(OS). In fact, you are using one as you read this on your computer. Now, you may have used popular OS's like Windows, Apple OS X, but here we will learn introduction to Linux operating system, Linux overview and what benefits it offers over other OS choices.

Who created Linux?

Linux is an operating system or a kernel which germinated as an idea in the mind of young and bright Linus Torvalds when he was a computer science student. He used to work on the UNIX OS (proprietary software) and thought that it needed improvements.

However, when his suggestions were rejected by the designers of UNIX, he thought of launching an OS which will be receptive to changes, modifications suggested by its users**.**

The Lone Kernel & the early days

So Linus devised a Kernel named Linux in 1991. Though he would need programs like File Manager, Document Editors, Audio -Video programs to run on it. Something as you have a cone but no ice-cream on top.

As time passed by, he collaborated with other programmers in places like MIT and applications for Linux started to appear. So around 1991, a working Linux operating system with some applications was officially launched, and this was the start of one of the most loved and open-source OS options available today.

The earlier versions of Linux OS were not so user-friendly as they were in use by computer programmers and Linus Torvalds never had it in mind to commercialize his product**.**

This definitely curbed the Linux's popularity as other commercially oriented Operating System Windows got famous. Nonetheless, the open-source aspect of the Linux operating system made it more robust.

Linux gets its due attention

The main advantage of Linux was that programmers were able to use the Linux Kernel to design their own custom operating systems. With time, a new range of user-friendly OS's stormed the computer world. Now, Linux is one of the most popular and widely used Kernel**,** and it is the backbone of popular operating systems **like** Debian, Knoppix, Ubuntu, and Fedora**.** Nevertheless, the list does not end here as there are thousands of Best versions of Linux OS based on the Linux Kernel available which offer a variety of functions to the users.

Linux Kernel is normally used in combination of GNU project by Dr. Richard Stallman. All mordern distributions of Linux are actually distributions of Linux/GNU

The benefits of using Linux

Linux OS now enjoys popularity at its prime, and it's famous among programmers as well as regular computer users around the world. Its main benefits are -

It offers a **free operating system**. You do not have to shell hundreds of dollars to get the OS like Windows!
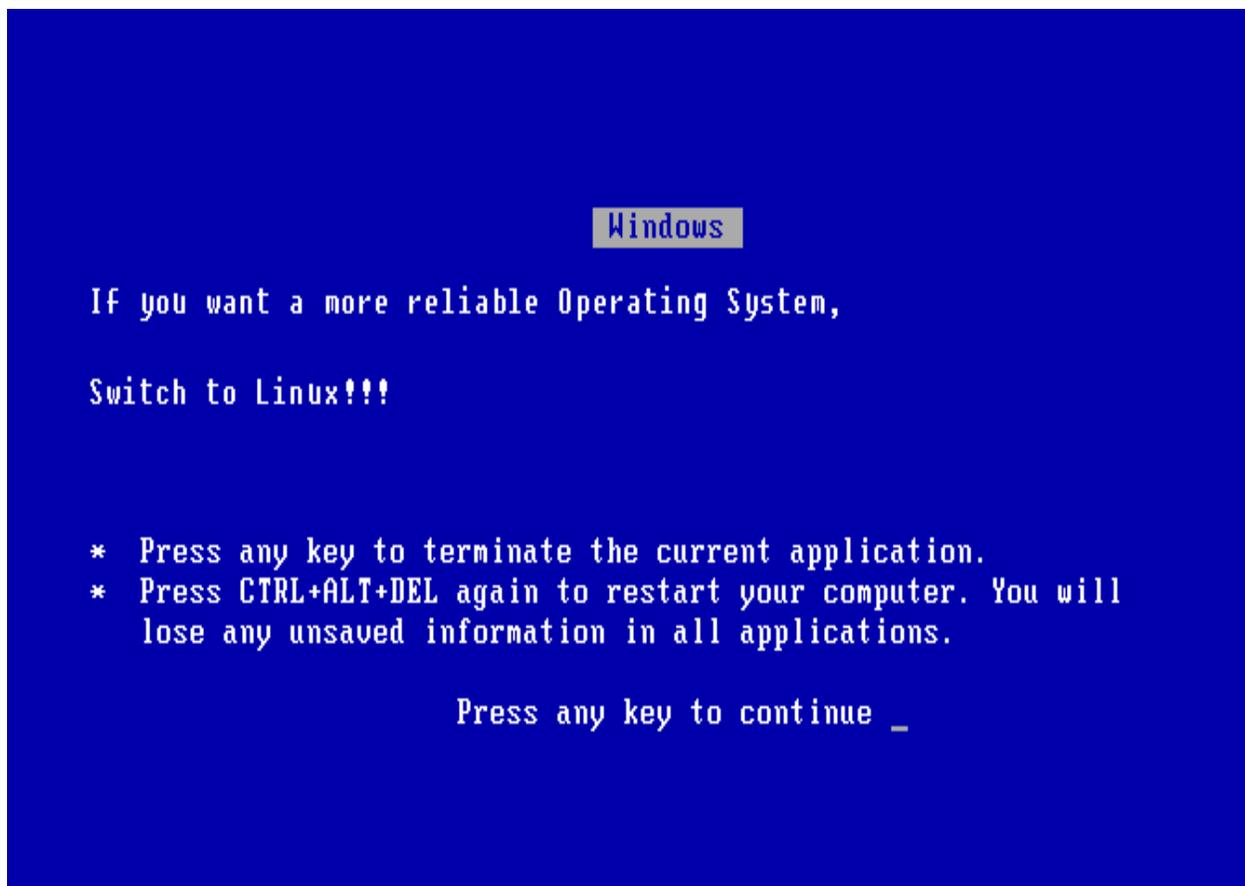
- Being open-source, anyone with programming knowledge can modify it.

- It is easy to learn Linux for beginners

- The Linux operating systems now offer millions of programs/applications and Linux softwares to choose from, most of them are free!

- Once you have Linux installed you no longer need an antivirus! Linux is a highly secure system. More so, there is a global development community constantly looking at ways to enhance its security. With each upgrade, the OS becomes more secure and robust

- Linux freeware is the OS of choice for Server environments due to its stability and reliability (Mega-companies like Amazon, Facebook, and Google use Linux for their Servers). A Linux based server could run non-stop without a reboot for years on end.

- Users, who are new to Linux, usually shun it by falsely considering it as a difficult and technical OS to operate but, to state the truth, in the last few years Linux operating systems have become a lot more user-friendly than their counterparts like Windows, so trying them is the best way to know whether Linux suits you or not.

- There are thousands of Best Linux OSs and Linux softwares available based on the Linux Kernel; most of them offer state-of-the-art security and applications, all of it for free!

- This is what Linux is all about, and now we will move on to how to install Linux and which Distribution you should choose.

- I am asked to Learn Unix? Then why Linux?

- UNIX is called the mother of operating systems which laid out the foundation to Linux. Unix is designed mainly for mainframes and is in enterprises and universities. While Linux is fast becoming a household name for computer users, developers, and server environment. You may have to pay for a Unix kernel while in Linux it is free.

- But, the commands used on both the operating systems are usually the same. There is not much difference between UNIX and Linux. Though they might seem different, at the core, they are essentially the same. Since Linux is a clone of UNIX. So learning one is same as learning another.

Windows

The oldest of all Microsoft's operating systems is MS-DOS (Microsoft Disk Operating System). MS-DOS is a text-based operating system. Users have to type commands rather than use the more friendly graphical user interfaces (GUI's) available today. Despite its very basic appearance, MS-DOS is a very powerful operating system. There are many advanced applications and games available for MS-DOS. A version of MS-DOS underpins Windows. Many advanced administration tasks in Windows can only be performed using MS-DOS.

The history of Microsoft Windows dates back to 1985, when Microsoft released Microsoft Windows Version 1.01. Microsoft's aim was to provide a friendly user-interface known as a GUI (graphical user interface) which allowed for easier navigation of the system features. Windows 1.01 never really caught on. (The amazing thing about Windows 1.01 is that it fitted on a single floppy disk!). In 1987 Microsoft revamped the operating system and released Windows 2.03.   The GUI was very slightly improved but still looked too similar to Windows 1.01. The operating system again failed to capture a wide audience.

Microsoft made an enormous impression with Windows 3.0 and 3.1. Graphics and functionality were drastically improved. The Windows 3.x family provided multimedia capabilities as well as vastly improved graphics and application support.

Building on the success of Windows 3.x, Microsoft released Microsoft Windows For Workgroups 3.11. This gave Windows the ability to function on a network. It is not uncommon to find companies still using Windows 3.11.

In 1993 Microsoft divided the operating system into two categories; Business and home user. Windows NT (New Technology) was a lot more reliable than Windows 3.x. Windows NT provided advanced network features. On the business front, Windows NT continued to develop with the release of version 3.51. Different versions were provided which offered different functionality. Server provided the higher network functions and Workstation was mainly for the client machines.

In 1995 Windows went through a major revamp and Microsoft Windows 95 was released. This provided greatly improved multimedia and a much more polished user interface. The now familiar desktop and Start Menu appeared. Internet and networking support was built in Although Windows 95 was a home user operating system, it proved to be very popular in schools and businesses.

After the success of Windows 95, Microsoft improved the GUI interface of Windows NT and released Windows NT 4.0. NT4 could be tailored to the size of the business, NT4 Server for small to medium sized businesses and Enterprise Server for larger networks. Microsoft continued to improve the Windows format. Although Microsoft Windows 98 was very similar to Windows 95, it offered a much tidier display and enhanced multimedia support.

Breaking with its own naming conventions, Microsoft released Windows 2000 (initially called NT 5.0) for the business market. It appeared in 4 models: Professional -which

replaced Workstation, Server, Advanced Server and Datacenter Server catered for differing business requirements.

Although Windows 2000 had a greatly improved user interface, the best of the enhancements appeared on the server side. Active Directory was introduced which allowed much greater control of security and organisation. Improvements to the overall operating system allowed for easier configuration and installation.

One big advantage of Windows 2000 was that operating system settings could be modified easily without the need to restart the machine. Windows 2000 proved to be a very stable operating system that offered enhanced security and ease of administration.

The last incarnation of the Windows 9x family was Windows Millennium Edition (ME). There were many different versions of Windows floating around at this stage that Microsoft decided the next release of Windows would consolidate both the business and home versions. Although Windows ME was visually similar to Windows 2000. Windows ME was based on the Windows 9x line. Windows 9x/ME systems are not as secure and stable as Windows NT and 2000 systems.

Because of the stability of Windows NT/2000, Microsoft decided to end the development of the Windows 9x line, and merge both the consumer and business products. Microsoft Windows XP comes as the Home Edition and Professional, each is based on Windows 2000. Windows 2000 Server has been upgraded to Windows 2003. This appears in four variants: Web Server, Standard Server, Enterprise Server and Datacenter Server, each fulfilling a different business role. Windows XP has a very polished look, but the overall functionality is very similar to Windows 2000.

Programming Environment

The GT.M Programming Environment is described in the following sections.

Managing Data

The scope of M data is either process local or global.

- Local variables last only for the duration of the current session; GT.M deletes them when the M process terminates.

- Global variables contain data that persists beyond the process. GT.M stores global variables on disk. A Global Directory organizes global variables and describes the organization of a database. The GT.M administrator uses the Global Directory Editor (GDE) to create and manage Global Directories. A Global Directory maps global names to a database file. GT.M uses this mapping when it stores and retrieves globals from the database. Several Global Directories may refer to a single database file.

For more information about the GT.M data management system, refer to the "Global Directory Editor", "MUPIP" and "GT.M Journaling" chapters in the GT.M Administration and Operations Guide.

Database Management Utilities

The Global Directory Editor (GDE) creates, modifies, maintains, and displays the characteristics of Global Directories. GDE also maps LOCKs on resource names to the region of the database specified for the corresponding global variables.

The M Peripheral Interchange Program (MUPIP) creates database files and provides tools for GT.M database management and database journaling.

For more information regarding GT.M database utilities, refer to the "Global Directory Editor", "MUPIP" and "GT.M Journaling" chapters in the GT.M Administration and Operations Guide.

Managing Source Code

In the GT.M programming environment, source routines are generated and stored as standard UNIX files. They are created and edited with standard UNIX text editors. GT.M accepts source lines of up to 8192 bytes. When GT.M encounters a line with a length greater than 8192 bytes in a source file, it emits an LSEXPECTED warning. This warning identifies cases where a line greater than 8192 bytes is split into multiple lines, which causes statements beyond the character prior to the limit to execute irrespective of any starting IF, ELSE or FOR commands. The 8192 byte limit applies to XECUTE command arguments and Direct Mode input as well.

GT.M is designed to work with the operating system utilities and enhances them when beneficial. The following sections describe the process of programming and debugging with GT.M and from the operating system.

Source File Management

In addition to standard M "percent" utilities, GT.M permits the use of the standard UNIX file manipulation tools, for example, the diff, grep, cp, and mv commands. The GT.M programmer can also use the powerful facilities provided by the UNIX directory structure, such as time and date information, tree-structured directories, and file protection codes.

GT.M programs are compatible with most source management software, for example, RCS and SCCS.

Programming and Debugging Facilities

The GT.M programmer can use any UNIX text editor to create M source files. If you generate a program from within the Direct Mode, it also accesses the UNIX text editor specified by the environment variable EDITOR and provides additional capabilities to automate and enhance the process.

The GT.M programmer also uses the Direct Mode facility to interactively debug, modify, and execute M routines. In Direct Mode, GT.M executes each M command immediately, as if it had been in-line at the point where GT.M initiated Direct Mode.

The following is a list of additional enhancements available from the Direct Mode:

- The capability to issue commands from Direct Mode to the shell

- A command recall function to display and reuse previously entered commands

- Many language extensions that specifically optimize the debugging environment

The GT.M Compiler

The GT.M compiler operates on source files to produce object files consisting of position-independent, native object code, which on some platforms can be linked into shared object libraries. GT.M provides syntax error checking at compile-time and allows you to enable or disable the compile-as-written mode. By default, GT.M produces an object file even if the compiler detects errors in the source code. This compile-as-written mode facilitates a flexible approach to debugging.

The Run-Time System

A GT.M programmer can execute an M routine from the shell or interactively, using the M commands from Direct Mode.

The run-time system executes compile-as-written code as long as it does not encounter the compile-time errors. If it detects an error, the run-time system suspends execution of a routine immediately and transfers control to Direct Mode or to a user-written error routine.

Automatic and Incremental Linking

The run-time system utilizes a GT.M facility called ZLINK to link in an M routine. When a program or a Direct Mode command refers to an M routine that is not part of the current process, GT.M automatically uses the ZLINK facility and attempts to link the referenced routine (auto-ZLINK). The ZLINK facility also determines whether recompilation of the routine is necessary. When compiling as a result of a ZLINK, GT.M typically ignores errors in the source code.

The run-time system also provides incremental linking. The ZLINK command adds an M routine to the current image. This feature facilitates the addition of code modifications during a debugging session. The GT.M programmer can also use the feature to add patches and generated code to a running M process.

Error Processing

The GT.M compiler detects and reports syntax errors at the following times:

- Compile-time - while producing the object module from a source file

○ Run-time - while compiling code for M indirection and XECUTEs

○ Run-time - when the user is working in Direct Mode.

The compile-time error message format displays the line containing the error and the location of the error on the line. The error message also indicates what was incorrect about the M statement.

GT.M can not detect certain types of errors associated with indirection, the functioning of I/O devices, and program logic until run-time.

The compile-as-written feature allows compilation to continue and produces an object module despite errors in the code. This permits testing of other pathways through the code. The errors are reported at run-time, when GT.M encounters them in the execution path.

The GT.M run-time system recognizes execution errors and reports them when they occur. It also reports errors flagged by the compiler when they occur in the execution path.

For more information, see Chapter 13: "Error Processing".

Input-Output Processing

GT.M supports input and output processing with the following system components:

- Terminals

- Sequential disk files

- Mailboxes

- FIFOs

- Null devices

- Socket devices

GT.M input/output processing is device-independent. Copying information from one device to another is accomplished without reformatting.

GT.M has special terminal-handling facilities. GT.M performs combined QIO operations to enhance terminal performance. The terminal control facilities that GT.M provides include escape sequences, control character traps, and echo suppression.

GT.M supports RMS sequential disk files that are accessed using a variety of deviceparameters.

GT.M supports block I/O with fixed and variable length records for file-structured (FILES-11) tapes and non-file-structured unlabeled (FOREIGN) tapes. GT.M supports the ASCII character set for unlabeled FOREIGN and FILES-11 tapes. GT.M supports

the EBCDIC character set for FOREIGN tapes only. GT.M also supports FOREIGN DOS-11 and ANSI labelled tapes or stream format records. It also supports ASCII and EBCDIC character sets.

GT.M uses permanent or temporary mailboxes fifos for interprocess communication. GT.M treats mailboxes as record-structured I/O devices.

GT.M provides the ability to direct output to a null device. This is an efficient way to discard unwanted output.

GT.M provides device-exception processing so that I/O exception handling need not be combined with process-related exception conditions. The OPEN, USE, and CLOSE EXCEPTION parameters define an XECUTE string as an error handler for an I/O device.

Integrating GT.M with Other Languages

GT.M offers capabilities that allow you to optimize your programming environment. These include allowing you to call into M routines from programs written in other programming languages, access your M databases with interfaces that provide functionality equivalent to M intrinsic database functions, and to alter your programming environment when working with languages other than American English. These include allowing you to call programs written in other programming languages that support C-like interfaces and to alter your programming environment when working with languages other than American English. This capability is described in more detail in chapters throughout this manual.

Access to Non-M Routines

GT.M routines can call external (non-M) routines using the external call function. This permits access to functions implemented in other programming languages. For more information, see Chapter 11: "Integrating External Routines".

Internationalization

GT.M allows the definition of alternative collation sequences and pattern matching codes for use with languages other than English. Chapter 12: "Internationalization" describes the details and requirements of this functionality.

Write and Execute the first program

Node.js is a popular open-source runtime environment that can execute JavaScript outside of the browser using the V8 JavaScript engine, which is the same engine used to power the Google Chrome web browser's JavaScript execution. The Node runtime is commonly used to create command line tools and web servers.

Learning Node.js will allow you to write your front-end code and your back-end code in the same language. Using JavaScript throughout your entire stack can help reduce time for context switching, and libraries are more easily shared between your back-end server and front-end projects.

Also, thanks to its support for asynchronous execution, Node.js excels at I/O-intensive tasks, which is what makes it so suitable for the web. Real-time applications, like video streaming, or applications that continuously send and receive data, can run more efficiently when written in Node.js.

In this tutorial you'll create your first program with the Node.js runtime. You'll be introduced to a few Node-specific concepts and build your way up to create a program that helps users inspect environment variables on their system. To do this, you'll learn how to output strings to the console, receive input from the user, and access environment variables.

Prerequisites

To complete this tutorial, you will need:

- Node.js installed on your development machine. This tutorial uses Node.js version 10.16.0. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the "Installing Using a PPA" section of How To Install Node.js on Ubuntu 18.04.

- A basic knowledge of JavaScript, which you can find here: How To Code in JavaScript.

Step 1 — Outputting to the Console

To write a "Hello, World!" program, open up a command line text editor such as nano and create a new file:

- nano hello.js
-

Copy
With the text editor opened, enter the following code:

<div>

hello.js

```
console.log("Hello World");
```
Copy

</div>

The console object in Node.js provides simple methods to write to stdout, stderr, or to any other Node.js stream, which in most cases is the command line. The log method prints to the stdout stream, so you can see it in your console.

In the context of Node.js, streams are objects that can either receive data, like the stdout stream, or objects that can output data, like a network socket or a file. In the case of the stdout and stderr streams, any data sent to them will then be shown in the console. One of the great things about streams is that they're easily redirected, in which case you can redirect the output of your program to a file, for example.

Save and exit nano by pressing CTRL+X, when prompted to save the file, press Y. Now your program is ready to run.

Step 2 — Running the Program

To run this program, use the node command as follows:

* node hello.js
*
  Copy
  The hello.js program will execute and display the following output:

Output
 Hello World

The Node.js interpreter read the file and executed console.log("Hello World"); by calling the log method of the global console object. The string "Hello World" was passed as an argument to the log function.

Although quotation marks are necessary in the code to indicate that the text is a string, they are not printed to the screen.

Having confirmed that the program works, let's make it more interactive.

Step 3 — Receiving User Input via Command Line Arguments

Every time you run the Node.js "Hello, World!" program, it produces the same output. In order to make the program more dynamic, let's get input from the user and display it on the screen.

Command line tools often accept various arguments that modify their behavior. For example, running node with the --version argument prints the installed version instead of running the interpreter. In this step, you will make your code accept user input via command line arguments.

Create a new file arguments.js with nano:

- nano arguments.js
-

Copy
Enter the following code:

arguments.js
console.log(process.argv);

Copy
The process object is a global Node.js object that contains functions and data all related to the currently running Node.js process. The argv property is an array of strings containing all the command line arguments given to a program.

Save and exit nano by typing CTRL+X, when prompted to save the file, press Y.

Now when you run this program, you provide a command line argument like this:

- node arguments.js hello world
-

Copy
The output looks like the following:

Output
```
[ '/usr/bin/node',
  '/home/sammy/first-program/arguments.js',
  'hello',
  'world' ]
```
The first argument in the process.argv array is always the location of the Node.js binary that is running the program. The second argument is always the location of the file being run. The remaining arguments are what the user entered, in this case: hello and world.

We are mostly interested in the arguments that the user entered, not the default ones that Node.js provides. Open the arguments.js file for editing:

- nano arguments.js
-

Copy
Change console.log(process.arg); to the following:

arguments.js

```
console.log(process.argv.slice(2));
```
Copy

Because argv is an array, you can use JavaScript's built-in slice method that returns a selection of elements. When you provide the slice function with 2 as its argument, you get all the elements of argv that comes after its second element; that is, the arguments the user entered.

Re-run the program with the node command and the same arguments as last time:

- node arguments.js hello world
-

Copy

Now, the output looks like this:

Output

 ['hello', 'world']

Now that you can collect input from the user, let's collect input from the program's environment.

Step 4 — Accessing Environment Variables

Environment variables are key-value data stored outside of a program and provided by the OS. They are typically set by the system or user and are available to all running processes for configuration or state purposes. You can use Node's process object to access them.

Use nano to create a new file environment.js:

- nano environment.js
-

Copy

Add the following code:

environment.js

```
console.log(process.env);
```
Copy

The env object stores all the environment variables that are available when Node.js is running the program.

Save and exit like before, and run the environment.js file with the node command.

- node environment.js

-

Copy

Upon running the program, you should see output similar to the following:

Output

```
{ SHELL: '/bin/bash',
  SESSION_MANAGER:
   'local/digitalocean:@/tmp/.ICE-unix/1003,unix/digitalocean:/tmp/.ICE-unix/1003',
  COLORTERM: 'truecolor',
  SSH_AUTH_SOCK: '/run/user/1000/keyring/ssh',
  XMODIFIERS: '@im=ibus',
  DESKTOP_SESSION: 'ubuntu',
  SSH_AGENT_PID: '1150',
  PWD: '/home/sammy/first-program',
  LOGNAME: 'sammy',
  GPG_AGENT_INFO: '/run/user/1000/gnupg/S.gpg-agent:0:1',
  GJS_DEBUG_TOPICS: 'JS ERROR;JS LOG',
  WINDOWPATH: '2',
  HOME: '/home/sammy',
  USERNAME: 'sammy',
  IM_CONFIG_PHASE: '2',
  LANG: 'en_US.UTF-8',
  VTE_VERSION: '5601',
  CLUTTER_IM_MODULE: 'xim',
  GJS_DEBUG_OUTPUT: 'stderr',
  LESSCLOSE: '/usr/bin/lesspipe %s %s',
  TERM: 'xterm-256color',
  LESSOPEN: '| /usr/bin/lesspipe %s',
  USER: 'sammy',
  DISPLAY: ':0',
  SHLVL: '1',
  PATH:

'/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin',
  DBUS_SESSION_BUS_ADDRESS: 'unix:path=/run/user/1000/bus',
  _: '/usr/bin/node',
  OLDPWD: '/home/sammy' }
```

Keep in mind that many of the environment variables you see are dependent on the configuration and settings of your system, and your output may look substantially different than what you see here. Rather than viewing a long list of environment variables, you might want to retrieve a specific one.

Step 5 — Accessing a Specified Environment Variable

In this step you'll view environment variables and their values using the global process.env object and print their values to the console.

The process.env object is a simple mapping between environment variable names and their values stored as strings. Like all objects in JavaScript, you access an individual property by referencing its name in square brackets.

Open the environment.js file for editing:

- nano environment.js
-

Copy
Change console.log(process.env); to:

environment.js

```
console.log(process.env["HOME"]);
```
Copy
Save the file and exit. Now run the environment.js program:

- node environment.js
-

Copy
The output now looks like this:

Output

/home/sammy

Instead of printing the entire object, you now only print the HOME property of process.env, which stores the value of the $HOME environment variable.

Again, keep in mind that the output from this code will likely be different than what you see here because it is specific to your system. Now that you can specify the environment variable to retrieve, you can enhance your program by asking the user for the variable they want to see.

Step 6 — Retrieving An Argument in Response to User Input

Next, you'll use the ability to read command line arguments and environment variables to create a command line utility that prints the value of an environment variable to the screen.

Use nano to create a new file echo.js:

- nano echo.js
-

Copy
Add the following code:

echo.js
const args = process.argv.slice(2);
console.log(process.env[args[0]]);

Copy
The first line of echo.js stores all the command line arguments that the user provided into a constant variable called args. The second line prints the environment variable stored in the first element of args; that is, the first command line argument the user provided.

Save and exit nano, then run the program as follows:

- node echo.js HOME
-

Copy
Now, the output would be:

Output
/home/sammy

The argument HOME was saved to the args array, which was then used to find its value in the environment via the process.env object.

At this point you can now access the value of any environment variable on your system. To verify this, try viewing the following variables: PWD, USER, PATH.

Retrieving single variables is good, but letting the user specify how many variables they want would be better.

Step 7 — Viewing Multiple Environment Variables

Currently, the application can only inspect one environment variable at a time. It would be useful if we could accept multiple command line arguments and get their corresponding value in the environment. Use nano to edit echo.js:

- nano echo.js
-

Copy
Edit the file so that it has the following code instead:

echo.js

```
const args = process.argv.slice(2);

args.forEach(arg => {
  console.log(process.env[arg]);
});
```
Copy
The forEach method is a standard JavaScript method on all array objects. It accepts a callback function that is used as it iterates over every element of the array. You use forEach on the args array, providing it a callback function that prints the current argument's value in the environment.

Save and exit the file. Now re-run the program with two arguments:

- node echo.js HOME PWD
-

Copy
You would see the following output:

Output

```
 /home/sammy
/home/sammy/first-program
```
The forEach function ensures that every command line argument in the args array is printed.

Now you have a way to retrieve the variables the user asks for, but we still need to handle the case where the user enters bad data.

Step 8 — Handling Undefined Input

To see what happens if you give the program an argument that is not a valid environment variable, run the following:

- node echo.js HOME PWD NOT_DEFINED
-

Copy
The output will look similar to the following:

Output
 /home/sammy
/home/sammy/first-program
undefined

The first two lines print as expected, and the last line only has undefined. In JavaScript, an undefined value means that a variable or property has not been assigned a value. Because NOT_DEFINED is not a valid environment variable, it is shown as undefined.

It would be more helpful to a user to see an error message if their command line argument was not found in the environment.

Open echo.js for editing:

- nano echo.js
-

Copy
Edit echo.js so that it has the following code:

echo.js

```
const args = process.argv.slice(2);

args.forEach(arg => {
  let envVar = process.env[arg];
  if (envVar === undefined) {
    console.error(`Could not find "${arg}" in environment`);
  } else {
    console.log(envVar);
  }
});
```

Copy
Here, you have modified the callback function provided to forEach to do the following things:

1. Get the command line argument's value in the environment and store it in a variable envVar.
2. Check if the value of envVar is undefined.

3. If the envVar is undefined, then we print a helpful message indicating that it could not be found.
4. If an environment variable was found, we print its value.

Note: The console.error function prints a message to the screen via the stderr stream, whereas console.log prints to the screen via the stdout stream. When you run this program via the command line, you won't notice the difference between the stdout and stderr streams, but it is good practice to print errors via the stderr stream so that they can be easier identified and processed by other programs, which can tell the difference.

Now run the following command once more:

- node echo.js HOME PWD NOT_DEFINED
-
  Copy
  This time the output will be:

  Output
   /home/sammy
  /home/sammy/first-program
  Could not find "NOT_DEFINED" in environment
  Now when you provide a command line argument that's not an environment variable, you get a clear error message stating so.

Conclusion

Your first program displayed "Hello World" to the screen, and now you have written a Node.js command line utility that reads user arguments to display environment variables.

If you want to take this further, you can change the behavior of this program even more. For example, you may want to validate the command line arguments before you print. If an argument is undefined, you can return an error, and the user will only get output if all arguments are valid environment variables.

Introduction to the Digital Computer

A Digital computer can be considered as a digital system that performs various computational tasks.

The first electronic digital computer was developed in the late 1940s and was used primarily for numerical computations.

By convention, the digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit.

A computer system is subdivided into two functional entities: Hardware and Software.

The hardware consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.

The software of the computer consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

Block diagram of a digital computer:

```
                    ┌────────────────────────┐
                    │  Random-access Memory   │
                    │         (RAM)           │
                    └────────────────────────┘
                          ↑         ↓
                    ┌────────────────────────┐
                    │  Central Processing Unit │
                    │         (CPU)           │
                    └────────────────────────┘
                          ↑         ↓
  ┌──────────┐      ┌────────────────────────┐      ┌──────────┐
  │  Input   │ ←─── │  Input - Output Processor│ ───→ │  Output  │
  │ devices  │      │         (IOP)           │      │ devices  │
  └──────────┘      └────────────────────────┘      └──────────┘
```
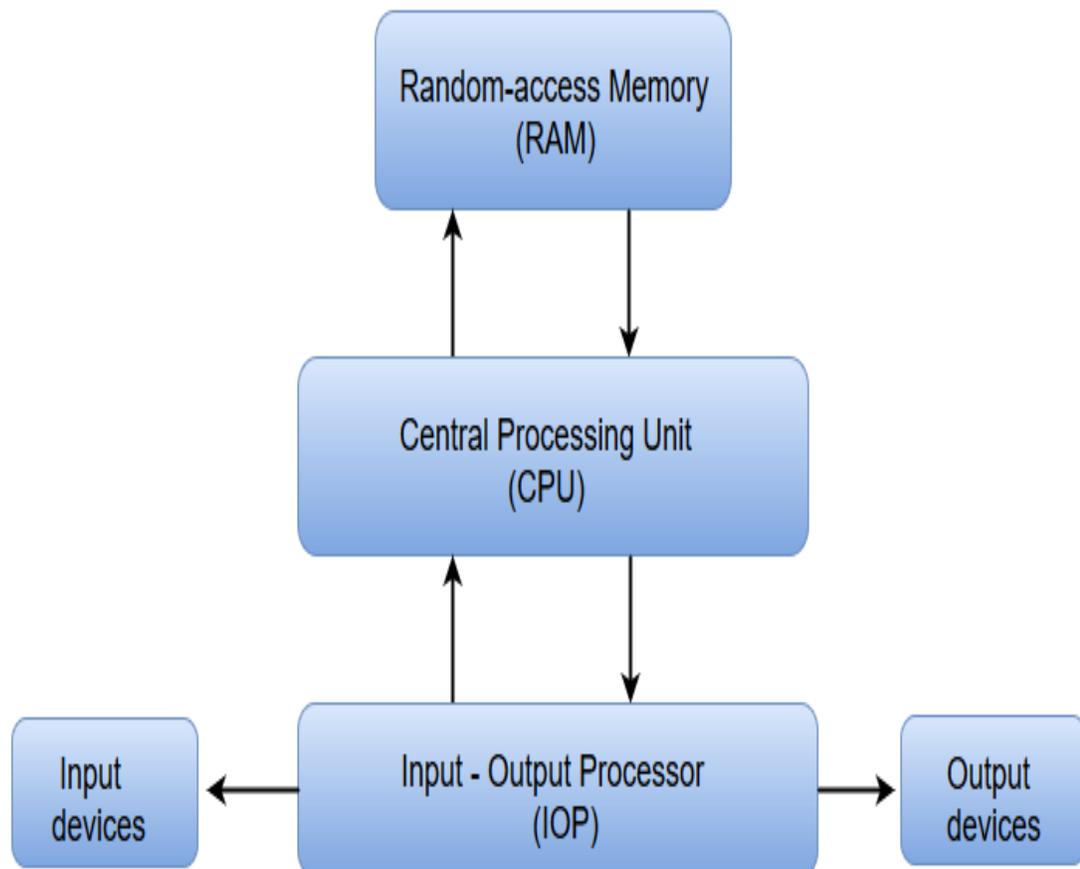
- The Central Processing Unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and a control circuit for fetching and executing instructions.
- The memory unit of a digital computer contains storage for instructions and data.
- The Random Access Memory (RAM) for real-time processing of the data.
- The Input-Output devices for generating inputs from the user and displaying the final results to the user.
- The Input-Output devices connected to the computer include the keyboard, mouse, terminals, magnetic disk drives, and other communication devices.

Concept of an algorithm

Although nowadays algorithms are primarily associated with software and computers, their origins lie much further in the past. They have been used intuitively for centuries, for example in the form of regulatory systems, instructions, rules for games, architectural plans and musical scores.

Even some illustrated art books in the Renaissance period, for example Albrecht Dürer's "Four Books on Measurement" in 1525, were in fact structured instructions for producing paintings, sculptures and buildings. In the history of music, too, from Bach and Mozart to Schönberg and Schillinger, we see mathematical methods and even small mechanical devices being used to make the process of musical composition easier.

Step by step instructions

But what exactly is meant by the term algorithm? Over time, the following descriptions have emerged as definitions: "An algorithm is a procedure for decision-making or an instruction on how to act which consists of a finite number of rules" or "… a limited sequence of unambiguous elementary instructions which exactly and completely describe the way to solve a specific problem." This applies regardless of whether it relates to mathematics, fine art or music.
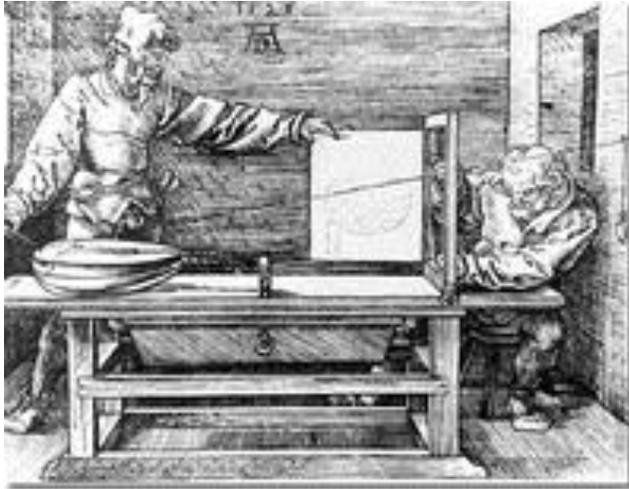
However, the most well-known application of algorithms is undoubtedly their use in computer programming. A program is an algorithm that is formulated in a language that allows it to be processed by a computer. Every computer program – a more advanced machine language – is therefore an algorithm. In this way, people pass the work of processing the procedures required in production or decision-making – often calculations requiring days or hours – to a machine.

Programming becomes ever more precise

The first algorithm designed for a mechanical computing machine – to calculate probabilities using Bernoulli numbers – was written in 1842-1843 by the British mathematician Ada Lovelace in her notes about the work of Charles Babbage's "Analytical Engine", designed in 1833.

However, because the English inventor, mathematician and philosopher never managed to complete his mechanical computing machine for general applications in his lifetime, the algorithm for it was also never implemented.

"The Babbage machine was never built because it was too unwieldy and complicated, even though the software for it was formulated," reports Dr Manuel Bachmann, a researcher at the University of Basel and lecturer at the University of Lucerne, in his book "The triumph of the algorithm – how the idea of software was invented". Nevertheless, other computing machines did see the light of day and, in parallel, more and more precise programming became necessary.

The most complex things can be calculated

In recent decades, algorithms have become a key aspect of information science and the theory of complexity and computability, in particular. Any problem that can be programmed can be resolved by an algorithm. Even the most complex things can be calculated using ones and zeros.

That includes the "Dream of Pythagoras" – the ability to explain the world in the relationships between whole numbers, and the vision of Gottfried Wilhelm Leibniz that all rational truths are based on a kind of calculus. This is all reflected in a digital philosophy and a view of the world based on algorithms which found its most recent formulation in a book called "A New Kind of Science", written in 2002 by the British physicist and mathematician, Stephen Wolfram.

Using numerous visual examples, he describes the power of cellular automata to explain nature, compared with more traditional mathematical models. His views are highly controversial in the scientific community – as has so often been the case in the history of the algorithm.

*DÜRER (21 May 1471 – 6 April 1528) was a painter, printmaker, and theorist of the German Renaissance. Born in Nuremberg, Dürer established his reputation and influence across Europe when he was still in his twenties due to his high-quality woodcut prints.

termination and correctness

In theoretical computer science, correctness of an algorithm is asserted when it is said that the algorithm is correct with respect to a specification. Functional correctness refers to the input-output behavior of the algorithm (i.e., for each input it produces the expected output).[1]

A distinction is made between partial correctness, which requires that if an answer is returned it will be correct, and total correctness, which additionally requires that the algorithm terminates. Since there is no general solution to the halting problem, total correctness is not decidable. A termination proof is a type of mathematical proof that plays a critical role in formal verification because total correctness of an algorithm depends on termination.[2]

For example, successively searching through integers 1, 2, 3, … to see if we can find an example of some phenomenon—say an odd perfect number—it is quite easy to write a partially correct program (using factorization to calculate each integer's aliquot sum). But to say this program is totally correct would be to assert something currently not known in number theory.

A proof would have to be a mathematical proof, assuming both the algorithm and specification are given formally. In particular it is not expected to be a correctness assertion for a given program implementing the algorithm on a given machine. That would involve such considerations as limitations on computer memory.

A deep result in proof theory, the Curry–Howard correspondence, states that a proof of functional correctness in constructive logic corresponds to a certain program in the lambda calculus. Converting a proof in this way is called program extraction.

Hoare logic is a specific formal system for reasoning rigorously about the correctness of computer programs.[3] It uses axiomatic techniques to define programming language semantics and argue about the correctness of programs through assertions known as Hoare triples.

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Although crucial to software quality and widely deployed by programmers and testers, software testing still remains an art, due to limited understanding of the principles of software. The difficulty in software testing stems from the complexity of software: we can not completely test a program with moderate complexity. Testing is more than just debugging. The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Testing can be used as a generic metric as well. Correctness testing and reliability testing are two major areas of testing. Software testing is a trade-off between budget, time and quality.

Algorithms to programs:-

Specification

The concept of an algorithm is fundamental to computer science. Algorithms exist for many common problems. and designing efficient algorithms plays a crucial role in developing large-scale computer systems. Therefore. before we proceed further. we discuss this concept more fully. We begin with a definition.

Definition:

An algorithm is a finite set of instructions that. if followed. accomplishes a particular task. In addition. all algorithms must satisfy the following criteria:

(1) Input. Zero or more quantities are externally supplied.

(2) Output, At least one quantity is produced.

(3) Definiteness. Each instruction is clear and unambiguous

.(4) Finiteness. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) Effectiveness. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

In computational theory, one distinguishes between an algorithm and a program, the latter of which does not have to satisfy the fourth condition. For example, we can think of an operating system that continues in a wait loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs will always terminate, we will use the terms algorithm and program interchangeably in this text.

We can describe an algorithm in many ways. We can use a natural language like English, although if we select this option, 'we must make sure that the resulting instructions are definite. Graphic representations called flowcharts are another possibility, but they work well only if the algorithm is small and simple. In this text, we will present most of our algorithms in C++, occasionally resorting to a combination elf English and C++ for our specifications. Two examples should help to illustrate the process of translating a problem into an algorithm.

Example 1.2 [Selection sort]: Suppose we must devise a program that sorts a collection of > integers. A simple solution is given by the following:
From those integers that are currently unsorted. find the smallest and place it next in the sorted list.

Although this statement adequately describes the sorting problem, it is not an algorithm in because it leaves several unanswered questions. For example, it does not tell us where and how the integers are initially stored or where we should place the result. We assume that tile integers are stored in an array, a, such that the ith integer is stored in a [i -1], 1< i < n. Program 1.5 is our first attempt at deriving a solution. Notice that it is written partially in C++ and partially in English.

To turn Program 1.5 into a real C++ program, two clearly defined subtasks remain: finding the smallest integer and interchanging it with a [i]. We can solve the latter,problem by using the code:

temp = a[i]; a[i] = a [j]; a[j] = temp;

The first subtask can be solved by assuming the minimum is a [i], checking a [i] with [i + 1], a [i + 2], … and whenever a smaller element is found, regarding it as the new.

```
for (int i = 0; i < n ; i++ ) {
    examine a [i] to a [n −1] and suppose the smallest integer is at a [j];
    interchange a [i] and a [j] ;
}
```

Program 1.5: Selection sort algorithm

minimum. Eventually a [n −1] is compared to the current minimum, and we are done. Putting all these observations together, we get the function sort (Program 1.6).

```
1  void sort (int *a, const int n)
2  // sort the n integers a[0] to a[n−1] into nondecreasing order
3  {
4      for ( int i = 0; i < n ; i++ )
5      {
6          int j = i;
7          // find smallest integer in a[i] to a[n − 1]
8          for ( int k = i + 1 ; k < n ; k++ )
9              if (a[k] < a[j]) j = k ;
10         // interchange
11         int temp = a[i]; a[i] = a[j]; a[j] = temp;
12     }
13 }
```

At this point, the obvious question to ask is: Does this function work correctly?

Theorem 1.1: sort(a,n) correctly sorts a set of $n \geq 1$ integers; the result remains in $a[0] \cdots a[n-1]$ such that $a[0] \leq a[1] \leq \cdots \leq a[n-1]$.

Proof: We first note that for any i, say $i = q$, following the execution of lines 6-11, it is the case that $a[q] \leq a[r]$, $q < r \leq n-1$. Also observe that when i becomes greater than q, $a[0] \cdots a[q]$ is unchanged. Hence, following the last execution of these lines (i.e., $i = n-1$), we have $a[0] \leq a[1] \leq \cdots \leq a[n-1]$. □

We observe at this point that the upper limit of the for loop in line 4 can be changed to $n - 1$ without damaging the correctness of the algorithm.

Example 1.3 [Binary search]: Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array $a[0] \cdots a[n-1]$. Our task is to determine if the integer x is present and if so to return j such that $x = a[j]$; otherwise return −1. By making use of the fact that the set is sorted, we conceive of the following efficient method:

Let left and right, respectively, denote the left and right ends of the list to be searched. Initially, left = 0 and right = n−1. Let middle = (left +right)/2 be the middle position in the list. If we compare a [middle] with x, we obtain one of three result·

(1)   x < a[middle]. In this case, if x is present, it must be in the positions between 0 and middle − 1. Therefore, we set right to middle − 1.

(2)   x == a[middle]. In this case, we return middle.

(3)   x > a[middle]. In this case, if x is present, it must be in the positions between middle + 1 and n−1. So, we set left to middle + 1.

If x has not. been found and there are still integers to check, we recalculate middle and continue the search. The algorithm contains two subtasks: (1) determining if there are any integers left to check and (2) comparing x to a[middle].

At this point you might try the method out on some sample numbers. This method is referred to as binary search. Note h~w at each stage the number of elements in the remaining set 'is decreased by about one-half. Note also that at each stage, x is compared with a [middle] and depending on whether x > a [middle], x < a [middle], or x == a [middle]; we do a different thing. To implement this in c++ we could use the if-else construct:

if (x > a [middle]) …
else if (x < a [middle])
else …

From this construct it is not readily apparent that we are considering the three cases that can result from the comparison between x and a [i]. To make the program more transparent, we introduce a compare function that has value >, <, or =, depending on the outcome of the comparison. This function is given in Program 1.7.

```
char compare (int x, int y)
{
    if (x > y) return '>';
    else if (x < y) return '<';
        else return '=';
} // end of compare
```

Program 1.7

We can now refine the description of binary search to get a pseudo-Cs+ function. The result is given in Program 1.8.

```
int BinarySearch (int *a, const int x, const int n)
// Search the sorted array a [0], · · · ,a [n−1] for x
{
    for(initialize left and right; while there are more elements;)
    {
        let middle be the middle element;
        switch (compare (x, a[middle])){
            case '>': set left to middle + 1; break;
            case '<': set right = middle −1; break;
            case '=': found x;
        } // end of switch
    } // end of for
    not found;
} // end of BinarySearch
```

Another refinement yields the C++ function of Program 1.9.

```
1  int BinarySearch (int *a, const int x, const int n)
2  // Search the sorted array a [0], ..., a [n−1] for x
3  {
4       for (int left = 0, right = n − 1; left <= right) { // while more elements
5            int middle = (left + right)/2;
6            switch (compare (x, a[middle])) {
7                 case '>': left = middle + 1; break; // x > a[middle]
8                 case '<': right = middle − 1; break; // x < a[middle]
9                 case '=': return middle; // x == a[middle]
10           } // end of switch
11      } // end of for
12      return −1; // not found
13 } // end of BinarySearch
```

Program 1.9

To prove this program correct we make assertions about the relationship between variables before and after-the Cor loop of lines 4-11. As we enter this loop and if x is present in a, the following holds:

left < right && a[left] < x < a[right] && SORTED (a, n)

Now, if control passes out of the for loop past line II, then we know the condition of line 4 is false, so left> right. This, combined with the above assertion, implies that x is not present.

Unfortunately, a complete proof takes us beyond the scope of this text. but those who wish to pursue program-proving should consult the references at the end of this chapter,

Recursive Algorithms

We have emphasized the need to structure a program to make it easier to achieve the goals of readability and correctness. One of the most useful syntactical features for accomplishing this is the function. A set of instructions that perform a logical operation, perhaps a very complex and long operation, can be grouped together as a function. The function name and its parameters are viewed as a new instruction that can be used in other programs. Given the input-output specifications of a function, we do not even have to know how the task is accomplished, only that it is available. This view of the function implies that it is invoked, executed and returns control to the appropriate place in the calling function. What this fails to stress is that functions may call themselves (direct recursion) before they are done or they may call other functions that again invoke the calling function (indirect recursion). These recursive mechanisms are extremely powerful, but even more importantly, often they can express an otherwise complex process very clearly. For these reasons we introduce recursion here.

Recursion is similar to the method of induction which is often used to prove mathematical statements. In mathematical induction, a statement about integers (e.g., the sum of the first n positive integers is n (n +1)12) is proved by showing that the statement can be proved for integer k if it is assumed to be true for integer k -1.

Similarly, in recursion, we write a function to produce an output (say n!) for some input (here, n) by assuming that the same function will compute the correct output for input n – I. In mathematical induction, we need a basis which can be directly proved (that is, the proof for the basis does not make any assumptions). Similarly, a recursive function requires a terminating condition. When the input to the function satisfies this terminating condition, the function directly computes the output without calling itself.

What kinds of problems are best solved by recursion? Typically, beginning programmers view recursion as a somewhat mystical technique that is useful only for some very special class of problems (such as computing factorials or Ackermann's function). This is unfortunate because any program that can be written using assignment, the if-else statement, and the while statement can also be written using assignment, if-else, and recursion. Of course, this does not mean that the resulting program will necessarily be easier to understand. However, there are many instances when this will be the case. When is recursion an appropriate mechanism for algorithm exposition? One instance is when the problem itself is recursively defined. Factorial fits this category, as well as binomial coefficients where

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

can be recursively computed by the formula

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

We use two examples to show you how to develop a recursive algorithm. In the first example, we take the binary search function that we created in Example 1.3 and transform it into a recursive function. In the second example, we generate all possible permutations of a list of characters. To understand a recursive function, you must.

(1) Formulate in your mind a statement of what it is that the function is supposed to do, for a given input.

(2) Verify that the function does achieve its goal if the recursive invocations to itself do what they are supposed to.

(3) Ensure that a finite number of recursive invocations of the function eventually lead to an invocation which satisfies the terminating condition (otherwise, the function will keep calling itself and not terminate!).·

(4) The function should perform the correct computations if the terminating condition is encountered. .

Example 1.4 [Recursive binary search]: Program 1.9 gave the iterative version of binary search. In the recursive version· We pass left and right as parameters (Program 1.10).

The for loop of Program. 1.9 has been replaced by recursive calls in Program 1.10. To invoke the recursive function, we use the statement

BinarySearch (a, x, 0, n -1);

You should verify that Binary'Search satisfies the four conditions stated above for recursive function's. Notice that both the iterative (Program 1.9) and recursive (Program 1.10) functions perform the same computation.

Example 1.5 [Permutation generator]:· Given a set of n > 1 elements, the problem is to print all possible permutations of this set. For example if the set is {a, b, c}, then the set of permutations is {(a, b, c), (a, c, b),(b, a, c),(b, c, a),(c, a, b),(c, b, a)}. It is easy to see that given n elements, there are n ! different permutations. A simple algorithm can be obtained by looking at the the case of four elements (a,b,c,d). The answer can be constructed by writing.

```
int BinarySearch (int *a, const int x, const int left, const int right)
// Search the sorted array a[left], ···, a[right] for x
{
    if (left <= right) {
        int middle = (left + right)/2;
        switch (compare (x, a[middle])) {
            case '>': return BinarySearch(a, x, middle + 1, right); // x > a[middle]
            case '<': return BinarySearch(a, x, left, middle - 1); // x < a[middle]
            case '=': return middle; // x == a[middle]
        } // end of switch
    } // end of if
    return -1; // not found
} // end of BinarySearch
```

Program 1.10

(I) a followed by all permutations of (b,c,d)

(2) b followed by all permutations of (a,c,d)

(3) c followed by all permutations of (a,b,d)

(4) d followed by all permutations of (a,b,c)

The expression "followed by all permutations" is the clue to 'recursion. It implies that we can solve the problem for a set with n elements if we have an algorithm that works on n – 1 elements. These observations lead to Program 1.11, which is invoked by perm (a, 0, n).

Try this algorithm out on sets of length one, two, and three to ensure that you understand how it works.

Another time when recursion is useful is when the data structure that the algorithm is to operate on is recursively defined. We shall see several important examples of such structures in this book.

1. Horner's rule is a means for evaluating a polynomial $A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ at a point x_0 using a minimum number of multiplications. This rule is:

$$A(x) = (\cdots (a_n x_0 + a_{n-1})x_0 + \cdots + a_1) x_0 + a_0$$

Write a C++ program to evaluate a polynomial using Horner's rule. Determine how many times each statement is executed.

```
void perm (char *a, const int k, const int n) // n is the size of a
// Generate all the permutations of a[k], ... a[n-1].
{
    if (k == n - 1) { // output permutation
        for (int i = 0; i < n; i++) cout << a[i] << " ";
        cout << endl;
    }
    else { // a[k], ... ,a[n-1] has more than one permutation. Generate these recursively.
        for (i = k; i < n; i++) {
            // interchange a[k] and a[i]
            char temp = a[k]; a[k] = a[i]; a[i] = temp;
            perm(a, k+1, n); // all permutations of a[k+1], ... ,a[n-1]
            // interchange a[k] and a[i] again, to return to the original configuration
            temp = a[k]; a[k] = a[i]; a[i] = temp;
        }
    } // end of else
} // end of perm
```

Program 1.11

2. Given n Boolean variables x!….., Xn we wish to print all possible combinations '. of truth values they can assume. For instance, if n=2, there are four possibilities: true, true; true, false; false, true; false, false. Write a C++ program to accomplish this and do a frequency count.

3. Write a C++ program that prints out the integer values of x. y. and z in nondecreasing order. What is the computing time of your method?

4.. Write a C++ function that searches an array a [n] for the element x. If x occurs, then set j to its position in the array. else set j to -I. Try writing this without using the goto statement.

5. Trace the action of the code

$$i = 0; \; j = n - 1;$$
$$\text{do } \{$$
$$\quad k = (i + j)/2;$$
$$\quad \text{if } (a[k] <= x) \; i = k + 1;$$
$$\quad \text{else } j = k - 1; \}$$
$$\text{while } (i <= j);$$

on the elements 2..4,6,8. 10. 12. 14. 16. 18, and 20 searching for x= 1.3, 13. or 21.

6. Take any version of binary search, express it using assignment, if-else, and goto, and then give an equivalent recursive program. .

7. The factorial function n! has value I when n ~ 1 and value n*(n-I)! when n > I. Write both a recursive and an iterative C++ function to compute n I.

8. Write an iterative function to compute a binomial coefficient; then transform it into an equivalent recursive function.

9. Ackermann's function A (m,n) is defined as follows:
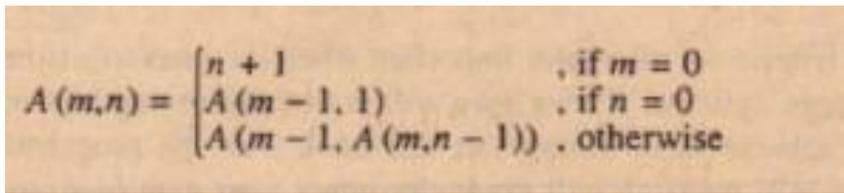
$$A(m,n) = \begin{cases} n+1 & \text{, if } m = 0 \\ A(m-1, 1) & \text{. if } n = 0 \\ A(m-1, A(m,n-1)) & \text{. otherwise} \end{cases}$$

10. The pigeonhole principle states that if a function f has n distinct inputs but less than n distinct outputs, then there exist two inputs a and b such that a :1: band f(a) = f(b). Write a program to find the values a and b for which the range values are equal. Assume that the inputs are I, 2, … , n.

11. Given n, a positive integer, determine if n is the sum of all of its divisors i.e., if is the sum of all t such that $1 < t < n$, and t divides n.

12. Consider the function F (x) defined by
if (x is even) F = x /2;
else F = F(F(3x + 1))
Prove that F(x) terminates for all integers x. (Hint: Consider integers of the form $(2i + I)2k – I$ and use induction.)

13. If S is a set Of n elements, the powerset of S is the set of all possible subsets of S. For example, if S = (a,b,c), then powerset (S) = ((), (a), (b), (c), (a,b), (a,c), b,c), (a,b,c)}. Write a recursive function to compute powerset (S).

14. [Towers of Hanoi) There are three towers and sixty-four disks of different diameters placed on the first tower. The disks are in order of decreasing diameter as one scans up the tower. Monks were supposed to move the disks from tower I to tower 3 obeying the following rules: (a) only one disk can be moved at any time and (b) no disk can be placed on top of a disk with smaller diameter. Write a recursive function that prints the sequence of moves that accomplish this task.

top-down development and stepwise refinement

Near the beginning of learning Java, you will hear something about top-down design, stepwise refinement and decomposition. This post today will explain what those terms

mean as well as provide the reason why you need to tackle programming design with this approach in mind.

Just for the sake of clarity, top-down design is the process of stepwise design and decomposition.

Before you start writing any code for a new program you want to create, you first need to design the program or at least have a specification so that you know exactly what you need to create.

The top-down design approach to programming

Now that you know what you need to accomplish with your program, you now need to know how you can accomplish it. The approach used to do that is called top-down design and this simply means putting all the important main parts of the program in to a list.
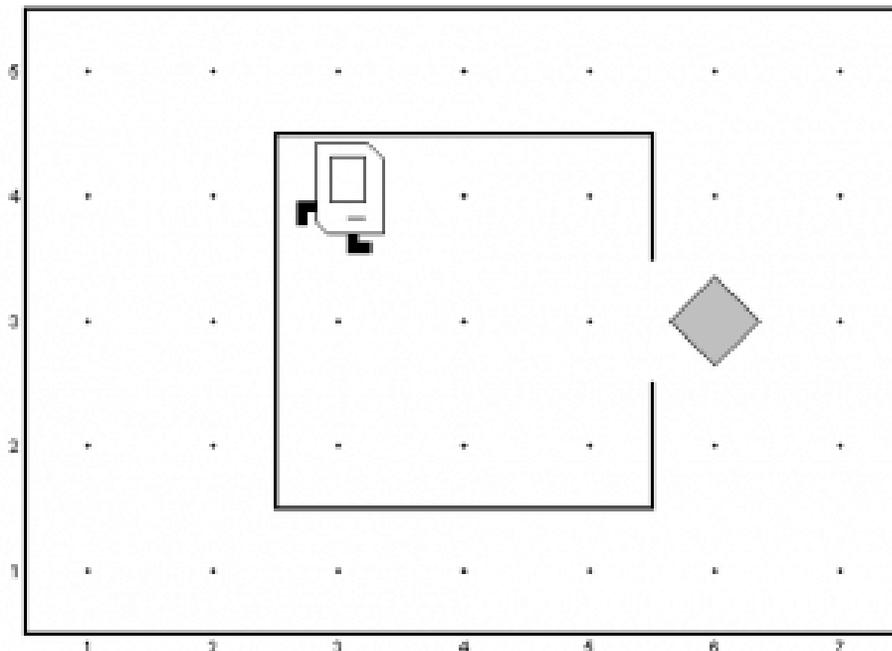
An example I want to use is from a Karel world. In this example, Karel needs to collect the beeper (which is called a newspaper in the assignment). The assignment states that Karel's world doesn't change and that the door is always where it is and the paper is always on point 6,3 and that when collected, Karel needs to return back to the start.

A top-down design approach could go something like this:

1. Walk to paper.

2. Pick it up.

3. Return back to start.

This is our top level that describes what needs to be done in the most basic form. I guess we could start at an even higher level and simply say "Collect paper", but programming problems can be tackled in many ways and the way above is simply the way I have decided to do it.

Stepwise Refinement/Decomposition

The next process is what is called stepwise refinement or decomposition. Rather than leaving step 1 as "walk to paper", we need to be more clear than this. So, to break this step down with stepwise refinement, we could say…

1. Walk to paper.

1a. Walk to wall

1b. turn right

1c. move

1d. turn left

As this program is relatively simple, I think it can be left at just the one breakdown. However, more complicated programs will require that steps 1a, 1b, 1c, 1d and 1e are decomposed several times.

For step 2 "pick it up", the assignment says we can assume there will always be a paper there. To be on the safe side, we might want to have a bit more detail here and put checks in place such as using an if statement to check if a paper is present. Also, we could look here at pre-conditions and post-conditions. We need to know where Karel will be after step 1 and where he needs to be to start step 3. In the breakdown of "walk to paper" I decided to leave Karel at the inside of the door facing out… so a pre-condition is that Karel needs to make 1 move to the paper. A post-condition I will set is for Karel is that he has returned to inside the house facing west which means we need to remember that as a pre-condition for step 3… returning to the start.

Breaking down part 2:

2. Pick it up

2a. Move to paper

2b. Check if paper available

2c. pick up paper

2d. turn around

2e. move (leave facing west)

Now we know that Karel is in the house, facing west we can break down step 3 as follows:

3. Return back to the start

3a. move to wall

3b. turn right

3c. move to wall

I decided to tackle point 3 a different way. Instead of sending Karel back to the couch the same way he came in… along the top and right wall. I decided to make him simply continue west till he hits the wall, then turn right and then move to the top wall.

Conclusion

When you are writing programs, remember to use an approach that helps. Top-down design is one way of doing this. It breaks the program in to bite-sized chunks and allows you to solve each problem one at a time, in any order. Remember that you need to make a note of pre and post conditions so that when you write the program out of sequence, or as part of a team, all people know that Karel is facing west at a certain point and can safely assume they just need to move him to a wall.

Also notice that 1a, 3a and 3c could technically use the same method… perhaps called "walkToWall()". As long as pre and post conditions are taken in to account, Karel is doing the same thing for each of those steps which means you have solved the problem in a more simple way.

The first assignment in Karel is relatively simple and top-down design with stepwise refinement and decomposition still finds a use here. In larger projects that you may end up working on as part of a team, a systematic approach is needed so that you can get

through a project more easily. Make sure your notes are up to speed and that you keep practicing and not forgetting comments or top-down design when creating your first and all other programs.

Introduction to Programming

Are you aiming to become a software engineer one day? Do you also want to develop a mobile application that people all over the world would love to use? Are you passionate enough to take the big step to enter the world of programming? Then you are in the right place because through this article you will get a brief introduction to programming. Now before we understand what programming is, you must know what is a computer. A computer is a device that can accept human instruction, processes it and responds to it or a computer is a computational device which is used to process the data under the control of a computer program. Program is a sequence of instruction along with data.
The basic components of a computer are:

1. Input unit

2. Central Processing Unit(CPU)

3. Output unit

The CPU is further divided into three parts-

☐ Memory unit

☐ Control unit

☐ Arithmetic Logic unit

Most of us have heard that CPU is called the brain of our computer because it accepts data, provides temporary memory space to it until it is stored(saved) on the hard disk, performs logical operations on it and hence processes(here also means converts) data into information. We all know that a computer consists of hardware and software. Software is a set of programs that performs multiple tasks together. An operating system is also a software (system software) that helps humans to interact with the computer system.

A program is a set of instructions given to a computer to perform a specific operation. or computer is a computational device which is used to process the data under the control of a computer program. While executing the program, raw data is processed into a desired output format. These computer programs are written in a programming language which are high level languages. High level languages are nearly human languages which are more complex then the computer understandable language which are called machine language, or low level language. So after knowing the basics, we are ready to create a very simple and basic program. Like we have different languages to communicate with

each other, likewise, we have different languages like C, C++, C#, Java, python, etc to communicate with the computers. The computer only understands binary language (the language of 0's and 1's) also called machine-understandable language or low-level language but the programs we are going to write are in a high-level language which is almost similar to human language.

The piece of code given below performs a basic task of printing "hello world! I am learning programming" on the console screen. We must know that keyboard, scanner, mouse, microphone, etc are various examples of input devices and monitor(console screen), printer, speaker, etc are the examples of output devices.

```
main()
{
clrscr();
printf("hello world! I am learning to program);
getch();
}
```

At this stage, you might not be able to understand in-depth how this code prints something on the screen. The main() is a standard function that you will always include in any program that you are going to create from now onwards. Note that the execution of the program starts from the main() function. The clrscr() function is used to see only the current output on the screen while the printf () function helps us to print the desired output on the screen. Also, getch() is a function that accepts any character input from the keyboard. In simple words, we need to press any key to continue(some people may say that getch() helps in holding the screen to see the output).

Between high-level language and machine language there are assembly language also called symbolic machine code. Assembly language are particularly computer architecture specific. Utility program (Assembler) is used to convert assembly code into executable machine code. High Level Programming Language are portable but require Interpretation or compiling toconvert it into a machine language which is computer understood.

Hierarchy of Computer language –

High level language

↓

Assembley language

↓

Machine Language

↓

Computer Hardware
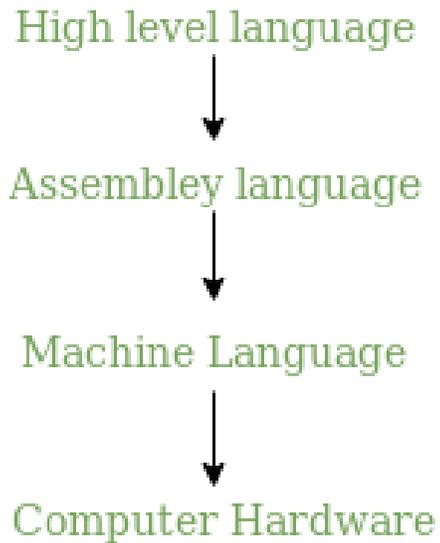
There have been many programming language some of them are listed below:

C	Python	C++
C#	R	Ruby
COBOL	ADA	Java
Fortran	BASIC	Altair BASIC
True BASIC	Visual BASIC	GW BASIC
QBASIC	PureBASIC	PASCAL
Turbo Pascal	GO	ALGOL
LISP	SCALA	Swift
Rust	Prolog	Reia

Racket	Scheme	Shimula
Perl	PHP	Java Script
CoffeeScript	VisualFoxPro	Babel
Logo	Lua	Smalltalk
Matlab	F	F#
Dart	Datalog	dbase
Haskell	dylan	Julia
ksh	metro	Mumps
Nim	OCaml	pick
TCL	D	CPL
Curry	ActionScript	Erlang
Clojure	DarkBASCIC	Assembly

Most Popular Programming Languages –

- C
- Python
- C++
- Java
- SCALA
- C#
- R
- Ruby
- Go
- Swift
- JavaScript

Characteristics of a programming Language –

- A programming language must be simple, easy to learn and use, have good readability and human recognizable.

- Abstraction is a must-have Characteristics for a programming language in which ability to define the complex structure and then its degree of usability comes.

- A portable programming language is always preferred.

- Programming language's efficiency must be high so that it can be easily converted into a machine code and executed consumes little space in memory.

- A programming language should be well structured and documented so that it is suitable for application development.

- Necessary tools for development, debugging, testing, maintenance of a program must be provided by a programming language.

- A programming language should provide single environment known as Integrated Development Environment(IDE).

- A programming language must be consistent in terms of syntax and semantics.

Use of high level programming language for the systematic development of programs

How do you think we communicate with a computer? A computer cannot understand any commands that you may give in English or in any other language. It has its own set of instructions for communication, or what we call computer languages. This is an important part of your syllabus for banking exams. Let us take a look.

Computer Languages

The user of a computer must be able to communicate with it. That means, he must be able to give the computer commands and understand the output that the computer generates. This is possible due to the invention of computer languages.

Basically, there are two main categories of computer languages, namely Low Level Language and High Level Language. Let us take a brief look at both these types of computer languages.

1] Low Level Languages

Low level languages are the basic computer instructions or better known as machine codes. A computer cannot understand any instruction given to it by the user in English or any other high level language. These low level languages are very easily understandable by the machine.

The main function of low level languages is to interact with the hardware of the computer. They help in operating, syncing and managing all the hardware and system components of the computer. They handle all the instructions which form the architecture of the hardware systems.

Browse more Topics under Basics Of Computers

- Number Systems

- Number System Conversions

- Generations of Computers

- Computer Organisation

- Computer Memory

- History of Computers

- Computers Abbreviations

- Basic Computer Terminology

- Basic Internet Knowledge and Protocols

- Hardware and Software

- Keyboard Shortcuts

- I/O Devices

- Practice Problems On Basics Of Computers

Machine Language

This is one of the most basic low level languages. The language was first developed to interact with the first generation computers. It is written in binary code or machine code, which means it basically comprises of only two digits – 1 and 0.

Assembly Language

This is the second generation programming language. It is a development on the machine language, where instead of using only numbers, we use English words, names, and symbols. It is the most basic computer language necessary for any processor.

2] High Level Language

When we talk about high level languages, these are programming languages. Some prominent examples are PASCAL, FORTRAN, C++ etc.

The important feature about such high level languages is that they allow the programmer to write programs for all types of computers and systems. Every instruction in high level language is converted to machine language for the computer to comprehend.

Scripting Languages

Scripting languages or scripts are essentially programming languages. These languages employ a high level construct which allows it to interpret and execute one command at a time.

Scripting languages are easier to learn and execute than compiled languages. Some examples are AppleScript, JavaScript, Pearl etc.

Object-Oriented Languages

These are high level languages that focus on the 'objects' rather than the 'actions'. To accomplish this, the focus will be on data than logic.

The reasoning behind is that the programmers really cares about the object they wish to manipulate rather than the logic needed to manipulate them. Some examples include Java, C+, C++, Python, Swift etc.

Procedural Programming Language

This is a type of programming language that has well structured steps and complex procedures within its programming to compose a complete program.

It has a systematic order functions and commands to complete a task or a program. FORTRAN, ALGOL, BASIC, COBOL are some examples.

Introduction to the design and implementation of correct

Software Design is the process to transform the user requirements into some suitable form, which helps the programmer in software coding and implementation. During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. Hence the aim of this phase is to transform the SRS document into the design document.
The following items are designed and documented during the design phase:

- Different modules required.
- Control relationships among modules.
- Interface among different modules.
- Data structure among the different modules.
- Algorithms required to implement among the individual modules.

Objectives of Software Design:

1. **Correctness:**

   A good design should be correct i.e. it should correctly implement all the functionalities of the system.

2. **Efficiency:**

   A good software design should address the resources, time and cost optimization issues.

3. **Understandability:**

   A good design should be easily understandable, for which it should be modular and all the modules are arranged in layers.

4. **Completeness:**

   The design should have all the components like data structures, modules, and external interfaces, etc.

5. **Maintainability:**

   A good software design should be easily amenable to change whenever a change request is made from the customer side.

Software Design Concepts:

Concepts are defined as a principal idea or invention that comes in our mind or in thought to understand something. The software design concept simply means the idea or

principle behind the design. It describes how you plan to solve the problem of designing software, the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system or software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:

Abstraction

Refactoring

Modularity

Software Design Concepts

Information hiding

Architecture

Refinement

Pattern
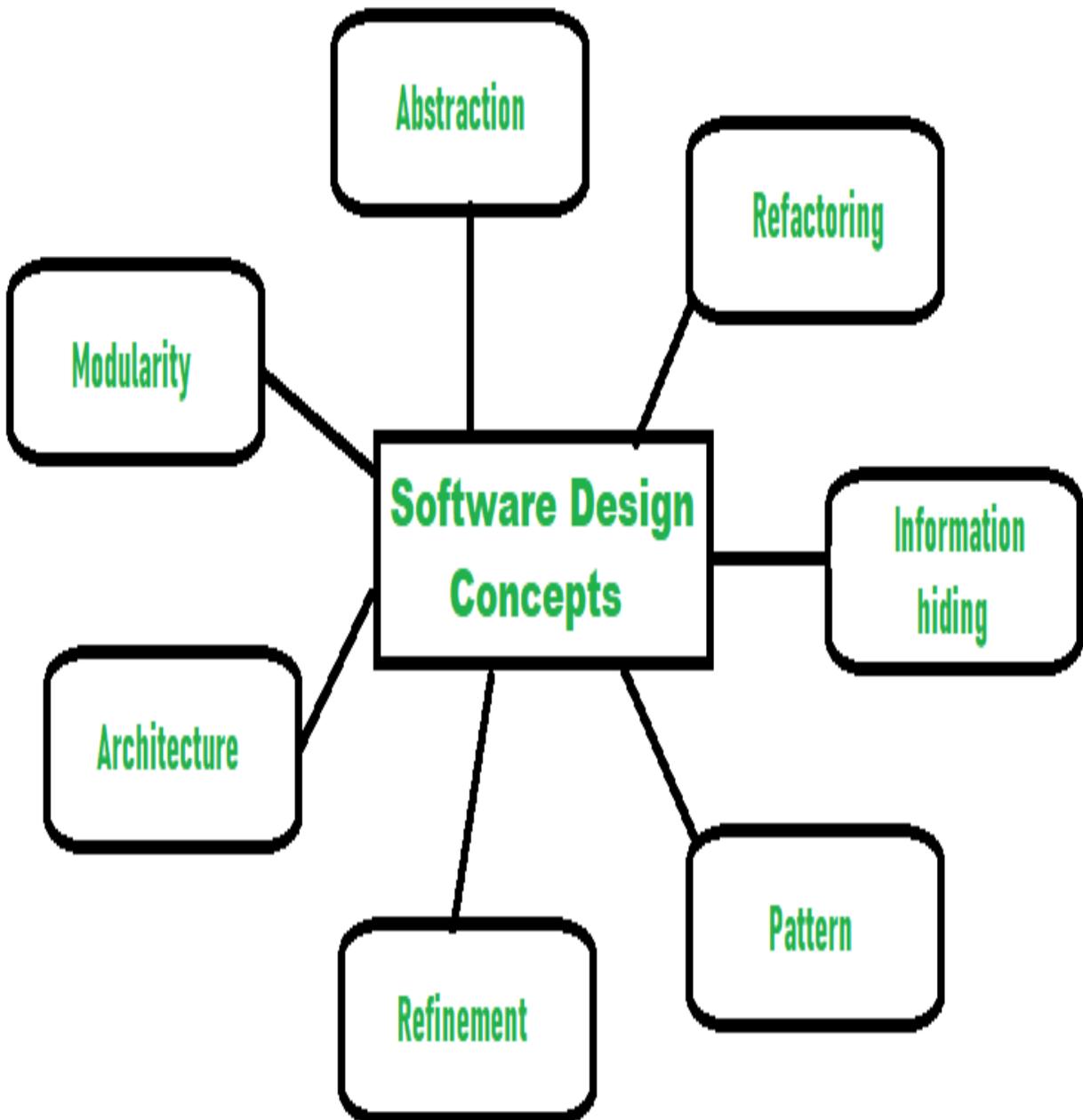
Following points should be considered while designing a Software:

1. **Abstraction- hide relevant data**

   Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broadways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.

2. **Modularity- subdivide the system**

   Modularity simply means to divide the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means to subdivide a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity is design has now become a trend and is also important.

3. **Architecture- design a structure of something**

   Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.

4. **Refinement- removes impurities**

   Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software. Refinement is very necessary to find out any error if present and then to reduce it.

5. **Pattern- a repeated form**

   The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.

6. **Information Hiding- hide the information**

   Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and it cant be accessed by any other modules.

7. **Refactoring- reconstruct something**

   Refactoring simply means to reconstruct something in such a way that it does not affect the behavior or any other features. Refactoring in software design means to reconstruct the design to reduce and complexity and simplify it without affecting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't affect the behavior of the design and improves the internal structure".

Different levels of Software Design:

There are three different levels of software design. They are:

1. **Architectural Design:**

   The architecture of a system can be viewed as the overall structure of the system & the way in which structure provides conceptual integrity of the system. The architectural design identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of the proposed solution domain.

2. **Preliminary or high-level design:**

   Here the problem is decomposed into a set of modules, the control relationship among various modules identified and also the interfaces among various modules are identified. The outcome of this stage is called the program architecture. Design representation techniques used in this stage are structure chart and UML.

3. **Detailed design:**

   Once the high level design is complete, detailed design is undertaken. In detailed design, each module is examined carefully to design the data structure and algorithms. The stage outcome is documented in the form of a module specification document.

efficient and maintainable programs

in this phase , the user's expectations are gathered to understand why the program or software has to be developed. Then , all the gathered requirements are analysed and the scope or objective of the over all software product is penned down. The last activity in this phase involves documenting every identified requirement of the user in order to avoid any doubts or uncertainty regarding the functionality of the program. The functionality, capability, performance, and availability of hardware and software components are all analysed in this phase.

Design:

The requirement documented in the previous phase act as the input to the design phase. In this phase, a plan of actions is made before the actual development process starts. This plan will be followed throughout the development process. Moreover, in the design phase, the core structure of the software or program is broken down in to modules. The solution of the program is then specified for each module in the form of algorithms or flow charts . The design phase, therefore specifies how the program or software will be developed.

Implementation:

In this phase, the designed algorithms are converted in to program code using any of the high level languages. The particular choice of language will depend on the type of program such as whether it is a system or an application program. C is preferred for writing system programs, whereas Visual basic might be preferred for an application program. The program codes are tested by the programmer to ensure their correctness.

Testing:
In this phase , all the modules are tested together to ensure that the overall system works well as a whole product. In this phase, the software is tested using a large number of varied inputs, also known as test data, to ensure that the software is working as expected by the user's requirements identified in the requirements analysis phase.

Software deployment, training, and support:

After the code is tested and the software or the program is approved by the user's it is then installed or deployed in the production environment.

Software training and support is a crucial phase. Program designers and developers spend a lot of time creating the software, but if nobody in the organization knows how to use it or to fix certain problems , then no one will want to use it.

Maintenance:

Maintenance and enhancements are ongoing activities that are done to cope with newly discovered problems or new requirements. Such activities may take a long time to complete.

Structured Programming

Since the invention by Von Neumann of the stored program computer, computer scientists have known that a tremendous power of computing equipment was the ability to alter its behavior, depending on the input data. Calculating machines had, for some time, been able to perform fixed arithmetic operations on data, but the potential of machines capable of making decisions opened up many new possibilities. Machines that could make decisions were capable of sorting records, tabulating and summarizing data, searching for information, and many more advanced operations that could not even be imagined at the time.

In early programming languages, like Fortran (first invented in 1954) and various low level machine languages, the goto statement allowed the computer to deviate from the sequential execution of the program instructions. The goto statement was recognized to be a very powerful construction, and soon, programs of increasing complexity and power were developed.

However, the increasingly complex code that resulted from goto statements became harder and harder to maintain. Dijkstra, in 1966, was one of the first persons to recognize that this run away complexity of programs was due to the overuse of the goto statement (Dijkstra, E. W., "Go To Considered Harmful," Communications of the ACM, March 1966). In fact, it was determined shortly thereafter, that the goto statement is not needed at all. Dijkstra showed that any program construction that could be created with goto statements could be created more simply with the sequence, repetition and decision constructions that are discussed in the following sections. This was the birth of the discipline of Structured Programming.

Structured Programming in Everyday Life

1. Sequence   Execute a list of statements in order.
> **Example: Baking Bread**

> **Add flour.**
> **Add salt.**
> **Add yeast.**
> **Mix.**
> **Add water.**
> **Knead.**
> **Let rise.**
> **Bake.**

2. Repetition   Repeat a block of statements while a condition is true.

> **Example: Washing Dishes**

> **Stack dishes by sink.**
> **Fill sink with hot soapy water.**
> **While moreDishes**
>   **Get dish from counter,**
>   **Wash dish,**
>   **Put dish in drain rack.**
> **End While**
> **Wipe off counter.**
> **Rinse out sink.**

3. Selection   Choose at most one action from several alternative conditions.

> **Example: Sorting Mail**

> **Get mail from mailbox.**
> **Put mail on table.**
> **While moreMailToSort**
>   **Get piece of mail from table.**
>   **If pieceIsPersonal Then**
>   **Read it.**
>   **ElseIf pieceIsMagazine Then**
>   **Put in magazine rack.**
>   **ElseIf pieceIsBill Then**
>   **Pay it,**

```
    ElseIf pieceIsJunkMail Then
        Throw in wastebasket.
    End If
End While
```

Structured Programming in Visual Basic

Structured programming is a program written with only the structured programming constructions: (1) sequence, (2) repetition, and (3) selection.

1. **Sequence.** Lines or blocks of code are written and executed in sequential order.

    Example:

    ```
    x = 5
    y = 11
    z = x + y
    WriteLine(z)
    ```

2. **Repetition.** Repeat a block of code (Action) while a condition is true. There is no limit to the number of times that the block can be executed.

    ```
    While condition
        action
    End While
    ```

    Example:

    ```
    x = 2
    While x < 100
        WriteLine(x)
        x = x * x
    End
    ```

3. **Selection.** Execute a block of code (Action) if a condition is true. The block of code is executed at most once.

If condition Then
   action
End If

Example:

x = ReadLine()
If x Mod 2 = 0
   WriteLine("The number is even.")
End If

Extensions to Structured Programming

To make programs easier to read, some additional constructs were added to the basic three original structured programming constructs:

1. **Definite Repetition (For Loop)** Combine initialization, checking a condition, and incrementing a counter in a single statement called a for statement. Here is the general form:

   For i = 1 To n Step k   ' Step k is optional
      action
   Next

   Example:

   For i = 1 To 20
      WriteLine(i)
   Next i

   Example:

   For i = 20 To 1 Step -1
      WriteLine(i)
   Next

2. **If-Then-Else Statements** Execute the first action whose corresponding condition is true. Here is the general form:

```
If condition1 Then
   action1
ElseIf condition2 Then
   action2
ElseIf condition3 Then
   action3
Else
   defaultAction
End If
```

Example:

```
If n = 1 Then
   WriteLine("One")
ElseIf n = 2 Then
   WriteLine("Two")
ElseIf n = 3 Then
   WriteLine("Three")
Else
   WriteLine("Many")
End If
```

3. **Select Statement**   Execute the action corresponding to the value of the expression.

```
Select Case value1
   Case value1
      action1
   Case value2
      action2
   Case value3
      action3
   Case Else
      defaultAction
End Select
```

Example:

```
Select Case n
   Case 1
      WriteLine("One")
   Case 2
      WriteLine("Two")
   Case 3
```

```
      WriteLine("Three")
   Case Else
      WriteLine("Many")
End Select
```

Trace an algorithm to depict the logic

A trace table is a technique used to test algorithms in order to make sure that no logical errors occur while the calculations are being processed. The table usually takes the form of a multi-column, multi-row table; With each column showing a variable, and each row showing each number input into the algorithm and the subsequent values of the variables.

Trace tables are typically used in schools and colleges when teaching students how to program. They can be an essential tool in teaching students how certain calculations works and the systematic process that is occurring when the algorithm is executed. They can also be useful for debugging applications, helping the programmer to easily detect what error is occurring, and why it may be occurring.

Example [edit]

```
int i, x = 0;
for (i = 1; i <= 10; i++)
{
    x = i * 2;
}
```

i	x
?	0
1	2
2	4
3	6

4	8
5	10
6	12
7	14
8	16
9	18
10	20
11	20

This example shows the systematic process that takes place whilst the algorithm is processed. The initial value of x is zero, but i, although defined, has not been assigned a value. Thus, its initial value is unknown. As we execute the program, line by line, the values of i and x change, reflecting each statement of the source code in execution. Their new values are recorded in the trace table. When i reaches the value of 11 because of the i++ statement in the for definition, the comparison i <= 10 evaluates to false, thus halting the loop. As we also reached the end of the program, the trace table also ends.

Number Systems and conversion methods

There are many methods or techniques which can be used to convert numbers from one base to another. We'll demonstrate here the following −

- Decimal to Other Base System
- Other Base System to Decimal
- Other Base System to Non-Decimal
- Shortcut method − Binary to Octal

- Shortcut method − Octal to Binary
- Shortcut method − Binary to Hexadecimal
- Shortcut method − Hexadecimal to Binary

Decimal to Other Base System

Steps

- **Step 1** − Divide the decimal number to be converted by the value of the new base.

- **Step 2** − Get the remainder from Step 1 as the rightmost digit (least significant digit) of new base number.

- **Step 3** − Divide the quotient of the previous divide by the new base.

- **Step 4** − Record the remainder from Step 3 as the next digit (to the left) of the new base number.

Repeat Steps 3 and 4, getting remainders from right to left, until the quotient becomes zero in Step 3.

The last remainder thus obtained will be the Most Significant Digit (MSD) of the new base number.

Example −

Decimal Number: 29_{10}

Calculating Binary Equivalent –

Step	Operation	Result	Remainder
Step 1	29 / 2	14	1
Step 2	14 / 2	7	0
Step 3	7 / 2	3	1
Step 4	3 / 2	1	1

Step 5	1 / 2	0	1

As mentioned in Steps 2 and 4, the remainders have to be arranged in the reverse order so that the first remainder becomes the Least Significant Digit (LSD) and the last remainder becomes the Most Significant Digit (MSD).

Decimal Number − 29_{10} = Binary Number − 11101_2.

Other Base System to Decimal System

Steps

- **Step 1** − Determine the column (positional) value of each digit (this depends on the position of the digit and the base of the number system).

- **Step 2** − Multiply the obtained column values (in Step 1) by the digits in the corresponding columns.

- **Step 3** − Sum the products calculated in Step 2. The total is the equivalent value in decimal.

Example

Binary Number − 11101_2

Calculating Decimal Equivalent –

Step	Binary Number	Decimal Number
Step 1	11101_2	$((1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	11101_2	$(16 + 8 + 4 + 0 + 1)_{10}$
Step 3	11101_2	29_{10}

Binary Number − 11101_2 = Decimal Number − 29_{10}

Other Base System to Non-Decimal System

Steps

- **Step 1** − Convert the original number to a decimal number (base 10).
- **Step 2** − Convert the decimal number so obtained to the new base number.

Example

Octal Number − 25_8

Calculating Binary Equivalent −

Step 1 − Convert to Decimal

Step	Octal Number	Decimal Number
Step 1	25_8	$((2 \times 8^1) + (5 \times 8^0))_{10}$
Step 2	25_8	$(16 + 5)_{10}$
Step 3	25_8	21_{10}

Octal Number − 25_8 = Decimal Number − 21_{10}

Step 2 − Convert Decimal to Binary

Step	Operation	Result	Remainder
Step 1	21 / 2	10	1
Step 2	10 / 2	5	0
Step 3	5 / 2	2	1
Step 4	2 / 2	1	0

| Step 5 | 1 / 2 | 0 | 1 |

Decimal Number − 21_{10} = Binary Number − 10101_2

Octal Number − 25_8 = Binary Number − 10101_2

Shortcut method - Binary to Octal

Steps

- **Step 1** − Divide the binary digits into groups of three (starting from the right).
- **Step 2** − Convert each group of three binary digits to one octal digit.

Example

Binary Number − 10101_2

Calculating Octal Equivalent −

Step	Binary Number	Octal Number
Step 1	10101_2	010 101
Step 2	10101_2	2_8 5_8
Step 3	10101_2	25_8

Binary Number − 10101_2 = Octal Number − 25_8

Shortcut method - Octal to Binary

Steps

- **Step 1** − Convert each octal digit to a 3 digit binary number (the octal digits may be treated as decimal for this conversion).
- **Step 2** − Combine all the resulting binary groups (of 3 digits each) into a single binary number.

Example

Octal Number − 25_8

Calculating Binary Equivalent −

Step	Octal Number	Binary Number
Step 1	25_8	2_{10} 5_{10}
Step 2	25_8	010_2 101_2
Step 3	25_8	010101_2

Octal Number − 25_8 = Binary Number − 10101_2

Shortcut method - Binary to Hexadecimal

 Steps

- **Step 1** − Divide the binary digits into groups of four (starting from the right).
- **Step 2** − Convert each group of four binary digits to one hexadecimal symbol.

Example

Binary Number − 10101_2

Calculating hexadecimal Equivalent –

Step	Binary Number	Hexadecimal Number
Step 1	10101_2	0001 0101
Step	10101_2	1_{10} 5_{10}

2		
Step 3	10101_2	15_{16}

Binary Number − 10101_2 = Hexadecimal Number − 15_{16}

Shortcut method - Hexadecimal to Binary

Steps

- **Step 1** − Convert each hexadecimal digit to a 4 digit binary number (the hexadecimal digits may be treated as decimal for this conversion).
- **Step 2** − Combine all the resulting binary groups (of 4 digits each) into a single binary number.

Example

Hexadecimal Number − 15_{16}

Calculating Binary Equivalent –

Step	Hexadecimal Number	Binary Number
Step 1	15_{16}	1_{10} 5_{10}
Step 2	15_{16}	0001_2 0101_2
Step 3	15_{16}	00010101_2

Hexadecimal Number − 15_{16} = Binary Number − 10101_2

UNIT II

Standard I/O in "C"

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

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

The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files 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 pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

The getchar() and 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 character 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 the 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() and 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 (End of File).

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

NOTE: Though it has been deprecated to use gets() function, Instead of using gets, you want to use fgets().

```
#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 the 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 the input from the standard input stream **stdin** and scans that input according to the **format** provided.

The **int printf(const char *format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the 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. Let us now proceed with a simple example to understand the concepts better –

```
#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() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

Fundamental Data Types and Storage Classes:

Character types

A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class is used to describe the following things:

- The variable scope.

- The location where the variable will be stored.

- The initialized value of a variable.

- A lifetime of a variable.

- Who can access a variable?

Thus a storage class is used to represent the information about a variable.

NOTE: A variable is not only associated with a data type, its value but also a storage class.

There are total four types of standard storage classes. The table below represents the storage classes in 'C'.

Storage class	Purpose
auto	It is a default storage class.
extern	It is a global variable.
static	It is a local variable which is capable of returning a value even when control is transferred to the function call.
register	It is a variable which is stored inside a Register.

- Auto storage class

- Extern storage class

  - First File: main.c
  - Second File: original.c

- Static storage class

- Register storage class

Auto storage class

The variables defined using auto storage class are called as local variables. Auto stands for automatic storage class. A variable is in auto storage class by default if it is not explicitly specified.

The scope of an auto variable is limited with the particular block only. Once the control goes out of the block, the access is destroyed. This means only the block in which the auto variable is declared can access it.

A keyword auto is used to define an auto storage class. By default, an auto variable contains a garbage value.

Example, auto int age;

The program below defines a function with has two local variables

```
int add(void) {
   int a=13;
   auto int b=48;
return a+b;}
```

We take another program which shows the scope level "visibility level" for auto variables in each block code which are independently to each other:

```
#include <stdio.h>
int main( )
{
  auto int j = 1;
  {
    auto int j= 2;
    {
      auto int j = 3;
      printf ( " %d ", j);
    }
    printf ( "\t %d ",j);
  }
  printf( "%d\n", j);}
```

OUTPUT:

```
 3 2 1
```

Extern storage class

Extern stands for external storage class. Extern storage class is used when we have global functions or variables which are shared between two or more files.

Keyword **extern** is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.

The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program. Notice that the extern variable cannot be initialized it has already been defined in the original file

Example, extern void display();

First File: main.c

```
#include <stdio.h>
extern i;
main() {
```

```
printf("value of the external integer is = %d\n", i);
return 0;}
```

Second File: original.c

```
#include <stdio.h>
i=48;
```

Result:

```
value of the external integer is = 48
```

In order to compile and run the above code, follow the below steps

Step 1) Create new project,

1. Select Console Application
2. Click Go
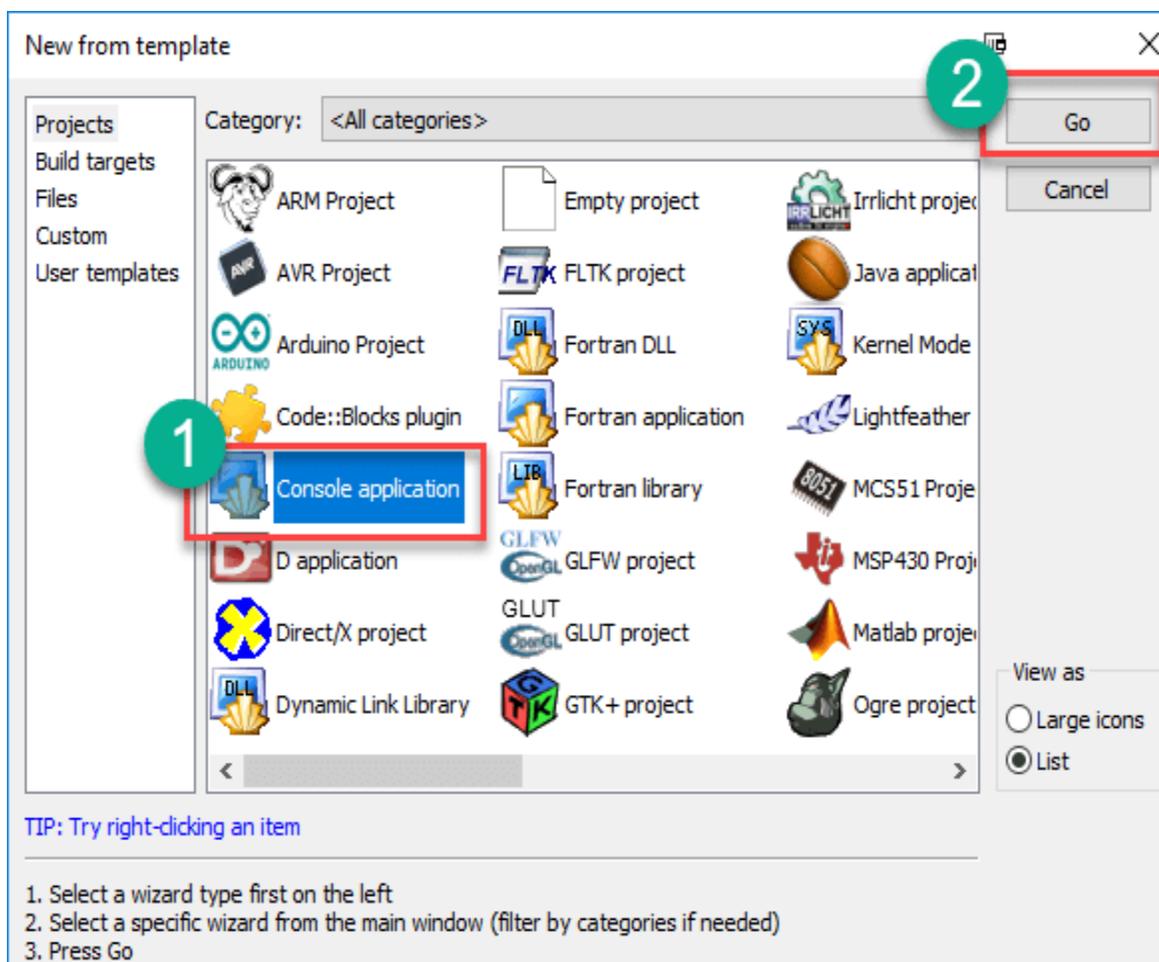
Step 2) Select C and click Next
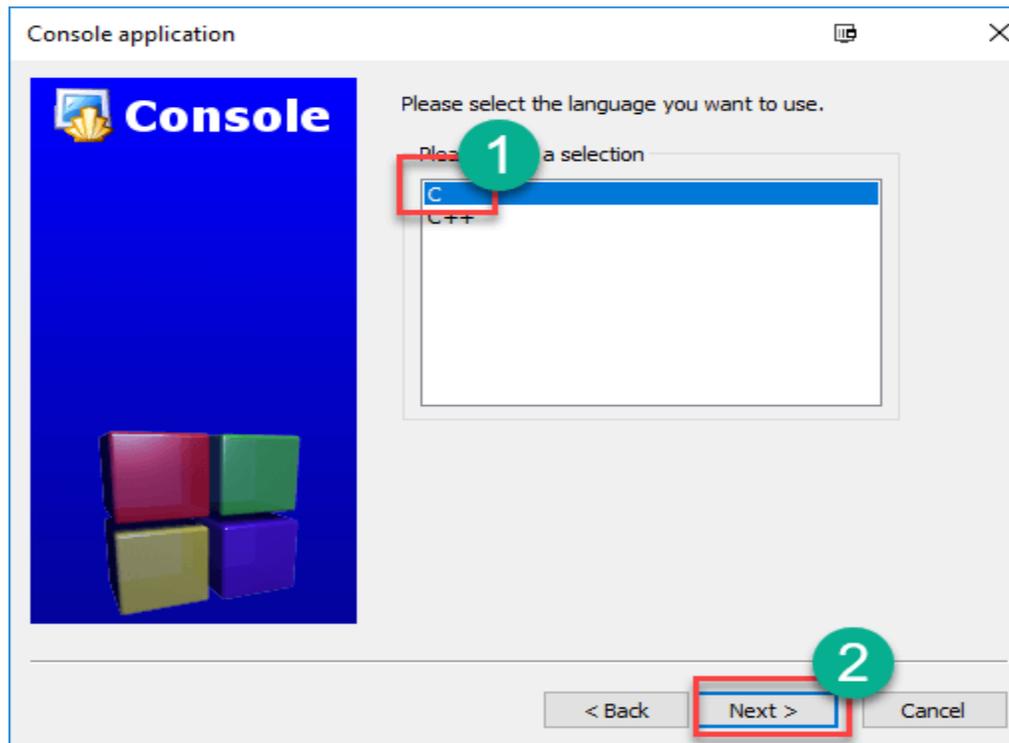
Step 3) Click Next

Step 4) Enter details and click Next
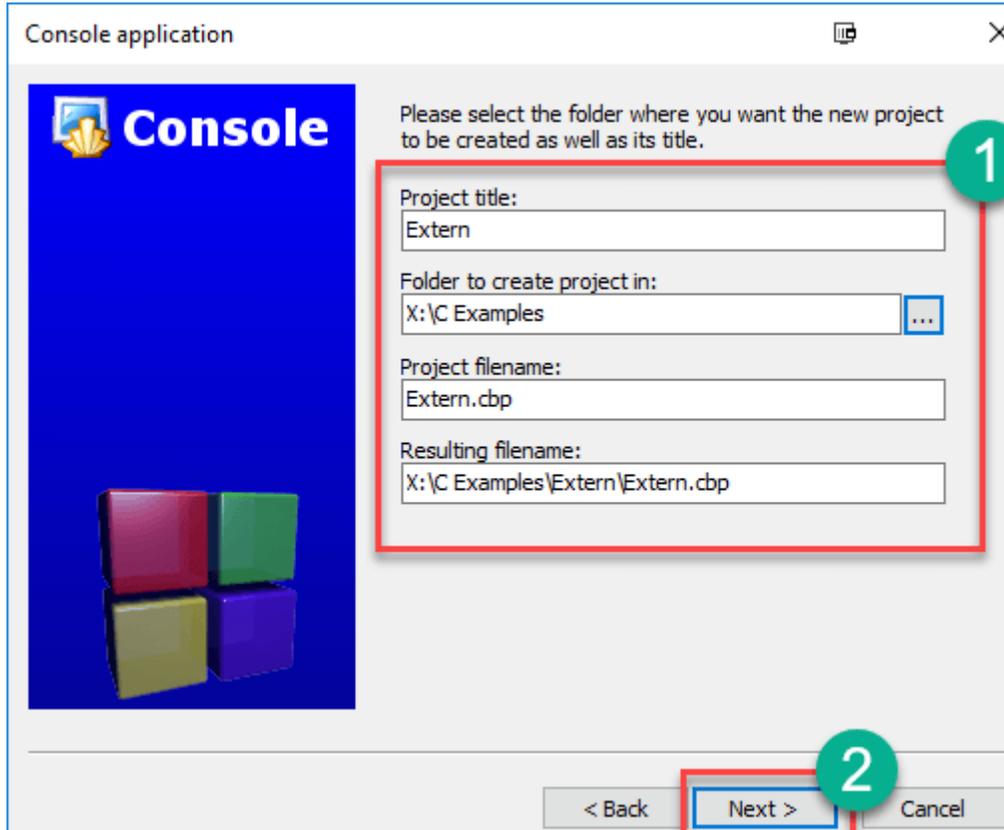
Step 5) Click Finish
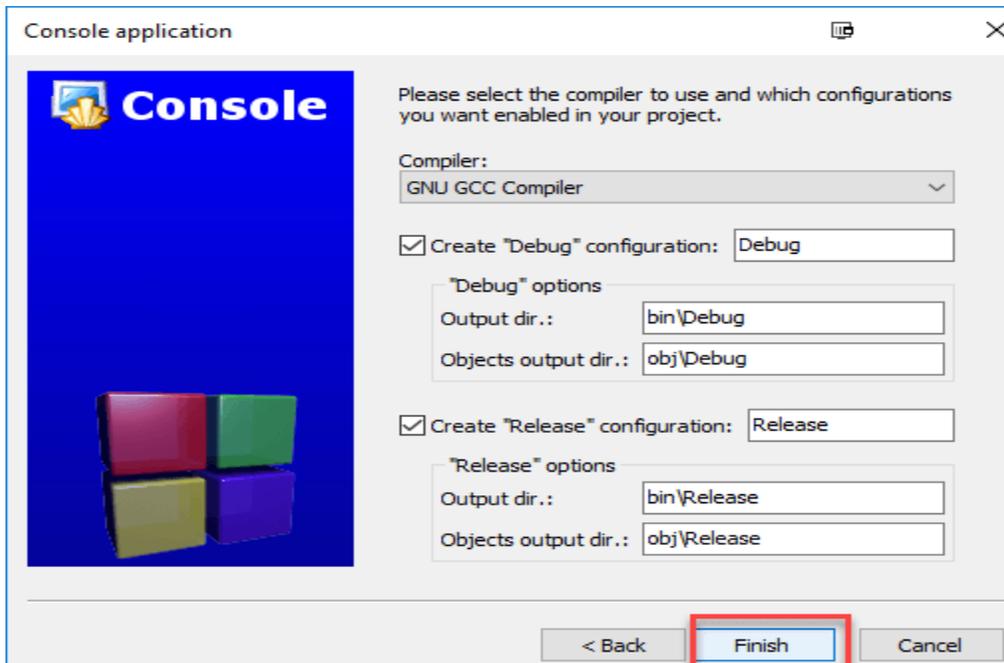
Step 6) Put the main code as shown in the previous program in the main.c file and save it
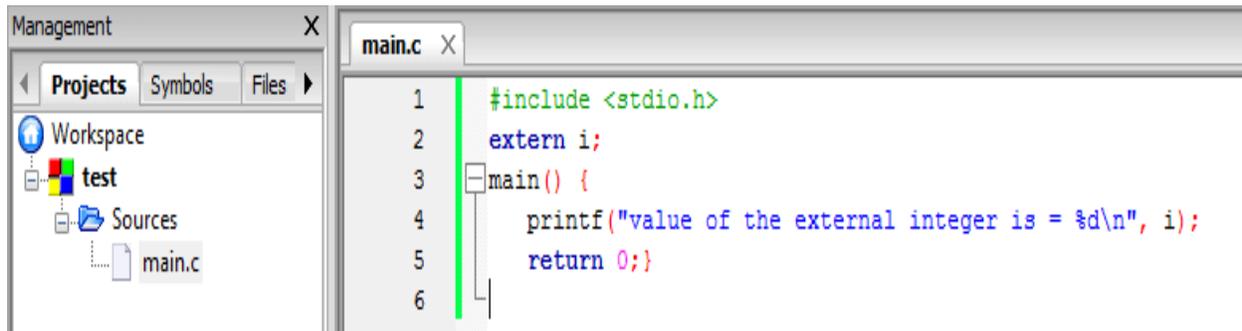
Management   X

◀ **Projects**   Symbols   Files ▶

🌐 Workspace
   test
     Sources
       main.c

main.c   X

```c
1    #include <stdio.h>
2    extern i;
3    main() {
4        printf("value of the external integer is = %d\n", i);
5        return 0;}
6
```

Step 7) Create a new C file [File -> new -> Empty File , save (as original.c) and add it to the current project by clicking "OK" in the dialogue box .

Add file to project   X

? Do you want to add this new file in the active project (has to be saved first)?

Oui   Non
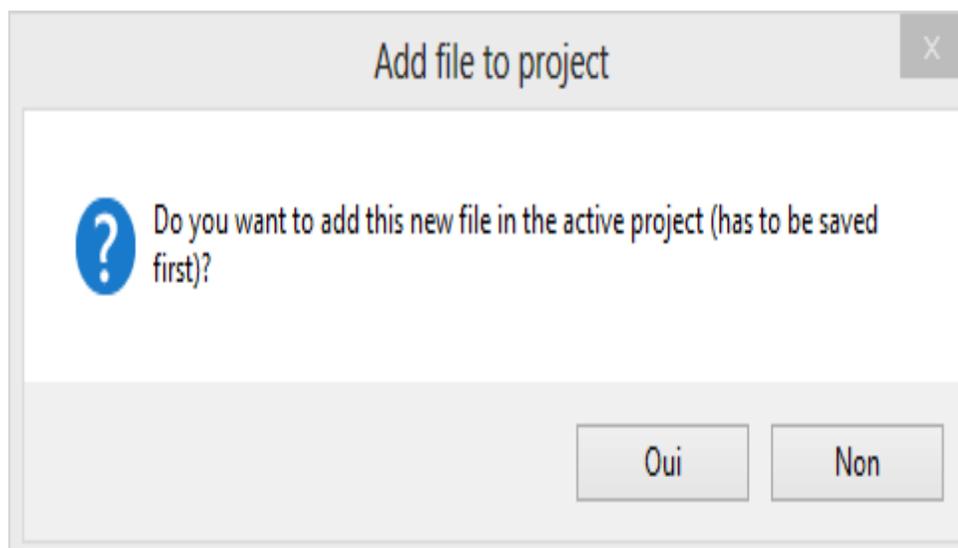
Step 8) Put and save the C code of the original.c file shown in the previous example without the main() function.

Management   X

◀ **Projects**   Symbols   Files ▶

🌐 Workspace
   test
     Sources
       main.c
       original.c

main.c   X   **original.c**   X

```c
1    #include <stdio.h>
2    i=48;
3
4
```

Step 9) Build and run your project. The result is shown in the next figure

```
value of the external integer is = 48
```

Static storage class

The static variables are used within function/ file as local static variables. They can also be used as a global variable

- Static local variable is a local variable that retains and stores its value between function calls or block and remains visible only to the function or block in which it is defined.
- Static global variables are global variables visible **only to the file in which it is declared.**

Example: static int count = 10;

Keep in mind that static variable has a default initial value zero and is initialized only once in its lifetime.

```
#include <stdio.h> /* function declaration */
void next(void);
static int counter = 7; /* global variable */
main() {
 while(counter<10) {
     next();
     counter++;   }
return 0;}
void next( void ) {    /* function definition */
   static int iteration = 13; /* local static variable */
   iteration ++;
   printf("iteration=%d and counter= %d\n", iteration, counter);}
```

Result:

```
iteration=14 and counter= 7
iteration=15 and counter= 8
iteration=16 and counter= 9
```

Global variables are accessible throughout the file whereas static variables are accessible only to the particular part of a code.

The lifespan of a static variable is in the entire program code. A variable which is declared or initialized using static keyword always contains zero as a default value.

79

Register storage class

You can use the register storage class when you want to store local variables within functions or blocks in CPU registers instead of RAM to have quick access to these variables. For example, "counters" are a good candidate to be stored in the register.

Example: register int age;

The keyword **register** is used to declare a register storage class. The variables declared using register storage class has lifespan throughout the program.

It is similar to the auto storage class. The variable is limited to the particular block. The only difference is that the variables declared using register storage class are stored inside CPU registers instead of a memory. Register has faster access than that of the main memory.

The variables declared using register storage class has no default value. These variables are often declared at the beginning of a program.

```
#include <stdio.h> /* function declaration */
main() {
{register int  weight;
int *ptr=&weight ;/*it produces an error when the compilation occurs ,we cannot get a m
emory location when dealing with CPU register*/}
}
```

OUTPUT:

error: address of register variable 'weight' requested

The next table summarizes the principal features of each storage class which are commonly used in C programming

Storage Class	Declaration	Storage	Default Initial Value	Scope	Lifetime
auto	Inside a function/block	Memory	Unpredictable	Within the function/block	Within the function/block
register	Inside a function/block	CPU Registers	Garbage	Within the function/block	Within the function/block

extern	Outside all functions	Memory	Zero	Entire the file and other files where the variable is declared as extern	program runtime
Static (local)	Inside a function/block	Memory	Zero	Within the function/block	program runtime
Static (global)	Outside all functions	Memory	Zero	Global	program runtime

Summary

In this tutorial we have discussed storage classes in C, to sum up:

- A storage class is used to represent additional information about a variable.

- Storage class represents the scope and lifespan of a variable.

- It also tells who can access a variable and from where?

- Auto, extern, register, static are the four storage classes in 'C'.

- auto is used for a local variable defined within a block or function

- register is used to store the variable in CPU registers rather memory location for quick access.

- Static is used for both global and local variables. Each one has its use case within a C program.

- Extern is used for data sharing between C project files.

Integer

Every programming language has in-built types to differentiate between the nature of various data (input or output or intermediate). Integer is a common data type which is widely use in general programming and in scientific computing.

Integer is defined as a number which has no fractional component. Numbers which have a fractional component is known floating point numbers. Despite the fact that floating point numbers can represent numbers accurately, integers have their own place in the world of computing due to:

- Integers consumes significantly less space than Floating point numbers

- Calculations using integers are much faster (over 2 times) due to hardware architecture

  In C programming language, integer data is represented by its own datatype known as **int**. It has several variants which differs based on memory consumption includes:

- int

- long

- short

- long long

  Usage
  In C, one can define an integer variable as:

```c
int main()
{
   int a = 1;
   short b = 1;
   long c = 1;
   long long d = 1;
}
C
Copy
```

Signed and Unsigned version
As the range of numbers determined by a datatype like int is limited and both negative and positive numbers are required, we have two options:

- signed integers: range is equally divided among negative and positive numbers (including 0)

- unsigned integers: range starts from 0 to the upper positive number limit

  Hence, unsigned integers are used when:

- negative numbers are not required

- increase the range of positive number by double

  One can defined an unsigned integer by placing the keyword **unsigned** before the usual declaration/ initialization like:

```C
int main()
{
   unsigned int a = 1;
   unsigned long b = 1;
}
```
Copy

  The default declaration is the signed version **signed**.
  Hence, there are 8 possible types for integer:

- int

- unsigned int

- short

- unsigned short

- long

- unsigned long

- long long

- unsigned long long

Format specifier

To print a value in C using printf, one needs to specify the datatype of the data to be printed. The format specifier of each variant of integer datatype is different in C.

For instance, int datatype has %d as the format specifier.

Following code demonstrates the idea:

```c
int main()
{
  unsigned int a = 1;
  int b = 1;
  unsigned long c = 1;
  long long d = 1;
  printf("%u", a);
  printf("%d", b);
  printf("%lu", c);
  printf("%lld", d);
}
```
C
Copy

Range and memory consumption

One can find the memory consumed by a data type as follows:

```c
int main()
{
   printf("size of int : %d\n",sizeof(int));
   printf("size of signed int : %d\n",sizeof(signed int));
   printf("size of unsigned long : %d\n",sizeof(unsigned long));
   return 0;
}
```
C
Copy

Ideally, memory consumed by the signed and unsigned variants are the same. It only differs in the range.

If Integer data type int is of 4 bytes, then the range is calculated as follows:

4 bytes = 4 X 8 = 32 bits

Each bit can store 2 values (0 and 1)

Hence, integer data type can hold 2^{32} values

In signed version, the most significant bit is reserved for sign. So, 0 denotes positive number and 1 denotes negative number.

Hence

- range of unsigned int is **0 to 2^32-1**
- range of signed int is **-2^31 to 2^31-1**

The exact value of memory and range depends on the hardware but remains same across several hardware types. Following table summarizes the values:

DATA TYPE	SIZE (IN BYTES)	RANGE	FORMAT SPECIFIER
int	4	-2147483648 to 2147483647	%d
unsigned int	4	0 to 4294967295	%u
short	2	-32768 to 32767	%hd
unsigned short	2	0 to 65535	%hu
long	8	-9223372036854775808 to 9223372036854775807	%ld
unsigned long	8	0 to 18446744073709551615	%lu
long long	8	-9223372036854775808 to 9223372036854775807	%lld
unsigned long long	8	0 to 18446744073709551615	%llu

In some platforms, long long and long refer to the same size but in other platforms, long long can be double the size of long.

In general, the rules are:

- signed and unsigned version will have the same size

- size of int is 4 bytes

- size of short <= size of int

- size of int <= size of long

- size of long <= size of long long

Integer overflow

As we have seen that each integer datatype has a fixed range beyond which it will fail. In case, a number falls beyond the range of a datatype, then the code will wrap around to give an errorneous result.

Consider the case of int where the range is **-2147483648 to 2147483647**.
Key points to note in case of signed int are:

- The number after 2147483647 is -2147483648.

- The number after -2147483648 is -2147483647

- 2147483648 is represented as -2147483648 as it is wrapped around

Short

Short int is integer variable. They are used to store value upto -32,768 to 32,767. The %hd is a short int placeholder(format specifier) that will be replaced by the value of short integer variable 'a' when the printf statement is executed.

Program

```
Short int is a integer variable.
They are used to store value upto -32,768 to 32,767.

#include <stdio.h>
                                    The %hd is a short int placeholder(format specifier)
int main()                          that will be replaced by the value of short integer
{                                   variable 'a' when the printf statement is executed.
    short int a=32000;

    printf("Value of 'a' = %hd",a);
    return 0;
}
```

```
program output
Value of 'a' = 32000
```

Program Source

```
#include <stdio.h>



int main()

{

    short int a=32000;



    printf("Value of 'a' = %hd",a);

    return 0;

}
```

Long

Long is a data type used in programming languages, such as Java, C++, and C#.
A constant or variable defined as long can store a single 64-bit signed integer.

So what constitutes a 64-bit signed integer? It helps to break down each word, starting from right to left. An integer is a whole number that does not include a decimal point. Examples include 1, 99, or 234536. "Signed" means the number can be either positive or negative, since it may be preceded by a minus (-) symbol. 64-bit means the number can store 2^{63} or 18,446,744,073,709,551,616 different values (since one bit is used for the sign). Because the long data type is signed, the possible integers range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, including 0.

In modern programming languages, the standard integer (int) data type typically stores a 32-bit whole number. Therefore, if a variable or constant may potentially store a

number larger than 2,147,483,647 ($2^{31} \div 2$), it should be defined as a long instead of an int.

Unsigned

char is the most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
Now character datatype can be divided into 2 types:

1. signed char

2. unsigned char
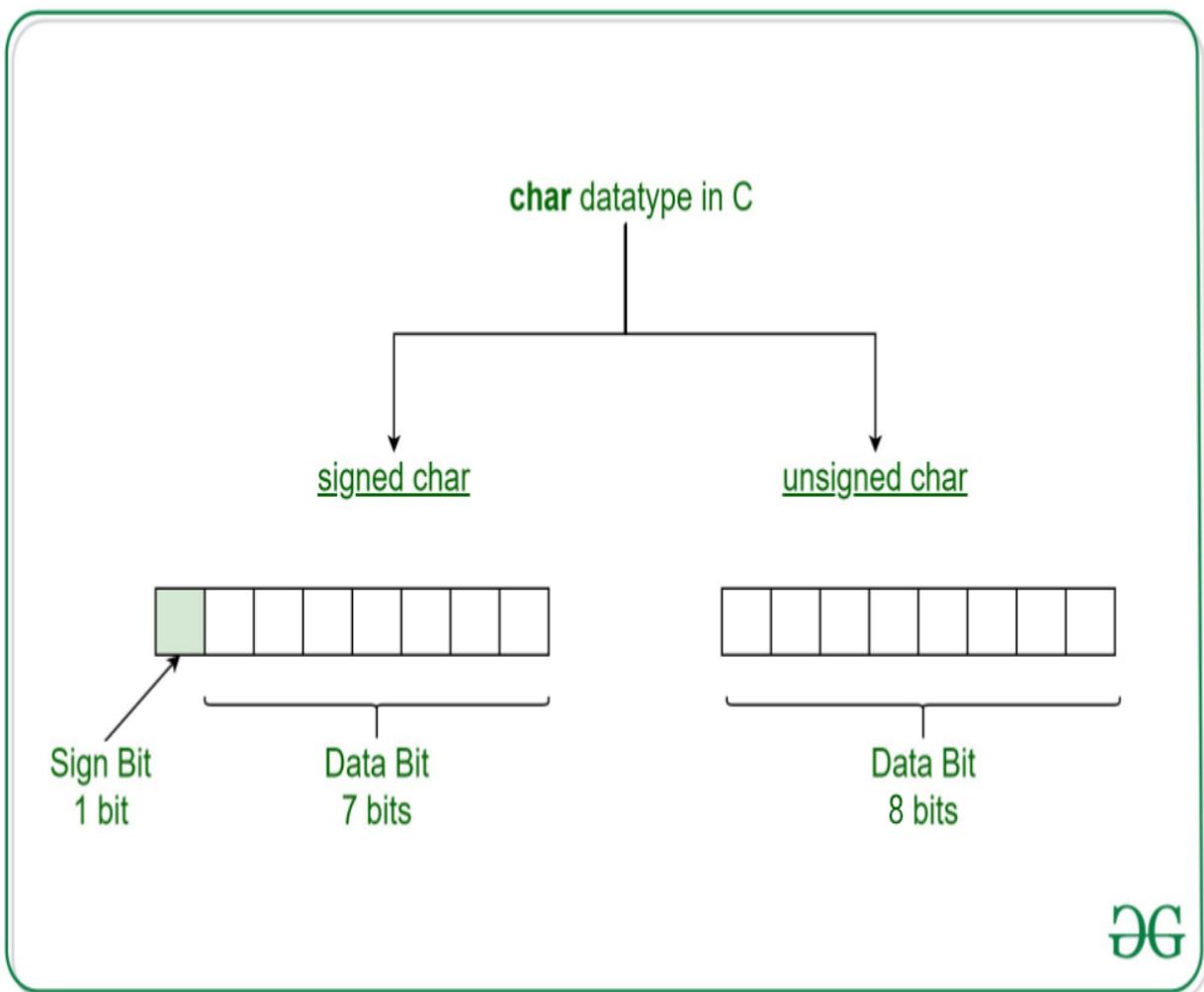
unsigned char is a character datatype where the variable consumes all the 8 bits of the memory and there is no sign bit (which is there in signed char). So it means that the range of unsigned char data type ranges from 0 to 255.

Syntax:

```
unsigned char [variable_name] = [value]
```

Example:

```
unsigned char ch = 'a';
```

- **Initializing an unsigned char:** Here we try to insert a char in the unsigned char variable with the help of ASCII value. So the ASCII value 97 will be converted to a character value, i.e. 'a' and it will be inserted in unsigned char.
  filter_none
  edit
  play_arrow
  brightness_4

  ```c
  // C program to show unsigned char

  #include <stdio.h>

  int main()
  {

      int chr = 97;
      unsigned char i = chr;
      printf("unsigned char: %c\n", i);

      return 0;
  }
  ```

  Output:

  ```
  unsigned char: a
  ```

  Initializing an unsigned char with signed value: Here we try to insert a char in the unsigned char variable with the help of ASCII value. So the ASCII value -1 will be first converted to a range 0-255 by rounding. So it will be 255. Now, this value will be converted to a character value, i.e. 'ÿ' and it will be inserted in unsigned char.
  filter_none
  edit
  play_arrow
  brightness_4

```c
// C program to show unsigned char

#include <stdio.h>

int main()
{

    int chr = -1;
    unsigned char i = chr;
    printf("unsigned char: %c\n", i);

    return 0;
}
```

Output:

unsigned char: ÿ

single and double-precision floating point

There are a few different ways to think about pi. As apple, pumpkin and key lime … or as the different ways to represent the mathematical constant of ▯, 3.14159, or, in binary, a long line of ones and zeroes.

An irrational number, pi has decimal digits that go on forever without repeating. So when doing calculations with pi, both humans and computers must pick how many decimal digits to include before truncating or rounding the number.

In grade school, one might do the math by hand, stopping at 3.14. A high schooler's graphing calculator might go to 10 decimal places — using a higher level of detail to express the same number. In computer science, that's called precision. Rather than decimals, it's usually measured in bits, or binary digits.

For complex scientific simulations, developers have long relied on high-precision math to understand events like the Big Bang or to predict the interaction of millions of atoms.

Having more bits or decimal places to represent each number gives scientists the flexibility to represent a larger range of values, with room for a fluctuating number of digits on either side of the decimal point during the course of a computation. With this range, they can run precise calculations for the largest galaxies and the smallest particles.

But the higher precision level a machine uses, the more computational resources, data transfer and memory storage it requires. It costs more and it consumes more power.

Since not every workload requires high precision, AI and HPC researchers can benefit by mixing and matching different levels of precision. NVIDIA Tensor Core GPUs support multi- and mixed-precision techniques, allowing developers to optimize computational resources and speed up the training of AI applications and those apps' inferencing capabilities.

Difference Between Single-Precision, Double-Precision and Half-Precision

Floating-Point Format

The IEEE Standard for Floating-Point Arithmetic is the common convention for representing numbers in binary on computers. In double-precision format, each number takes up 64 bits. Single-precision format uses 32 bits, while half-precision is just 16 bits.

To see how this works, let's return to pi. In traditional scientific notation, pi is written as 3.14×10^0. But computers store that information in binary as a floating-point, a series of ones and zeroes that represent a number and its corresponding exponent, in this case 1.1001001×2^1.
In single-precision, 32-bit format, one bit is used to tell whether the number is positive or negative. Eight bits are reserved for the exponent, which (because it's binary) is 2 raised to some power. The remaining 23 bits are used to represent the digits that make up the number, called the significand.

Double precision instead reserves 11 bits for the exponent and 52 bits for the significand, dramatically expanding the range and size of numbers it can represent. Half precision takes an even smaller slice of the pie, with just five for bits for the exponent and 10 for the significand.

Here's what pi looks like at each precision level:

Double precision	0100000000001001001000011111101101010100010001000010110100011000
Single precision	01000000100100100001111111011011
Half precision	0100001001001000

Difference Between Multi-Precision and Mixed-Precision Computing

Multi-precision computing means using processors that are capable of calculating at different precisions — using double precision when needed, and relying on half- or single-precision arithmetic for other parts of the application.

Mixed-precision, also known as transprecision, computing instead uses different precision levels within a single operation to achieve computational efficiency without sacrificing accuracy.

In mixed precision, calculations start with half-precision values for rapid matrix math. But as the numbers are computed, the machine stores the result at a higher precision. For instance, if multiplying two 16-bit matrices together, the answer is 32 bits in size.

With this method, by the time the application gets to the end of a calculation, the accumulated answers are comparable in accuracy to running the whole thing in double-precision arithmetic.

This technique can accelerate traditional double-precision applications by up to 25x, while shrinking the memory, runtime and power consumption required to run them. It can be used for AI and simulation HPC workloads.

As mixed-precision arithmetic grew in popularity for modern supercomputing applications, HPC luminary Jack Dongarra outlined a new benchmark, HPL-AI, to estimate the performance of supercomputers on mixed-precision calculations. When NVIDIA ran HPL-AI computations in a test run on Summit, the fastest supercomputer in the world, the system achieved unprecedented performance levels of nearly 550 petaflops, over 3x faster than its official performance on the TOP500 ranking of supercomputers.

How to Get Started with Mixed-Precision Computing

NVIDIA Volta and Turing GPUs feature Tensor Cores, which are built to simplify and accelerate multi- and mixed-precision computing. And with just a few lines of code, developers can enable the automatic mixed-precision feature in the TensorFlow, PyTorch and MXNet deep learning frameworks. The tool gives researchers speedups of up to 3x for AI training.

The NGC catalog of GPU-accelerated software also includes iterative refinement solver and cuTensor libraries that make it easy to deploy mixed-precision applications for HPC.

For more information, check out our developer resources on training with mixed precision.

What Is Mixed-Precision Used for?

Researchers and companies rely on the mixed-precision capabilities of NVIDIA GPUs to power scientific simulation, AI and natural language processing workloads. A few examples:

Earth Sciences

- Researchers from the University of Tokyo, Oak Ridge National Laboratory and the Swiss National Supercomputing Center used AI and mixed-precision techniques for earthquake simulation. Using a 3D simulation of the city of Tokyo, the scientists modeled how a seismic wave would impact hard soil, soft soil, above-ground buildings, underground malls and subway systems. They achieved a 25x speedup with their new model, which ran on the Summit supercomputer and used a combination of double-, single- and half-precision calculations.
- A Gordon Bell prize-winning team from Lawrence Berkeley National Laboratory used AI to identify extreme weather patterns from high-resolution climate simulations, helping scientists analyze how extreme weather is likely to change in the future. Using the mixed-precision capabilities of NVIDIA V100 Tensor Core GPUs on Summit, they achieved performance of 1.13 exaflops.

Medical Research and Healthcare

- San Francisco-based Fathom, a member of the NVIDIA Inception virtual accelerator program, is using mixed-precision computing on NVIDIA V100 Tensor Core GPUs to speed up training of its deep learning algorithms, which automate medical coding. The startup works with many of the largest medical coding operations in the U.S., turning doctors' typed notes into alphanumeric codes that represent every diagnosis and procedure insurance providers and patients are billed for.
- Researchers at Oak Ridge National Laboratory were awarded the Gordon Bell prize for their groundbreaking work on opioid addiction, which leveraged mixed-precision techniques to achieve a peak throughput of 2.31 exaops. The research analyzes genetic variations within a population, identifying gene patterns that contribute to complex traits.

Nuclear Energy

- Nuclear fusion reactions are highly unstable and tricky for scientists to sustain for more than a few seconds. Another team at Oak Ridge is simulating these reactions to give physicists more information about the variables at play within the reactor. Using mixed-

precision capabilities of Tensor Core GPUs, the team was able to accelerate their simulations by 3.5x.

storage classes

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

- o Automatic
- o External
- o Static
- o Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

Automatic

- o Automatic variables are allocated memory automatically at runtime.
- o The visibility of the automatic variables is limited to the block in which they are defined.

The scope of the automatic variables is limited to the block in which they are defined.

- o   The automatic variables are initialized to garbage by default.
- o   The memory assigned to automatic variables gets freed upon exiting from the block.
- o   The keyword used for defining automatic variables is auto.
- o   Every local variable is automatic in C by default.

Example 1
1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a; //auto
5. **char** b;
6. **float** c;
7. printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
8. **return** 0;
9. }

Output:

garbage garbage garbage

Example 2
1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a = 10,i;
5. printf("%d ",++a);
6. {
7. **int** a = 20;
8. **for** (i=0;i<3;i++)
9. {
10. printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
11. }
12. }
13. printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
14. }

Output:

11 20 20 20 11

Static

- o The variables defined as static specifier can hold their value between the multiple function calls.
- o Static local variables are visible only to the function or the block in which they are defined.
- o A same static variable can be declared many times but can be assigned at only one time.
- o Default initial value of the static integral variable is 0 otherwise null.
- o The visibility of the static global variable is limited to the file in which it has declared.
- o The keyword used to define static variable is static.

Example 1
1. #include<stdio.h>
2. **static char** c;
3. **static int** i;
4. **static float** f;
5. **static char** s[100];
6. **void** main ()
7. {
8. printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printed.
9. }

Output:

0 0 0.000000 (null)

Example 2
1. #include<stdio.h>
2. **void** sum()
3. {
4. **static int** a = 10;
5. **static int** b = 24;
6. printf("%d %d \n",a,b);
7. a++;
8. b++;

9. }
10. **void** main()
11. {
12. **int** i;
13. **for**(i = 0; i< 3; i++)
14. {
15. sum(); // The static variables holds their value between multiple function calls.
16. }
17. }

Output:

```
10 24
11 25
12 26
```

Register

- o   The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- o   We can not dereference the register variables, i.e., we can not use &operator for the register variable.
- o   The access time of the register variables is faster than the automatic variables.
- o   The initial default value of the register local variables is 0.
- o   The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler?s choice whether or not; the variables can be stored in the register.
- o   We can store pointers into the register, i.e., a register can store the address of a variable.
- o   Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Example 1
1. #include <stdio.h>
2. **int** main()
3. {
4. **register int** a; // variable a is allocated memory in the CPU register. The initial default v alue of a is 0.
5. printf("%d",a);
6. }

Output:

0

Example 2

1. #include <stdio.h>
2. **int** main()
3. {
4. **register int** a = 0;
5. printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.
6. }

Output:

main.c:5:5: error: address of register variable ?a? requested
printf("%u",&a);
^~~~~~

External

- o The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- o The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- o The default initial value of external integral type is 0 otherwise null.
- o We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- o An external variable can be declared many times but can be initialized at only once.
- o If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

1. #include <stdio.h>
2. **int** main()
3. {
4. **extern int** a;
5. printf("%d",a);

6. }

Output

main.c:(.text+0x6): undefined reference to `a'
collect2: error: ld returned 1 exit status

Example 2

1. #include <stdio.h>
2. **int** a;
3. **int** main()
4. {
5. **extern int** a; // variable a is defined globally, the memory will not be allocated to a
6. printf("%d",a);
7. }

Output

0

Example 3

1. #include <stdio.h>
2. **int** a;
3. **int** main()
4. {
5. **extern int** a = 0; // this will show a compiler error since we can not use extern and initiali
    zer at same time
6. printf("%d",a);
7. }

Output

compile time error
main.c: In function ?main?:
main.c:5:16: error: ?a? has both ?extern? and initializer
extern int a = 0;

Example 4

1. #include <stdio.h>
2. **int** main()
3. {
4. **extern int** a; // Compiler will search here for a variable a defined and initialized somewh
    ere in the pogram or not.
5. printf("%d",a);

6. }
7. **int** a = 20;

Output

20

Example 5

1. **extern int** a;
2. **int** a = 10;
3. #include <stdio.h>
4. **int** main()
5. {
6. printf("%d",a);
7. }
8. **int** a = 20; // compiler will show an error at this line

Output

compile time error

automatic

The variables which are declared inside a block are known as **automatic** or **local variables**; these variables allocates memory automatically upon entry to that block and free the occupied memory upon exit from that block.

These variables have local scope to that block only that means these can be accessed in which variable declared.

Keyword 'auto' may be used to declare automatic variable but we can declare these variable without using 'auto' keywords.

Consider the following declarations

int main()

{

        auto int a;

        int b;

```
        ....

        return 0;

   }
```

Here, both variables a and b are automatic variables.

Automatic variables in other user defined functions

An automatic or local variable can be declared in any user define function in the starting of the block.

Consider the following code

```
void myFunction(void)
{
        int x;
        float y;
        char z;
        ...
}

int main()
{
        int a,b;
        myFunction();
        ....
        return 0;
}
```

In this code snippet, variables x, y and z are the local/automatic variable of myFunction() function, while variables a and b are the local/automatic variables of main() function.

Register

Register variables tell the compiler to store the variable in CPU register instead of memory. Frequently used variables are kept in registers and they have faster accessibility. We can never get the addresses of these variables. "register" keyword is used to declare the register variables.

Scope − They are local to the function.

Default value − Default initialized value is the garbage value.

Lifetime − Till the end of the execution of the block in which it is defined.

Here is an example of register variable in C language,

Example

```
#include <stdio.h>

int main() {

   register char x = 'S';

   register int a = 10;

   auto int b = 8;

   printf("The value of register variable b : %c\n",x);

   printf("The sum of auto and register variable : %d",(a+b));

   return 0;

}
```

Output

```
The value of register variable b : S
The sum of auto and register variable : 18
```

Register keyword can be used with pointer also. It can have address of memory location. It will not create any error.

Here is an example of register keyword in C language

Example

```
#include<stdio.h>

int main() {

   int i = 10;

   register int *a = &i;

   printf("The value of pointer : %d", *a);

   getchar();

   return 0;
```

```
}
```

Output

The value of pointer : 10

static and external

In C, variable declaration & definition are implicitly tied together.

Here, definition = storage allocation + possible initialization.

By default, functions and global variables are visible within all linked files.

"extern" keyword allows for declaration sans definition.

But, this would mean that global variables are visible from everywhere.

So, **"static" keyword lets us limit the visibility of things within the same file.** A static global variable or a static function is "seen" only in the file it's declared in (so that the user won't be able to access them. This is encapsulation, a good practice). Thus, **"static" forces the lifetime of variables to be equivalent to global.**

This means that static variables are stored in static memory as opposed to stack. Thus, having a static variable inside a function would keeps its value between function invocations. This could lead to code being not thread-safe and harder to understand.

sum.c

```c
int sumWithI(int x, int y) {
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
sum.c: In function 'sumWithI':
sum.c:2: error: 'I' undeclared (first use in this function)
main.c: In function 'main':
main.c:6: warning: implicit declaration of function 'sumWithI'
```

Here, variable "I" is not found in file sum.c

sum.c

```c
int sumWithI(int x, int y) {
    int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
-1073743741
```

Here, variable "I" declared in sumWithI function has garbage value or zero initial value [depending on system]

sum.c

```c
int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
13
```

extern keyword helps us find the variable "I" from main.c

sum.c

```c
int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

static int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
Undefined symbols:
  "_I", referenced from:
      _sumWithI in ccmvi0RF.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

the static variable initialized in main.c does not allow the variable to be visible outside the file.
the function sumWithI is made static and thus cannot be accessed from main.c

the static variable inside a function holds its value on successive function calls.

Recap: By default, variables declared inside functions have local lifetimes (stack-bound) and using "static" lets us change their storage class to static (aka "global")

Operators and Expressions:

Using numeric and relational operators

Caché supports many different operators, which perform various actions, including mathematical actions, logical comparisons, and so on. Operators act on expressions, which are variables or other entities that ultimately evaluated to a value. This chapter describes expressions and the various ObjectScript operators. It contains the following topics:

- Introduction to Operators and Expressions

- String-to-Number Conversion

- Arithmetic Operators

- Logical Comparison Operators

- String Concatenate Operator

- Numeric Relational Operators

- String Relational Operators

- Pattern Matching

- Indirection

Introduction to Operators and Expressions

Operators are symbolic characters that specify the action to be performed on their associated operands. Each operand consists of one or more expressions or expression atoms. When used together, an operator and its associated operands have the following form:

[operand] operator operand

Some operators take only one operand and are known as unary operators; others take two operands and are known as binary operators.

An operator and any of its operands taken together constitute an expression. Such expressions produce a result that is the effect of the operator on the operand(s). They are classified based on the types of operators they contain.

- An arithmetic expression contains arithmetic operators, gives a numeric interpretation to the operands, and produces a numeric result.

- A string expression contains string operators, gives a string interpretation to the operands, and produces a string result.

- A logical expression contains relational and logical operators, gives a logical interpretation to the operands, and produces a boolean result: TRUE (1) or FALSE (0).

Table of Operator Symbols

ObjectScript includes the following operators:

ObjectScript Operators

Operator	Operation Performed
.	Object property or method access.
()	Array index or function call arguments.
+	Addition (Binary), Positive (Unary)
–	Subtraction (Binary), Negative (Unary)
*	Multiplication
/	Division

\	Integer division
**	Exponentiation
#	Modulus (remainder)
_	Concatenation
'	Logical complement (NOT)
=	Test for equality, Assignment
'=	Test for non-equality
>	Greater than
'> <=	Not greater than (less than or equal to)
<	Less than
'< >=	Not less than (greater than or equal to)
[	Contains
]	Follows
]]	Sorts After
& &&	Logical AND (&& is "short-circuit" AND)
! ||	Logical OR (|| is "short-circuit" OR)

@	Indirection
?	Pattern Match

These are described in more detail in the following sections.

Operator Precedence

Operator precedence in ObjectScript is strictly left-to-right; within an expression operations are performed in the order in which they appear. This is different from other languages in which certain operators have higher precedence than others. You can use explicit parentheses within an expression to force certain operations to be carried ahead of others.

```
WRITE "1 + 2 * 3 = ", 1 + 2 * 3,!  // returns 9
WRITE "2 * 3 + 1 = ", 2 * 3 + 1,!  // returns 7
WRITE "1 + (2 * 3) = ", 1 + (2 * 3),!  // returns 7
WRITE "2 * (3 + 1) = ", 2 * (3 + 1),!  // returns 8
```

Unary Negative Operators

ObjectScript gives the unary negative operator precedence over the binary arithmetic operators. ObjectScript first scans a numeric expression and performs any unary negative operations. Then, ObjectScript evaluates the expression and produces a result.

```
WRITE -123 - 3,!      // returns -126
WRITE -123 + -3,!     // returns -126
WRITE -(123 - 3),!    // returns -120
```

Parentheses and Precedence

You can change the order of evaluation by nesting expressions within each other with matching parentheses. The parentheses group the enclosed expressions (both arithmetic and relational) and control the order in which ObjectScript performs operations on the expressions. Consider the following expression:

```
SET TorF = ((4 + 7) > (6 + 6)) // False (0)
WRITE TorF
```

Here, because of the parentheses, four and seven are added, as are six and six; this results in the logical expression **11 > 12**, which is false. Compare this to:

```
SET Value = (4 + 7 > 6 + 6) // 7
WRITE Value
```

In this case, precedence proceeds from left to right, so four and seven are added. Their sum, eleven, is compared to six; since eleven is greater than six, the result of this logical operation is one (TRUE). One is then added to six, and the result is seven.

Note that the precedence even determines the result type, since the first expression's final operation results in a boolean and the second expression's final operation results in a numeric.

The following example shows multiple levels of nesting:

```
WRITE 1+2*3-4*5,!  // returns 25
WRITE 1+(2*3)-4*5,!  // returns 15
WRITE 1+(2*(3-4))*5,!  // returns -5
WRITE 1+(((2*3)-4)*5),!  // returns 11
```

Precedence from the innermost nested expression and proceeds out level by level, evaluating left to right at each level.

Tip:

> For all but the simplest ObjectScript expressions, it is good practice to fully parenthesize expressions. This is to eliminate any ambiguity about the order of evaluation and to also eliminate any future questions about the original intention of the code.

For example, because the "&&" operator, like all operators, is subject to left-to-right precedence, the final statement in the following code fragment evaluates to 0:

```
SET x = 3
SET y = 2
IF x && y = 2 {
  WRITE "True",! }
ELSE {
  WRITE "False",! }
```

This is because the evaluation occurs as follows:

1. The first action is to check if x is defined and has a non-zero value. Since x equals 3, evaluation continues.
2. Next, there is a check if y is defined and has a non-zero value. Since y equals 2, evaluation continues.
3. Next, the value of **3 && 2** is evaluated. Since neither 3 nor 2 equal 0, this expression is true and evaluates to 1.
4. The next action is to compare the returned value to 2. Since 1 does not equal 2, this evaluation returns 0.

For those accustomed to many programming languages, this is an unexpected result. If the intent is to return True if x is defined with a non-zero value and if y equals 2, then parentheses are required:

```
SET x = 3
SET y = 2
IF x && (y = 2) {
  WRITE "True",! }
ELSE {
  WRITE "False",! }
```

Functions and Precedence

Some types of expressions, such as functions, can have side effects. Suppose you have the following logical expression:

```
IF var1 = ($$ONE + (var2 * 5)) {
   DO ^Test
}
```

ObjectScript first evaluates var1, then the function **$$ONE**, then var2. It then multiplies var2 by 5. Finally, ObjectScript tests to see if the result of the addition is equal to the value in var1. If it is, it executes the DO command to call the **Test** routine.

As another example, consider the following logical expression:

```
SET var8=25,var7=23
IF var8 = 25 * (var7 < 24) {
  WRITE !,"True" }
ELSE {
  WRITE !,"False" }
```

Caché evaluates expressions strictly left-to-right. The programmer must use parentheses to establish any precedence. In this case, Caché first evaluates var8=25, resulting in 1. It then multiplies this 1 by the results of the expression in parentheses. Because var7 is less than 24, the expression in parentheses evaluates to 1. Therefore, Caché multiplies 1 * 1, resulting in 1 (true).

Expressions

An ObjectScript expression is one or more "tokens" that can be evaluated to yield a value. The simplest expression is simply a literal or variable:

```
SET expr = 22
SET expr = "hello"
SET expr = x
```

You can create more complex expressions using arrays, operators, or one of the many ObjectScript functions:

```
SET expr = +x
SET expr = x + 22
SET expr = array(1)
SET expr = ^data("x",1)
SET expr = $Length(x)
```

An expression may consist of, or include, an object property, instance method call, or class method call:

```
SET expr = person.Name
SET expr = obj.Add(1,2)
SET expr = ##class(MyApp.MyClass).Method()
```

You can directly invoke an ObjectScript routine call within an expression by placing $$ in front of the routine call:

```
SET expr = $$MyFunc^MyRoutine(1)
```

Expressions can be classified according to what kind of value they return:

- An arithmetic expression contains arithmetic operators, gives a numeric interpretation to the operands, and produces a numeric result:

- SET expr = 1 + 2
- SET expr = +x
- SET expr = a + b

Note that a string used within an arithmetic expression is evaluated as a numeric value (or 0 if it is not a valid numeric value). Also note that using the unary addition operator (+) will implicitly convert a string value to a numeric value.

- A string expression contains string operators, gives a string interpretation to the operands, and produces a string result.
- SET expr = "hello"
- SET expr = "hello" _ x

- A logical expression contains relational and logical operators, gives a logical interpretation to the operands, and produces a boolean result: TRUE (1) or FALSE (0):
- SET expr = 1 && 0
- SET expr = a && b
- SET expr = a > b

- An object expression produces an object reference as a result:
- SET expr = object
- SET expr = employee.Company
  SET expr = ##class(Person).%New()

Logical Expressions

Logical expressions use logical operators, numeric relational operators, and string relational operators. They evaluate expressions and result in a Boolean value: 1 (TRUE) or 0 (FALSE). Logical expressions are most commonly used with:

- The IF command
- The $SELECT function
- Postconditional Expressions

In a Boolean test, any expression that evaluates to a non-zero numeric value returns a Boolean 1 (TRUE) value. Any expression that evaluates to a zero numeric value returns a Boolean 0 (FALSE) value. Caché evaluates a non-numeric string as having a zero numeric value. For further details, refer to String-to-Number Conversion.

You can combine multiple Boolean logical expressions by using logical operators. Like all Caché expressions, they are evaluated in strict left-to-right order. There are two types of logical operators: regular logical operators (& and !) and short-circuit logical operators (&& and ||).

When regular logical operators are used to combine logical expressions, Caché evaluates all of the specified expressions, even when the Boolean result is known before all of the expressions have been evaluated. This assures that all expressions are valid.

When short-circuit logical operators are used to combine logical expressions, Caché evaluates only as many expressions as are needed to determine the Boolean result. For example, if there are multiple AND tests, the first expression that returns 0 determines the overall Boolean result. Any logical expressions to the right of this expression are not evaluated. This allows you to avoid unnecessary time-consuming expression evaluations.

Some commands allow you to specify a comma-separated list as an argument value. In this case, Caché handles each listed argument like an independent command statement. Therefore, **IF x=7,y=4,z=2** is parsed as **IF x=7 THEN IF y=4 THEN IF z=2**, which is functionally identical to the short-circuit logical operators statement **IF (x=7)&&(y=4)&&(z=2)**.

In the following example, the IF test uses a regular logical operator (&). Therefore, all functions are executed even though the first function returns 0 (FALSE) which automatically makes the result of the entire expression FALSE:

```
LogExp
 IF $$One() & $$Two() {
   WRITE !,"Expression is TRUE."  }
 ELSE {
   WRITE !,"Expression is FALSE." }
One()
 WRITE !,"one"
 QUIT 0
Two()
 WRITE !,"two"
 QUIT 1
```

In the following example, the IF test uses a short-circuit logical operator (&&). Therefore, the first function is executed and returns 0 (FALSE) which automatically makes the result of the entire expression FALSE. The second function is not executed:

```
LogExp
 IF $$One() && $$Two() {
   WRITE !,"Expression is TRUE."  }
 ELSE {
   WRITE !,"Expression is FALSE." }
One()
 WRITE !,"one"
 QUIT 0
Two()
 WRITE !,"two"
 QUIT 1
```


In the following example, the IF test specifies comma-separated arguments. The comma is not a logical operator, but has the same effect as specifying the short-circuit && logical operator. The first function is executed and returns 0 (FALSE) which automatically makes the result of the entire expression FALSE. The second function is not executed:

```
LogExp
 IF $$One(),$$Two() {
   WRITE !,"Expression is TRUE."  }
 ELSE {
   WRITE !,"Expression is FALSE." }
One()
 WRITE !,"one"
 QUIT 0
Two()
 WRITE !,"two"
 QUIT 1
```

Assignment

Within ObjectScript the SET command is used along with the assignment operator (=) to assign a value to a variable. The right-hand side of an assignment command is an expression:

```
SET value = 0
SET value = a + b
```

Within ObjectScript it is also possible to use certain functions on the left-hand side of an assignment command:

```
SET pies = "apple,banana,cherry"
WRITE "Before: ",pies,!

// set the 3rd comma-delimited piece of pies to coconut
SET $Piece(pies,",",3) = "coconut"
WRITE "After: ",pies
```

String-to-Number Conversion

A string can be numeric, partially numeric, or non-numeric.

- A numeric string consists entirely of numeric characters. For example, **"123"**, **"+123"**, **".123"**, **"++0007"**, **"-0"**.
- A partially numeric string is a string that begins with numeric symbols, followed by non-numeric characters. For example, **"3 blind mice"**, **"-12 degrees"**.
- A non-numeric string begins with a non-numeric character. For example, **" 123"**, **"the 3 blind mice"**, **"three blind mice"**.

Numeric Strings

When a numeric string or partially numeric string is used in an arithmetic expression, it is interpreted as a number. This numeric value is obtained by scanning the string from left to right to find the longest sequence of leading characters that can be interpreted as a numeric literal. The following characters are permitted:

- The digits 0 through 9.

- The PlusSign and MinusSign property values. By default these are the "**+**" and "**-**" characters, but are locale-dependent. Use the %SYS.NLS.Format.GetFormatItem() method to return the current settings.

- The DecimalSeparator property value. By default this is the "**.**" character, but is locale-dependent. Use the %SYS.NLS.Format.GetFormatItem() method to return the current setting.

- The letters "**e**", and "**E**" may be included as part of a numeric string when in a sequence representing scientific notation, such as 4E3.

Note that the NumericGroupSeparator property value (the "**,**" character, by default) is not considered a numeric character. Therefore, the string **"123,456"** is a partially numeric string that resolves to the number **"123"**.

Numeric strings and partial numeric strings are converted to canonical form prior to arithmetic operations (such as addition and subtraction) and greater than/less than comparison operations (<, >, <=, >=). Numeric strings are not converted to canonical form prior to equality comparisons (=, '=), because these operators are also used for string comparisons.

The following example shows arithmetic comparisons of numeric strings:

```
WRITE "3" + 4,!          // returns 7
WRITE "003.0" + 4,!      // returns 7
WRITE "++--3" + 4,!      // returns 7
WRITE "3 blind mice" + 4,!  // returns 7
```

The following example shows less than (<) comparisons of numeric strings:

```
WRITE "3" < 4,!          // returns 1
WRITE "003.0" < 4,!      // returns 1
WRITE "++--3" < 4,!      // returns 1
WRITE "3 blind mice" < 4,!  // returns 1
```

The following example shows <= comparisons of numeric strings:

```
WRITE "4" <= 4,!          // returns 1
WRITE "004.0" <= 4,!      // returns 1
WRITE "++--4" <= 4,!      // returns 1
WRITE "4 horsemen" <= 4,!    // returns 1
```

The following example shows equality comparisons of numeric strings. Non-canonical numeric strings are compared as character strings, not as numbers. Note that –0 is a non-canonical numeric string, and is therefore compared as a string, not a number:

```
WRITE "4" = 4.00,!        // returns 1
WRITE "004.0" = 4,!       // returns 0
WRITE "++--4" = 4,!       // returns 0
WRITE "4 horsemen" = 4,!    // returns 0
WRITE "-4" = -4,!         // returns 1
```

```
WRITE "0" = 0,!          // returns 1
WRITE "-0" = 0,!         // returns 0
WRITE "-0" = -0,!        // returns 0
```

Non-numeric Strings

If the leading characters of the string are not numeric characters, the string's numeric value is 0 for all arithmetic operations. For <, >, '>, <=, '<, and >= comparisons a non-numeric string is also treated as the number 0. Because the equal sign is used for both the numeric equality operator and the string comparison operator, string comparison takes precedence for = and '= operations. You can append the PlusSign property value (+ by default) to force numeric evaluation of a string. This results in the following logical values, when x and y are different non-numeric strings (for example x="Fred", y="Wilma").

x=y is FALSE	**x=x** is TRUE	**+x=y** is FALSE	**+x=+y** is TRUE	**+x=+x** is TRUE
x'=y is TRUE	**x'=x** is FALSE	**+x'=y** is TRUE	**+x'=+y** is FALSE	**+x'=+x** is FALSE
x<y is FALSE	**x<x** is FALSE	**+x<y** is FALSE	**+x<+y** is FALSE	**+x<+x** is FALSE
x<=y is TRUE	**x<=x** is TRUE	**+x<=y** is TRUE	**+x<=+y** is TRUE	**+x<=+x** is TRUE

Arithmetic Operators

The arithmetic operators interpret their operands as numeric values and produce numeric results. When operating on a string, an arithmetic operators treats the string as its numeric value, according to the rules described in the section "String-to-Number Conversion."

Decimal and $DOUBLE Floating-Point Numbers

Caché supports two representations of decimal-point numbers: ObjectScript decimal floating-point and IEEE double-precision binary floating-point.

- Caché represents a numeric constant as an ObjectScript decimal floating-point value by default. This is referred to as a $DECIMAL number. A $DECIMAL number can exactly represent a fractional value, such as 2.2. You use the $DECIMAL() function to explicitly convert an IEEE double-precision binary floating-point number to the corresponding ObjectScript decimal floating-point number.
- Caché also supports IEEE double-precision binary floating-point values. You use the $DOUBLE() function to explicitly convert a numeric constant to the corresponding IEEE double-precision binary floating-point value (referred to as a $DOUBLE number). A $DOUBLE number can only approximately represent a fractional value, such as 2.2. $DOUBLE representation is usually preferred when doing high-speed scientific calculations because most computers include high-speed hardware for binary floating-point arithmetic.

ObjectScript automatically converts a numeric to the corresponding $DOUBLE value in the following situations:

- If an arithmetic operation involves a $DOUBLE value, ObjectScript converts all numbers in the operation to $DOUBLE. For example, **2.2 + $DOUBLE(.1)** is the same as **$DOUBLE(2.2) + $DOUBLE(.1)**.
- If an operation results in a number that is too large to be represented in ObjectScript decimal floating-point (larger than 9.223372036854775807E145), ObjectScript automatically converts this number to $DOUBLE, rather than issuing a <MAXNUMBER> error.

Unary Positive Operator (+)

The unary positive operator (+) gives its single operand a numeric interpretation. If its operand has a string value, it converts it to a numeric value. It does this by sequentially parsing the characters of the string as a number, until it encounters an invalid character. It then returns whatever leading portion of the string was a well-formed numeric. For example:

```
WRITE + "32 dollars and 64 cents"       // 32
```

If the string has no leading numeric characters, the unary positive operator gives the operand a value of zero. For example:

```
WRITE + "Thirty-two dollars and 64 cents" // 0
```

The unary positive operator has no effect on numeric values. It does not alter the sign of either positive or negative numbers. For example:

```
SET x = -23
WRITE " x: ", x,! // -23
WRITE "+x: ",+x,! // -23
```

Unary Negative Operator (-)

The unary negative operator (-) reverses the sign of a numerically interpreted operand. For example:

```
SET x = -60
WRITE " x: ", x,! // -60
WRITE "-x: ",-x,! // 60
```

If its operand has a string value, the unary negative operator interprets it as a numeric value before reversing its sign. This numeric interpretation is exactly the same as that performed by the unary positive operator, described above. For example:

```
SET x = -23
WRITE -"32 dollars and 64 cents" // -32
```

ObjectScript gives the unary negative operator precedence over the binary arithmetic operators. ObjectScript first scans a numeric expression and performs any unary negative operations. Then, ObjectScript evaluates the expression and produces a result.

In the following example, ObjectScript scans the string and encounters the numeric value of 2 and stops there. It then applies the unary negative operator to the value and uses the Concatenate operator (_) to concatenate the value "Rats" from the second string to the numeric value.

```
WRITE -"2Cats"_"Rats" // -2Rats
```

To return the absolute value of a numeric expression, use the $ZABS function.

Addition Operator (+)

The addition operator produces the sum of two numerically interpreted operands. It uses any leading valid numeric characters as the numeric values of the operands and produces a value that is the sum of the numeric value of the operands.

The following example performs addition on two numeric literals:

```
WRITE 2936.22 + 301.45 //  3237.67
```

The following example performs addition on two locally defined variables:

```
SET x = 4
SET y = 5
WRITE "x + y = ",x + y // 9
```

The following example performs string arithmetic on two operands that have leading digits, adding the resulting numerics:

```
WRITE "4 Motorcycles" + "5 bicycles" // 9
```

The following example illustrates that leading zeros on a numerically evaluated operand do not affect the results the operator produces:

```
WRITE "007" + 10 // 17
```

Subtraction Operator (-)

The subtraction operator produces the difference between two numerically interpreted operands. It interprets any leading, valid numeric characters as the numeric values of the operand and produces a value that is the remainder after subtraction.

The following example performs subtraction on two numeric literals:

```
WRITE 2936.22 - 301.45 // 2634.77
```

The following example performs subtraction on two locally defined variables:

```
SET x = 4
SET y = 5
WRITE "x - y = ",x - y // -1
```

The following example performs string arithmetic on two operands that have leading digits, subtracting the resulting numerics:

```
WRITE "8 apples" - "4 oranges" // 4
```

If the operand has no leading numeric characters, ObjectScript assumes its value to be zero. For example:

```
WRITE "8 apples" - "four oranges" // 8
```

Multiplication Operator (*)

Binary Multiply produces the product of two numerically interpreted operands. It uses any leading numeric characters as the numeric value of the operands and produces a result that is the product.

The following example performs multiplication on two numeric literals:

```
WRITE 9 * 5.5 // 49.5
```

The following example performs multiplication on two locally defined variables:

```
SET x = 4
SET y = 5
WRITE x * y // 20
```

The following example performs string arithmetic on two operands that have leading digits, multiplying the resulting numerics:

```
WRITE "8 apples" * "4 oranges"  // 32
```

If an operand has no leading numeric characters, Binary Multiply assigns it a value of zero.

```
WRITE "8 apples"*"four oranges" // 0
```

Division Operator (/)

Binary Divide produces the result of dividing two numerically interpreted operands. It uses any leading numeric characters as the numeric value of the operands and products a result that is the quotient.

The following example performs division on two numeric literals:

```
WRITE 9 / 5.5 // 1.636363636363636364
```

The following example performs division on two locally defined variables:

```
SET x = 4
SET y = 5
WRITE x / y // .8
```

The following example performs string arithmetic on two operands that have leading digits, dividing the resulting numerics:

```
WRITE "8 apples" / "4 oranges"  // 2
```

If the operand has no leading numeric characters, Binary Divide assumes its value to be zero. For example:

```
WRITE "eight apples" / "4 oranges" // 0
// "8 apples"/"four oranges" generates a <DIVIDE> error
```

mixed operands and type conversion

When the variables and constants of different types are mixed in an expression ,they are all converted to the same type .
The compiler converts all operands up to the type of the largest operand , which is called type promotion .

Generally , the lower type of the operator is promoted to higher type operand and the result will be of the higher type .

The ordering the data type is :
char < int < long < float <double .
Also , an unsigned value out ranks the corresponding signed type .

Conversion Rules :

The following are the rules applicable to the arithmetic operations between two operators having different data types .
·   If char and short int values are used as operands , the char operand is automatically elevated to int .
·   If float and double values are used as operands , the float operand is automatically elevated to double .
·   If int and float values are used as operands , the int operand is automatically elevated to float .
·   If float and double values are used as operands ,the float is automatically elevated to double .
·   If long and unsigned int are used as operands , both the operands are automatically elevated to unsigned long.

Type Casting :

To convert the value of an expression to a different type ,the expression must be preceded by the name of the desired data type , enclosed in parenthesis ; (data type) expression .

This type of conversion is known as type casting . The operators with in C are grouped hierarchically according to their precedence (i.e, order of evaluation).

Operations with higher precedence are carried out before operations having a lower precedence . The natural order of evaluation can be altered through the use of paranthesis.


The arithmetic operators * , / and % fall in to precedence group , + and – falls in to another . The first group has a higher precedence than the second one .

The order in which consecutive operations with in the same precedence group are carried out is known as associately .with in each of the precedence groups described above , the associativity is from left to right.

Example Program:

```c
#include<stdio.h>
int main()
{
    int x,y;
    float n;
    printf("Enter x and y value : ");
    scanf("%d%d",&x,&y);
    n=(float)x/y;
    printf("n = %f",n);
}
```

Output :
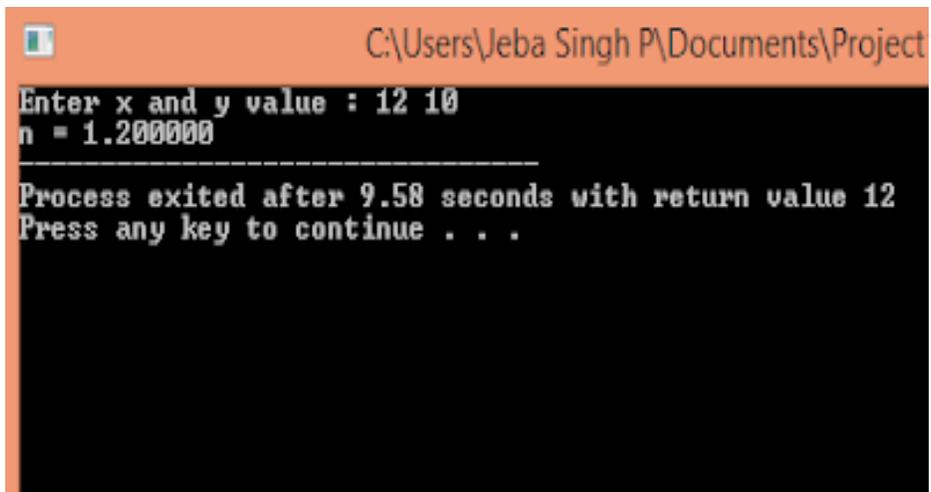
```
C:\Users\Jeba Singh P\Documents\Project
Enter x and y value : 12 10
n = 1.200000
-------------------------------------
Process exited after 9.58 seconds with return value 12
Press any key to continue . . .
```

Logical operators

Logical operators are mainly used to control program flow. Usually, you will find them as part of an if, a while, or some other control statement (Chapter 6)

The Logical operators are:

op1 && op2
    -- Performs a logical AND of the two operands.
op1 || op2
    -- Performs a logical OR of the two operands.
!op1
    -- Performs a logical NOT of the operand.

The concept of logical operators is simple. They allow a program to make a decision based on multiple conditions. Each operand is considered a condition that can be evaluated to a true or false value. Then the value of the conditions is used to determine the overall value of the op1 operator op2 or !op1 grouping. The following examples demonstrate different ways that logical conditions can be used.

The && operator is used to determine whether both operands or conditions are true and.pl.

For example:

```
if ($firstVar == 10 && $secondVar == 9) {

    print("Error!");

};
```

If either of the two conditions is false or incorrect, then the print command is bypassed.

The || operator is used to determine whether either of the conditions is true.

For example:

```
if ($firstVar == 9 || $firstVar == 10) {

    print("Error!");
```

If either of the two conditions is true, then the print command is run.

Caution If the first operand of the || operator evaluates to true, the second operand will not be evaluated. This could be a source of bugs if you are not careful.

For instance, in the following code fragment:

if ($firstVar++ || $secondVar++) { print("\n"); }

variable $secondVar will not be incremented if $firstVar++ evaluates to true.

The ! operator is used to convert true values to false and false values to true. In other words, it inverts a value. Perl considers any non-zero value to be true-even string values. For example:

$firstVar = 10;

$secondVar = !$firstVar;

if ($secondVar == 0) {

   print("zero\n");

};

is equal to 0- and the program produces the following output:

zero

You could replace the 10 in the first line with "ten," 'ten,' or any non-zero, non-null value.

Bit operations

The following table lists the Bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e.,

		0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Example

Try the following example to understand all the bitwise operators available in C −

```c
#include <stdio.h>

main() {

   unsigned int a = 60;      /* 60 = 0011 1100 */
   unsigned int b = 13;      /* 13 = 0000 1101 */
   int c = 0;

   c = a & b;       /* 12 = 0000 1100 */
   printf("Line 1 - Value of c is %d\n", c );

   c = a | b;       /* 61 = 0011 1101 */
```

```
   printf("Line 2 - Value of c is %d\n", c );

   c = a ^ b;      /* 49 = 0011 0001 */
   printf("Line 3 - Value of c is %d\n", c );

   c = ~a;        /*-61 = 1100 0011 */
   printf("Line 4 - Value of c is %d\n", c );

   c = a << 2;    /* 240 = 1111 0000 */
   printf("Line 5 - Value of c is %d\n", c );

   c = a >> 2;    /* 15 = 0000 1111 */
   printf("Line 6 - Value of c is %d\n", c );
}
```

When you compile and execute the above program, it produces the following result −

Line 1 - Value of c is 12

Line 2 - Value of c is 61

Line 3 - Value of c is 49

Line 4 - Value of c is -61

Line 5 - Value of c is 240

Line 6 - Value of c is 15

Operator precedence and associativity

Operator precedence: It dictates the order of evaluation of operators in an expression.

Associativity: It defines the order in which operators of the same precedence are evaluated in an expression. Associativity can be either from left to right or right to left.

Consider the following example:

24 + 5 * 4

Here we have two operators + and *, Which operation do you think will be evaluated first, addition or multiplication? If the addition is applied first then answer will be 116 and if the multiplication is applied first answer will be 44. To answer such question we need to consult the operator precedence table.

In C, each operator has a fixed priority or precedence in relation to other operators. As a result, the operator with higher precedence is evaluated before the operator with lower precedence. Operators that appear in the same group have the same precedence. The following table lists operator precedence and associativity.

Operators in the top have higher precedence and it decreases as we move towards the bottom.

From the precedence table, we can conclude that the * operator is above the + operator, so the * operator has higher precedence than the + operator, therefore in the expression 24 + 5 * 4, subexpression 5 * 4 will be evaluated first.

24 + 5 * 4   ①

20

24 + 20   ②
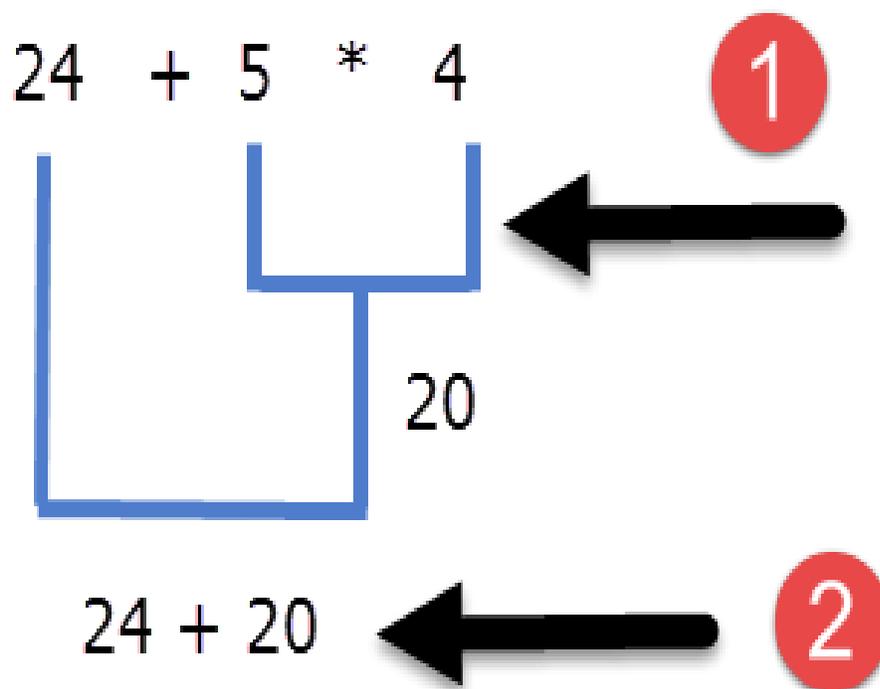
Ans : 44

Here are some more examples:

Example 1:

34 + 12/4 - 45
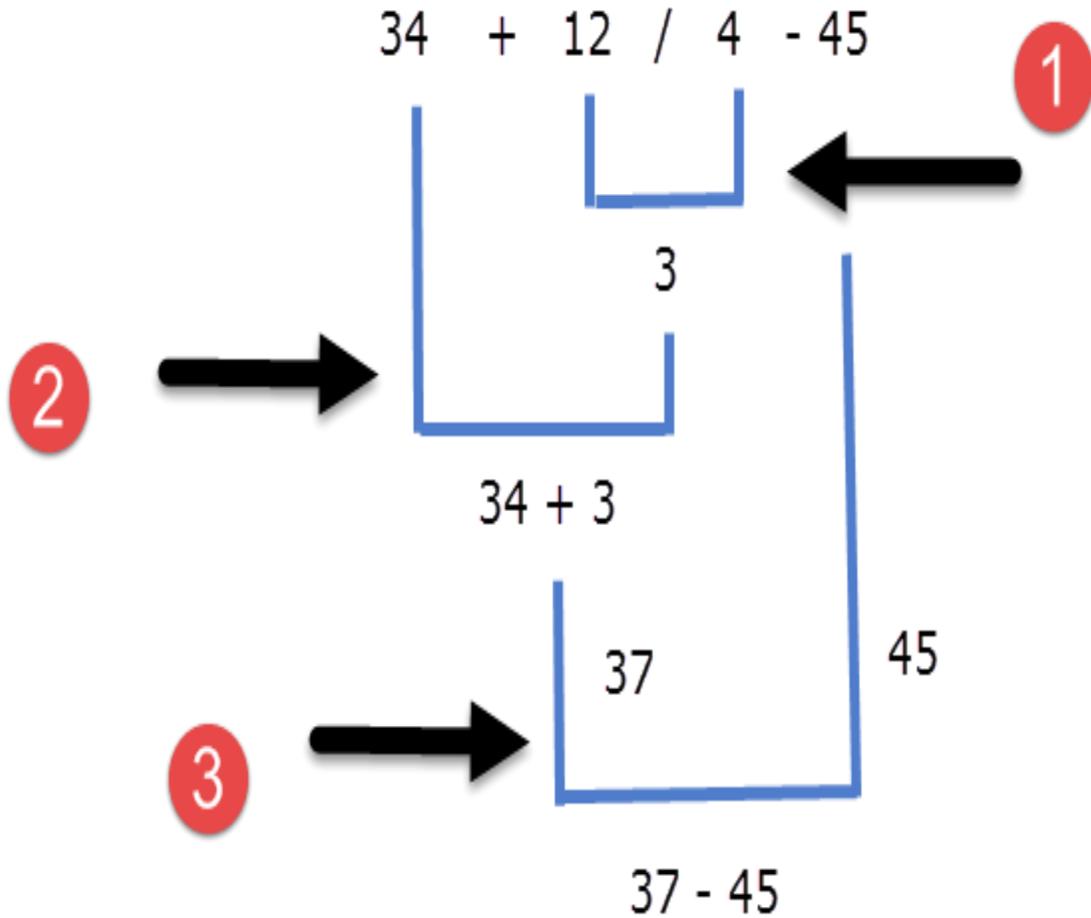
Ans : -8

Here the / operator has higher precedence hence 12/4 is evaluated first. The operators + and - have the same precedence because they are in the same group. So which one of them will be evaluated first? To solve this problem you need to consult the

associativity of the operator. As you can see in the table, the operators + and - have the same precedence and associates from left to right therefore in our expression 34 + 12/4 - 45 after division, addition (+) will be performed before subtraction (-).

Using Parentheses

If you look at the precedence table you will find that the precedence of parentheses (()) operator is the highest. Consequently, just as we did in school, we can use parentheses to change the sequence of operations. Consider the following example:

3 + 4 * 2

Here, the * operator will be evaluated first followed by the + operator.

What if you want the addition to take place first followed by multiplication?

We can do this using parentheses, as follows:

(3 + 4) * 2

Whatever you have wrapped inside the parentheses will be evaluated first. As a result, in this expression the addition will take place first followed by multiplication.

You can also nest parentheses like this:

(2 + (3 + 2)) * 10

In such cases expression inside the innermost parentheses is evaluated first, then the next innermost parentheses and so on.

We can also use parentheses to make complex expressions a little more readable. For example:

```
1 age < 18 && height < 48 || age > 60 && height > 72
2 (age < 18 && height < 48) || (age > 60 && height > 72) // much better than the above
```
Both expressions give the same result, but adding parentheses makes our intent much clear.

We haven't yet discussed relational and logical operators. So the above expression might not make perfect sense. Relational and Logical operators are discussed in detail in Relational Operators in C and Logical Operators in C, respectively. In the next chapter, we will learn about the if else statement in C.

UNIT III

Conditional Program Execution:

Applying if and switch statements

The switch case statement is used when we have multiple options and we need to perform a different task for each option.

C – Switch Case Statement

Before we see how a switch case statement works in a C program, let's checkout the syntax of it.

```
switch (variable or an integer expression)
{
    case constant:
    //C Statements
    ;
    case constant:
    //C Statements
    ;
    default:
    //C Statements
    ;
}
```

Flow Diagram of Switch Case
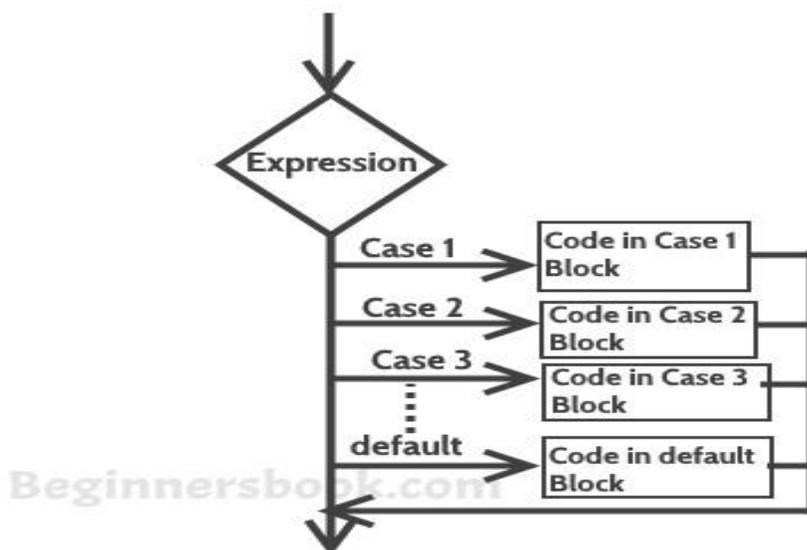
Example of Switch Case in C

Let's take a simple example to understand the working of a switch case statement in C program.

```c
#include <stdio.h>
int main()
{
   int num=2;
   switch(num+2)
   {
      case 1:
        printf("Case1: Value is: %d", num);
      case 2:
        printf("Case1: Value is: %d", num);
      case 3:
        printf("Case1: Value is: %d", num);
      default:
        printf("Default: Value is: %d", num);
   }
   return 0;
}
```

Output:

Default: value is: 2

Explanation:

In switch I gave an expression, you can give variable also. I gave num+2, where num value is 2 and after addition the expression resulted 4. Since there is no case defined with value 4 the default case is executed.

Twist in a story – Introducing Break statement

Before we discuss more about break statement, guess the output of this C program.

```c
#include <stdio.h>
int main()
{
   int i=2;
   switch (i)
```

```
    {
       case 1:
          printf("Case1 ");
       case 2:
          printf("Case2 ");
       case 3:
          printf("Case3 ");
       case 4:
          printf("Case4 ");
       default:
          printf("Default ");
    }
    return 0;
}
```

Output:

```
Case2 Case3 Case4 Default
```
I passed a variable to switch, the value of the variable is 2 so the control jumped to the case 2, However there are no such statements in the above program which could break the flow after the execution of case 2. That's the reason after case 2, all the subsequent cases and default statements got executed.

How to avoid this situation?

We can use break statement to break the flow of control after every case block.

Break statement in Switch Case

Break statements are useful when you want your program-flow to come out of the switch body. Whenever a break statement is encountered in the switch body, the control comes out of the switch case statement.

Example of Switch Case with break

I'm taking the same above that we have seen above but this time we are using break.

```
#include <stdio.h>
int main()
{
    int i=2;
    switch (i)
    {
```

```c
        case 1:
            printf("Case1 ");
            break;
        case 2:
            printf("Case2 ");
            break;
        case 3:
            printf("Case3 ");
            break;
        case 4:
            printf("Case4 ");
            break;
        default:
            printf("Default ");
    }
    return 0;
}
```

Output:

Case 2

Why didn't I use break statement after default?

The control would itself come out of the switch after default so I didn't use it, however if you want to use the break after default then you can use it, there is no harm in doing that.

Few Important points regarding Switch Case

1) Case doesn't always need to have order 1, 2, 3 and so on. They can have any integer value after case keyword. Also, case doesn't need to be in an ascending order always, you can specify them in any order as per the need of the program.

2) You can also use characters in switch case. for example –

```c
#include <stdio.h>
int main()
{
    char ch='b';
    switch (ch)
    {
        case 'd':
            printf("CaseD ");
            break;
```

```
        case 'b':
          printf("CaseB");
          break;
        case 'c':
          printf("CaseC");
          break;
        case 'z':
          printf("CaseZ ");
          break;
        default:
          printf("Default ");
    }
    return 0;
}
```

Output:

```
CaseB
```

3) The expression provided in the switch should result in a constant value otherwise it would not be valid.

For example:

Valid expressions for switch –

```
switch(1+2+23)
switch(1*2+3%4)
```

Invalid switch expressions –

```
switch(ab+cd)
switch(a+b+c)
```

4) Nesting of switch statements are allowed, which means you can have switch statements inside another switch. However nested switch statements should be avoided as it makes program more complex and less readable.
5) Duplicate case values are not allowed. For example, the following program is incorrect:
This program is **wrong** because we have two case 'A' here which is wrong as we cannot have duplicate case values.

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

```c
    char ch='B';
    switch (ch)
    {
       case 'A':
         printf("CaseA");
         break;
       case 'A':
         printf("CaseA");
         break;
       case 'B':
         printf("CaseB");
         break;
       case 'C':
         printf("CaseC ");
         break;
       default:
         printf("Default ");
    }
    return 0;
}
```

6) The default statement is optional, if you don't have a default in the program, it would run just fine without any issues. However it is a good practice to have a default statement so that the default executes if no case is matched. This is especially useful when we are taking input from user for the case choices, since user can sometime enter wrong value, we can remind the user with a proper error message that we can set in the default statement.

nesting if and else

Conditional Statements in C programming are used to make decisions based on the conditions. Conditional statements execute sequentially when there is no condition around the statements. If you put some condition for a block of statements, the execution flow may change based on the result evaluated by the condition. This process is called decision making in 'C.'

In 'C' programming conditional statements are possible with the help of the following two constructs:

1. If statement

2. If-else statement

It is also called as branching as a program decides which statement to execute based on the result of the evaluated condition.

In this tutorial, you will learn-

- What is a Conditional Statement?
- If statement
- Relational Operators
- The If-Else statement
- Conditional Expressions
- Nested If-else Statements
- Nested Else-if statements

If statement

It is one of the powerful conditional statement. If statement is responsible for modifying the flow of execution of a program. If statement is always used with a condition. The condition is evaluated first before executing any statement inside the body of If. The syntax for if statement is as follows:

```
if (condition)
    instruction;
```

The condition evaluates to either true or false. True is always a non-zero value, and false is a value that contains zero. Instructions can be a single instruction or a code block enclosed by curly braces { }.

Following program illustrates the use of if construct in 'C' programming:

```
#include<stdio.h>
int main()
{
        int num1=1;
        int num2=2;
        if(num1<num2)              //test-condition
        {
                printf("num1 is smaller than num2");
        }
        return 0;
}
```

Output:

num1 is smaller than num2

The above program illustrates the use of if construct to check equality of two numbers.

```c
#include<stdio.h>
int main()
{
    int num1=1;
    int num2=2;
    if(num1<num2)     //test-condition
    {
        printf("num1 is smaller than num2");
    }

    return 0;

}
```

1. In the above program, we have initialized two variables with num1, num2 with value as 1, 2 respectively.
2. Then, we have used if with a test-expression to check which number is the smallest and which number is the largest. We have used a relational expression in if construct. Since the value of num1 is smaller than num2, the condition will evaluate to true.
3. Thus it will print the statement inside the block of If. After that, the control will go outside of the block and program will be terminated with a successful result.

Relational Operators

C has six relational operators that can be used to formulate a Boolean expression for making a decision and testing conditions, which returns true or false :

< less than

<= less than or equal to

> greater than

>= greater than or equal to

== equal to

!= not equal to

Notice that the equal test (==) is different from the assignment operator (=) because it is one of the most common problems that a programmer faces by mixing them up.

For example:

```
int x = 41;
x =x+ 1;
if (x == 42) {
   printf("You succeed!");}
```

Output :

You succeed

Keep in mind that a condition that evaluates to a non-zero value is considered as true.

For example:

```
int present = 1;
if (present)
  printf("There is someone present in the classroom \n");
```

Output :

There is someone present in the classroom

The If-Else statement
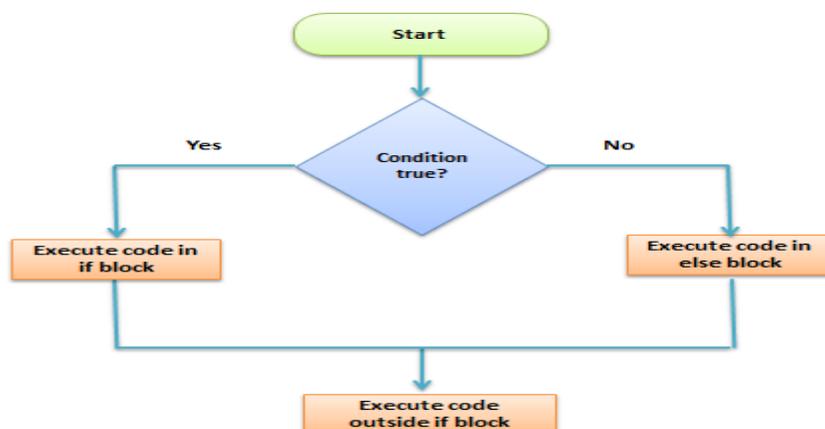
The if-else is statement is an extended version of If. The general form of if-else is as follows:

```
if (test-expression)
{
    True block of statements
}
Else
{
    False block of statements
}
Statements;
```

n this type of a construct, if the value of test-expression is true, then the true block of statements will be executed. If the value of test-expression if false, then the false block of statements will be executed. In any case, after the execution, the control will be automatically transferred to the statements appearing outside the block of If.

Following programs illustrate the use of the if-else construct:

We will initialize a variable with some value and write a program to determine if the value is less than ten or greater than ten.

Let's start.

```
#include<stdio.h>
int main()
{
        int num=19;
        if(num<10)
        {
                printf("The value is less than 10");
        }
        else
        {
                printf("The value is greater than 10");
        }
        return 0;
}
```

Output:

The value is greater than 10

```c
#include<stdio.h>
int main()
{
    int num=19;       ①
    if(num<10)②
    {
     ③printf("The value is less than 10");
    }
    else
    {
     ④printf("The value is greater than 10");
    }
    return 0;
}
```

1. We have initialized a variable with value 19. We have to find out whether the number is bigger or smaller than 10 using a 'C' program. To do this, we have used the if-else construct.
2. Here we have provided a condition num<10 because we have to compare our value with 10.
3. As you can see the first block is always a true block which means, if the value of test-expression is true then the first block which is If, will be executed.
4. The second block is an else block. This block contains the statements which will be executed if the value of the test-expression becomes false. In our program, the value of num is greater than ten hence the test-condition becomes false and else block is executed. Thus, our output will be from an else block which is "The value is greater than 10". After the if-else, the program will terminate with a successful result.

In 'C' programming we can use multiple if-else constructs within each other which are referred to as nesting of if-else statements.

Conditional Expressions

There is another way to express an if-else statement is by introducing the **?:** operator. In a conditional expression the **?:** operator has only one statement associated with the if and the else.

For example:

```
#include <stdio.h>
int main() {
  int y;
  int x = 2;
  y = (x >= 6) ?  6 : x;/* This is equivalent to:  if (x >= 5)   y = 5;  else   y = x; */
  printf("y =%d ",y);
  return 0;}
```

Output :

```
y =2
```

Nested If-else Statements

When a series of decision is required, nested if-else is used. Nesting means using one if-else construct within another one.

Let's write a program to illustrate the use of nested if-else.

```
#include<stdio.h>
int main()
{
        int num=1;
        if(num<10)
        {
                if(num==1)
                {
                        printf("The value is:%d\n",num);
                }
                else
                {
                        printf("The value is greater than 1");
                }
        }
        else
        {
                printf("The value is greater than 10");
        }
```

```
        return 0;
}
```

Output:

The value is:1

The above program checks if a number is less or greater than 10 and prints the result using nested if-else construct.

```c
#include<stdio.h>
int main()
{
    int num=1;
    if(num<10){
        if(num==1){
            printf("The value is:%d\n",num);
        }
        else{
            printf("The value is greater than 1");
        }
    }
    else{
        printf("The value is greater than 10");
    }
    return 0;
}
```

1. Firstly, we have declared a variable num with value as 1. Then we have used if-else construct.
2. In the outer if-else, the condition provided checks if a number is less than 10. If the condition is true then and only then it will execute the inner loop. In this case, the condition is true hence the inner block is processed.

3. In the inner block, we again have a condition that checks if our variable contains the value 1 or not. When a condition is true, then it will process the If block otherwise it will process an else block. In this case, the condition is true hence the If a block is executed and the value is printed on the output screen.
4. The above program will print the value of a variable and exit with success.

Try changing the value of variable see how the program behaves.

NOTE: In nested if-else, we have to be careful with the indentation because multiple if-else constructs are involved in this process, so it becomes difficult to figure out individual constructs. Proper indentation makes it easy to read the program.

Nested Else-if statements

Nested else-if is used when multipath decisions are required.

The general syntax of how else-if ladders are constructed in 'C' programming is as follows:

```
 if (test - expression 1) {
    statement1;
} else if (test - expression 2) {
    Statement2;
} else if (test - expression 3) {
    Statement3;
} else if (test - expression n) {
    Statement n;
} else {
    default;
}
Statement x;
```

This type of structure is known as the else-if ladder. This chain generally looks like a ladder hence it is also called as an else-if ladder. The test-expressions are evaluated from top to bottom. Whenever a true test-expression if found, statement associated with it is executed. When all the n test-expressions becomes false, then the default else statement is executed.

Let us see the actual working with the help of a program.

```
#include<stdio.h>
int main()
{
        int marks=83;
        if(marks>75){
```

```
                printf("First class");
        }
        else if(marks>65){
                printf("Second class");
        }
        else if(marks>55){
                printf("Third class");
        }
        else{
                printf("Fourth class");
        }
        return 0;
}
```

Output:

First class

The above program prints the grade as per the marks scored in a test. We have used the else-if ladder construct in the above program.

1. We have initialized a variable with marks. In the else-if ladder structure, we have provided various conditions.
2. The value from the variable marks will be compared with the first condition since it is true the statement associated with it will be printed on the output screen.
3. If the first test condition turns out false, then it is compared with the second condition.
4. This process will go on until the all expression is evaluated otherwise control will go out of the else-if ladder, and default statement will be printed.

Try modifying the value and notice the change in the output.

Summary

- Decision making or branching statements are used to select one path based on the result of the evaluated expression.

- It is also called as control statements because it controls the flow of execution of a program.

- 'C' provides if, if-else constructs for decision-making statements.

- We can also nest if-else within one another when multiple paths have to be tested.

- The else-if ladder is used when we have to check various ways based upon the result of the expression.

restrictions on switch values

In this tutorial, we will discuss what are Java switch statements, how to use them, the allowed data types that can be used in the switch statements and the advantages and restrictions that apply to them.

Contents hide

What is a "switch" statement

A "switch" statement in Java is a conditional operator used to direct the execution of an algorithm to a specific code path. Inside a switch statement, multiple execution paths are defined. The path that is taken will depend upon an input value provided to the switch statement.

An analogy can be drawn from the good old days with a telephone switch board. A person calls the switch board staff and asks to be routed to a specific number, and the operator on the switch board will connect your call to a specific routing port, depending on the number you as for.

How to use a switch statement

The following is an example of a switch statement. The switch statement is fed the character value "grade", and one of the paths will be executed depending on the value of grade.

```java
package com.nullbeans;

public class Main {

    public static void main(String[] args) {


        char grade = 'B';

        switch (grade){
            case 'Á': System.out.println("Excellent!"); break;
            case 'B': System.out.println("Very good!"); break;
            case 'C': System.out.println("Average!"); break;
            case 'D': System.out.println("Poor!"); break;
            case 'E': System.out.println("Very poor!"); break;
            case 'F': System.out.println("Failed"); break;
        }

    }
}
```

When running this program, we will get the following output.

Very good!

Process finished with exit code

Let us go over some keywords in that example:

switch: This signifies the start of a switch statement. The switch statement takes a single input value to be evaluated.

case: The "case" keyword" signifies an entry point for execution. Each case is defined for a specific value. The entry point can be entered if the input value to the switch statement matches the one in the case. As you can see in our example, since we used the value "B", we started the execution from the "case B" part of the switch statement.

break: The "break" keyword signals the program to exit the switch statement. It is not mandatory to have the break keyword in each case. However, if it is missing, the program will not exit the switch statement and will continue executing until either a break statement is found, or we exit the program. Let us discuss this in more details in the next section.

How does the "break" keyword work

The break keyword is a java keyword that allows the execution to exit from its current conditional scope. For example, if it is inside a "while" loop, the program will skip the execution of the loop and will exit the loop's scope. Similarly, if the break keyword is encountered inside a switch statement, it will prompt Java to exit the while switch statement.

Take for example the following diagram. Since the code of every case is followed by a "break", only that code will be executed.

This diagram illustrates a switch statement with a break after each case. Since there is a break after each case, only the code related to the single matching case is executed. (Click to enlarge)

In our previous program, only one statement was executed, since after each case, a break keyword was present. So what if we did not have a break keyword? Then the program will continue execution until either a break statement is encountered or the end of the switch statement is reached.

This diagram illustrates a switch statement where case 1 and case 3 do not have the break keywords. Using this setup, if X is equal to 1, then the code from case 1 and 2 is executed. The program stops at case 2 because there is a break keyword there. If X is equal to 2, then only the code from case 2 is executed. If X is equal to 3, then the code from case 3 is executed and the switch statement is terminated. (Click to enlarge)

Let us discuss the next example. The following program is used to print out the features a customer would get when they choose a hosting plan from our imaginary web hosting provider. We can describe the plans as follows:

- Our plans start with a "Basic" package with 2 free domain names.
- Then there is a "Basic plus" plan, which includes whatever is in the "Basic" plan plus unlimited bandwidth.
- Then comes the "Premium" plan which includes whatever is in "Basic plus" and "Basic", plus an extra 1GB of RAM.
- And finally, there comes the "Premium plus" plan. The plan includes whatever is in the "Premium", the "Basic plus" and the "Basic", plus an extra 1 CPU.

Now, let us model this program in Java using a switch statement. Since we can ristrict the entry point using the "case" keyword, we can write our program in a way that includes all the correct features by simply defining the switch case order from the most inclusive case to the least inclusive.

```
String hostingPlan = "Premium";

System.out.println("Your hosting plan includes:");

switch(hostingPlan){

    case "Premium plus": System.out.println("- Extra 1 CPU");
    case "Premium": System.out.println("- Extra 1GB RAM");
    case "Basic plus": System.out.println("- Unlimited bandwidth");
    case "Basic": System.out.println("- 2 Domain names");
}
```

Let us run our program and check the results.

```
Your hosting plan includes:
- Extra 1GB RAM
- Unlimited bandwidth
- 2 Domain names

Process finished with exit code 0
```

Notice here that the println statement of the "Premium plus" case was not executed as the program did not enter the switch statement in this case. The program entered from the "Premium" case. Notice also that, even though the input value "Premium" did not match the "Basic plus" and "Basic" cases, the statements from these cases were

executed anyway. **This is because the break statement was missing. Therefore, nothing stopped the execution of the rest of the switch statement cases.**

The "default" case

What if the input to the switch statement did not match any of the cases? Then non of the statements would be executed. What if we wanted to print out an error, to tell the user that they inserted an unrecognized value? The default case comes to the rescue.

The default case is a a switch statement case which is executed if non of the other switch cases matches the switch statement input. Let us modify our previous example to include a default case.

```java
String hostingPlan = "Super premium";

System.out.println("Your hosting plan includes:");

switch(hostingPlan){

    case "Premium plus": System.out.println("- Extra 1 CPU");
    case "Premium": System.out.println("- Extra 1GB RAM");
    case "Basic plus": System.out.println("- Unlimited bandwidth");
    case "Basic": System.out.println("- 2 Domain names"); break;
    default: System.out.println("Hosting plan not recognized");

}
```

Notice that we added the "break" keyword after the "Basic" case in order to stop the execution of the switch statement if one of the cases match. Now let us run the program.

```
Your hosting plan includes:
Hosting plan not recognized

Process finished with exit code 0
```

Since non of the switch cases matches the input "Super premium", the default case was executed, Please note that the default case does not need to be the last case in the switch statement. It can be added at the beginning or in the middle of the cases. It will behave just like any other entry point. But in the "default" case, it is entered only when no other case is matched.

Advantages of using a switch statement over a traditional if-else cascade

While it is true that the same functionalities of the switch statement can also be implemented using an if-else cascade, the switch statement allows us in certain situations to write a more clear, compact and readable code.

For example, if we wanted to convert our previous example to an if-else cascade, it would look as follows.

```java
String hostingPlan = "Premium";

System.out.println("Your hosting plan includes:");

if(hostingPlan.equals("Premium plus")){
    System.out.println("- Extra 1 CPU");
    System.out.println("- Extra 1GB RAM");
    System.out.println("- Unlimited bandwidth");
    System.out.println("- 2 Domain names");
}else if(hostingPlan.equals("Premium")){
    System.out.println("- Extra 1GB RAM");
    System.out.println("- Unlimited bandwidth");
    System.out.println("- 2 Domain names");
}else if(hostingPlan.equals("Basic plus")){
    System.out.println("- Unlimited bandwidth");
    System.out.println("- 2 Domain names");
}else if(hostingPlan.equals("Basic")){
    System.out.println("- 2 Domain names");
}else {
    System.out.println("Hosting plan not recognized");
}

}
```

Notice that we have to perform a lot of code repetition. This is not ideal as if we need to change something, we would have to change it in other places as well. This is error prone and time consuming. For example, if we decide to provide 3 domain names instead of 2, then we will have to update the if-then part of each if statement.

Restrictions on switch statements

While switch statements can be very efficient and cleaner, they also have certain restrictions. Let us go through these restrictions one by one.

Accepted data types to the switch statement input

The switch statement will accept the following data types as input:

- byte
- short
- int
- long
- char
- String (only Java version 7 and above)
- Byte
- Short
- Integer
- Long
- Enum

Unlike if statements and the conditional operator, floating point numbers and other Objects are not allowed to be used inside a switch statement.

Configuration restrictions to the switch cases

Another restriction that comes to switch statements are the values provided to the case configurations. **Each case should be configured with either a constant value or a final value variable**. Otherwise, the program will not compile.

The following example is allowed in Java.

```
switch (x){

    case 32: //bla bla code here
          //.....

}
```

And this is also allowed

```
final int y = 32;


switch (x){

    case y: //bla bla code here
```

```
    //.....

}
```

However, if we remove the final keyword as in the example below, we will get an error that a "constant expression is required".

```
//program does not compile
int y = 32;


switch (x){

    case y: //bla bla code here
            //.....

}
```

Another restriction is that while switch statement inputs can be an Object such as Integer, Byte, etc (as mentioned in the previous section), **switch cases are more restrictive and allow only primitive data types (int, short, byte, long, char), String and enums.**

Summary

In this post, we discussed how to use a Java switch statement. We discussed what is the break keyword and how to use it. We also discussed the default keyword and its purpose. We also discussed some of the advantages of using a switch statement over a traditional if-then-else cascade. Finally, we discussed what are the restriction that apply to switch statement inputs and case configurations.

use of break and default with switch

In the last topic, we studied different types of loops. We also saw how loops can be nested.

Normally, if we have to choose one case among many choices, if-else is used. But if the number of choices is large, switch..case makes it a bit easier and less complex.

Let's control Case Wise

switch...case is another way to control and decide the execution of statements other than if/else. This is used when we are given a number of choices (cases) and we want to perform a different task for each choice.

Let's first have a look at its syntax.

switch(expression)
{
 case constant1:
 statement(s);
 break;
 case constant2:
 statement(s);
 break;
 /* you can give any number of cases */
 default:
 statement(s);
}

In switch...case, the value of **expression** enclosed within the brackets **( )** following switch is checked. If the value of the expression matches the value of constant in any of the case, the statement(s) corresponding to that case are executed.

If expression does not match any of the constant values, then the statements corresponding to default are executed.

Let's see an example.

```c
#include <stdio.h>

int main()

{

    char grade ;

    printf("Enter your grade\n");

    scanf(" %c" , &grade);

    switch(grade)

    {

        case 'A':

            printf("Excellent!\n");

            break;

        case 'B':

            printf("Outstanding!\n");
```

```
        break;
    case 'C':
        printf("Good!\n");
        break;
    case 'D':
        printf("Can do better\n");
        break;
    case 'E':
        printf("Just passed\n");
        break;
    case 'F':
        printf("You failed\n");
        break;
    default:
        printf("Invalid grade\n");
    }
    return 0;
}
```

break is used to break or terminate a loop whenever we want and is also used with switch.

In this example, the value of 'grade' is 'D'. Since the value of the constants of the first three cases is not 'D', so case 'D' will be executed and 'Can do better' will be printed. Then break statement will terminate the execution without checking the rest of the cases.

If there is no break in any statement, then after execution of the correct case, every case will also be executed. Look at the following code for an example.

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

```c
    char grade ;
    printf("Enter your grade\n");
    scanf(" %c" , &grade);
    switch(grade)
    {
        case'A':
            printf("Excellent!\n");

        case 'B':
            printf("Outstanding!\n");

        case 'C':
            printf("Good!\n");

        case 'D':
            printf("Can do better\n");

        case 'E':
            printf("Just passed\n");

        case 'F':
            printf("You failed\n");

        default:
            printf("Invalid grade\n");
    }
    return 0;
}
```

Output

In the above example, value of grade is **'D'**, so the control jumped to case 'D'. Since there is no break statement after any case, so all the statements after case 'D' also get executed.

As you can see, all the cases after case D have been executed.

If you want to execute only that case whose constant value equals the value of expression of the switch statement, then use the break statement.

Always enclose the character values within ' '.

Now let's see an example with the expression value as an integer.

```c
#include <stdio.h>
int main()
{
    int i = 2;
    switch(i)
    {
        case 1:
            printf("Number is 1\n");
            break;
        case 2:
            printf("Number is 2\n");
            break;
        default:
            printf("Number is greater than 2\n");
    }
    return 0;
}
```

Output

Using break with loops

We can also terminate a loop in the middle of its execution using break. Just type break; after the statement after which you want to break the loop. As simple as that!

Let's consider an example.

```c
#include <stdio.h>
int main()
{
    int a;
    for(a = 1; a <= 10; a ++)
    {
        printf("Hello World\n");
        if(a == 2)
        {
            //loop will now stop
            break;
        }
    }
    return 0;
}
```

In this example, after the first iteration of the loop, a++ increases the value of 'a' to 2 and 'Hello World' got printed. Since the condition of if satisfies this time, break will be executed and the loop will terminate.

Continue

The **continue** statement works similar to break statement. The only difference is that break statement terminates the loop whereas continue statement passes control to the **conditional test** i.e., where the condition is checked, skipping the rest of the statements of the loop.

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

```c
{
    int a ;
    for(a = 1; a <= 10; a ++)
    {
        printf("Hello World\n");
        if (a == 2)
        {

            //this time further statements will not be executed. Control will go to for
            continue;
        }
        printf("a is not 2\n");
    }
    return 0;
}
```

Notice that at the second time, 'a is not 2' is not printed. It means that when 'a' was 2, then 'continue' got executed and control went to for loop without executing further codes.

Program Loops and Iteration:

Uses of while

A **Loop** executes the sequence of statements many times until the stated condition becomes false. A loop consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false. The purpose of the loop is to repeat the same code a number of times.

In this tutorial, you will learn-

- Types of Loops in C
- While Loop in C
- Do-While loop in C

- For loop in C
- Break Statement in C
- Continue Statement in C
- Which loop to Select?

Types of Loops in C

Depending upon the position of a control statement in a program, looping in C is classified into two types:

1. Entry controlled loop

2. Exit controlled loop

In an **entry controlled loop,** a condition is checked before executing the body of a loop. It is also called as a pre-checking loop.

In an **exit controlled loop**, a condition is checked after executing the body of a loop. It is also called as a post-checking loop.
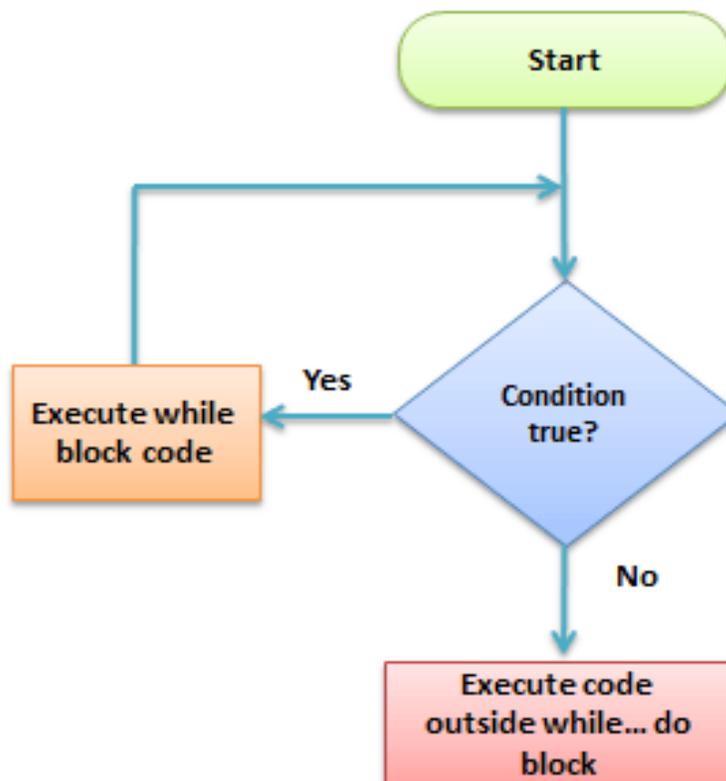
Sample Loop

The control conditions must be well defined and specified otherwise the loop will execute an infinite number of times. The loop that does not stop executing and processes the statements number of times is called as an **infinite loop**. An infinite loop is also called as an "**Endless loop**." Following are some characteristics of an infinite loop:

1. No termination condition is specified.

2. The specified conditions never meet.

The specified condition determines whether to execute the loop body or not.

'C' programming language provides us with three types of loop constructs:

1. The while loop

2. The do-while loop

3. The for loop

While Loop in C

A while loop is the most straightforward looping structure. Syntax of while loop in C programming language is as follows:

```
while (condition) {
        statements;
}
```

It is an entry-controlled loop. In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed. After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false. Once the condition becomes false, the control goes out of the loop.

After exiting the loop, the control goes to the statements which are immediately after the loop. The body of a loop can contain more than one statement. If it contains only one statement, then the curly braces are not compulsory. It is a good practice though to use the curly braces even we have a single statement in the body.

In while loop, if the condition is not true, then the body of a loop will not be executed, not even once. It is different in do while loop which we will see shortly.

Following program illustrates while loop in C programming example:

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

```
#include<conio.h>
int main()
{
        int num=1;          //initializing the variable
        while(num<=10) //while loop with condition
        {
                printf("%d\n",num);
                num++;              //incrementing operation
        }
        return 0;
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

The above program illustrates the use of while loop. In the above program, we have printed series of numbers from 1 to 10 using a while loop.

1. We have initialized a variable called num with value 1. We are going to print from 1 to 10 hence the variable is initialized with value 1. If you want to print from 0, then assign the value 0 during initialization.
2. In a while loop, we have provided a condition (num<=10), which means the loop will execute the body until the value of num becomes 10. After that, the loop will be terminated, and control will fall outside the loop.
3. In the body of a loop, we have a print function to print our number and an increment operation to increment the value per execution of a loop. An initial value of num is 1, after the execution, it will become 2, and during the next execution, it will become 3. This process will continue until the value becomes 10 and then it will print the series on console and terminate the loop.

\n is used for formatting purposes which means the value will be printed on a new line.

Do-While loop in C

A do...while loop in C is similar to the while loop except that the condition is always executed after the body of a loop. It is also called an exit-controlled loop.

Syntax of do...while loop in C programming language is as follows:

```
do {
 statements
} while (expression);
```

As we saw in a while loop, the body is executed if and only if the condition is true. In some cases, we have to execute a body of the loop at least once even if the condition is false. This type of operation can be achieved by using a do-while loop.

In the do-while loop, the body of a loop is always executed at least once. After the body is executed, then it checks the condition. If the condition is true, then it will again execute the body of a loop otherwise control is transferred out of the loop.

Similar to the while loop, once the control goes out of the loop the statements which are immediately after the loop is executed.

The critical difference between the while and do-while loop is that in while loop the while is written at the beginning. In do-while loop, the while condition is written at the end and terminates with a semi-colon (;)

The following loop program in C illustrates the working of a do-while loop:

Below is a do-while loop in C example to print a table of number 2:

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

```c
int main()
{
        int num=1;        //initializing the variable
        do        //do-while loop
        {
                printf("%d\n",2*num);
                num++;              //incrementing operation
        }while(num<=10);
        return 0;
}
```

Output:

```
2
4
6
8
10
12
14
16
18
20
```

In the above example, we have printed multiplication table of 2 using a do-while loop. Let's see how the program was able to print the series.

1. First, we have initialized a variable 'num' with value 1. Then we have written a do-while loop.

2. In a loop, we have a print function that will print the series by multiplying the value of num with 2.

3. After each increment, the value of num will increase by 1, and it will be printed on the screen.

4. Initially, the value of num is 1. In a body of a loop, the print function will be executed in this way: 2*num where num=1, then 2*1=2 hence the value two will be printed. This will go on until the value of num becomes 10. After that loop will be terminated and a statement which is immediately after the loop will be executed. In this case return 0.

For loop in C

A for loop is a more efficient loop structure in 'C' programming. The general structure of for loop syntax in C is as follows:

```
for (initial value; condition; incrementation or decrementation )
{
  statements;
}
```

- The initial value of the for loop is performed only once.
- The condition is a Boolean expression that tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.
- The incrementation/decrementation increases (or decreases) the counter by a set value.

Following program illustrates the for loop in C programming example:

```
#include<stdio.h>
int main()
{
        int number;
        for(number=1;number<=10;number++)      //for loop to print 1-10 numbers
        {
                printf("%d\n",number);          //to print the number
        }
        return 0;
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

The above program prints the number series from 1-10 using for loop.

```c
#include<stdio.h>
#include<conio.h>
int main()
{
    int num=1;        //initializing
    do    //do-while loop
    {
        printf("%d\n", 2*num);
        num++;              //increment
    }while(num<=10);
    return 0;
}
```

1. We have declared a variable of an int data type to store values.

2. In for loop, in the initialization part, we have assigned value 1 to the variable number. In the condition part, we have specified our condition and then the increment part.

3. In the body of a loop, we have a print function to print the numbers on a new line in the console. We have the value one stored in number, after the first iteration the value will be incremented, and it will become 2. Now the variable number has the value 2. The condition will be rechecked and since the condition is true loop will be executed, and it will print two on the screen. This loop will keep on

executing until the value of the variable becomes 10. After that, the loop will be terminated, and a series of 1-10 will be printed on the screen.

In C, the for loop can have multiple expressions separated by commas in each part.

For example:

```
for (x = 0, y = num; x < y; i++, y--) {
  statements;
}
```

Also, we can skip the initial value expression, condition and/or increment by adding a semicolon.

For example:

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

Notice that loops can also be nested where there is an outer loop and an inner loop. For each iteration of the outer loop, the inner loop repeats its entire cycle.

Consider the following example, that uses nested for loop in C programming to output a multiplication table:

```
#include <stdio.h>
int main() {
int i, j;
int table = 2;
int max = 5;
for (i = 1; i <= table; i++) { // outer loop
  for (j = 0; j <= max; j++) { // inner loop
    printf("%d x %d = %d\n", i, j, i*j);
  }
  printf("\n"); /* blank line between tables */
}}
```

Output:

```
1 x 0 = 0
1 x 1 = 1
```

```
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5

2 x 0 = 0
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
```

The nesting of for loops can be done up-to any level. The nested loops should be adequately indented to make code readable. In some versions of 'C,' the nesting is limited up to 15 loops, but some provide more.

The nested loops are mostly used in array applications which we will see in further tutorials.

Break Statement in C

The break statement is used mainly in in the switch statement. It is also useful for immediately stopping a loop.

We consider the following program which introduces a break to exit a while loop:

```
#include <stdio.h>
int main() {
int num = 5;
while (num > 0) {
  if (num == 3)
    break;
  printf("%d\n", num);
  num--;
}}
```

Output:

```
5
4
```

Continue Statement in C

When you want to skip to the next iteration but remain in the loop, you should use the continue statement.

For example:

```
#include <stdio.h>
int main() {
int nb = 7;
while (nb > 0) {
  nb--;
  if (nb == 5)
    continue;
 printf("%d\n", nb);
}}
```

Output:

```
6
4
3
2
1
```

So, the value 5 is skipped.

Which loop to Select?

Selection of a loop is always a tough task for a programmer, to select a loop do the following steps:

- Analyze the problem and check whether it requires a pre-test or a post-test loop.

- If pre-test is required, use a while or for a loop.

- If post-test is required, use a do-while loop.

Summary

- Looping is one of the key concepts on any programming language.

- A block of looping statements in C are executed for number of times until the condition becomes false.

- Loops are of 2 types: entry-controlled and exit-controlled.

- 'C' programming provides us 1) while 2) do-while and 3) for loop.

- For and while loop is entry-controlled loops.

- Do-while is an exit-controlled loop.

do and for loops

In this tutorial, you will learn to create while and do...while loop in C programming with the help of examples.

In programming, loops are used to repeat a block of code until a specified condition is met.

C programming has three types of loops.

1. for loop

2. while loop

3. do...while loop

In the previous tutorial, we learned about for loop. In this tutorial, we will learn about while and do..while loop.

while loop

The syntax of the while loop is:

```
while (testExpression)
{
    // statements inside the body of the loop
}
```

How while loop works?

- The while loop evaluates the test expression inside the parenthesis ().
- If the test expression is true, statements inside the body of while loop are executed. Then, the test expression is evaluated again.
- The process goes on until the test expression is evaluated to false.

- If the test expression is false, the loop terminates (ends).

To learn more about test expression (when the test expression is evaluated to true and false), check out relational and logical operators.

Flowchart of while loop
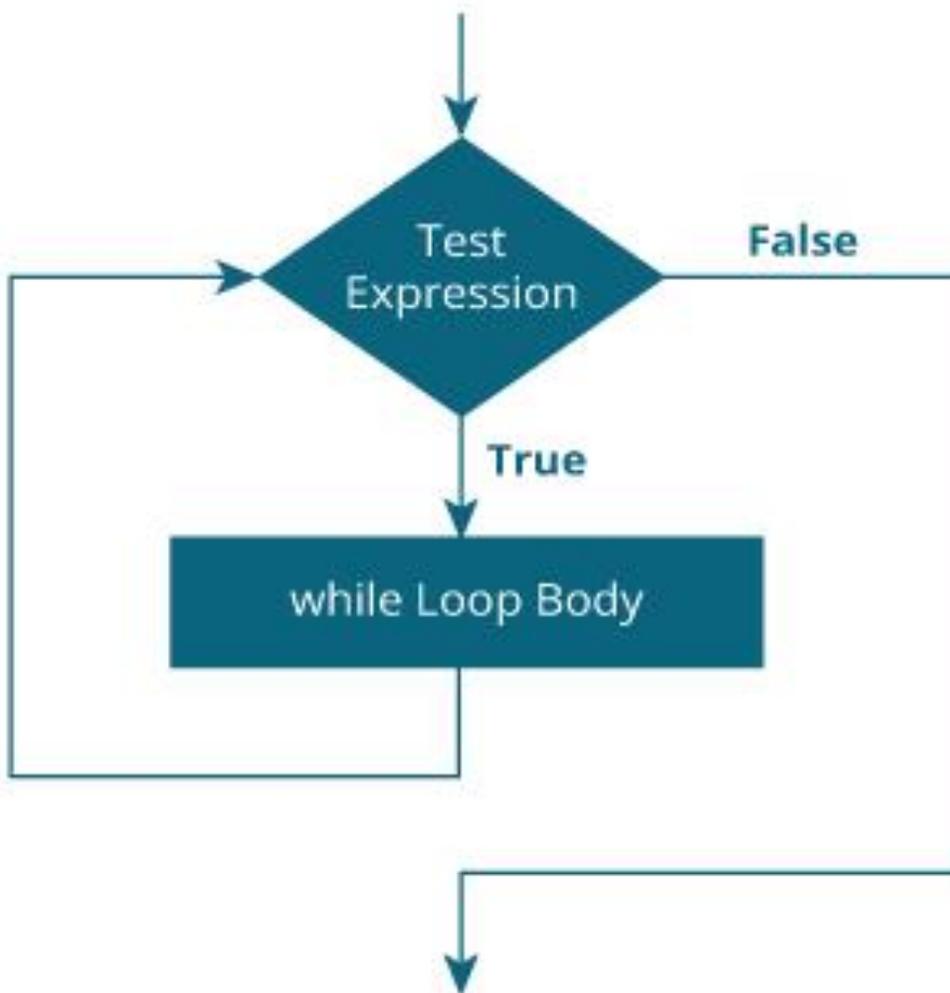
Example 1: while loop

```
// Print numbers from 1 to 5

#include <stdio.h>
int main()
{
    int i = 1;

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

    return 0;
}
```

Output

```
1
2
3
4
5
```

Here, we have initialized i to 1.
1. When i is 1, the test expression i <= 5 is true. Hence, the body of the while loop is executed. This prints 1 on the screen and the value of i is increased to 2.
2. Now, i is 2, the test expression i <= 5 is again true. The body of the while loop is executed again. This prints 2 on the screen and the value of i is increased to 3.
3. This process goes on until i becomes 6. When i is 6, the test expression i <= 5 will be false and the loop terminates.

do...while loop

The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated. The syntax of the do...while loop is:

```
do
```

```
{
    // statements inside the body of the loop
}
while (testExpression);
```

How do...while loop works?

- The body of do...while loop is executed once. Only then, the test expression is evaluated.

- If the test expression is true, the body of the loop is executed again and the test expression is evaluated.

- This process goes on until the test expression becomes false.

- If the test expression is false, the loop ends.

Flowchart of do...while Loop
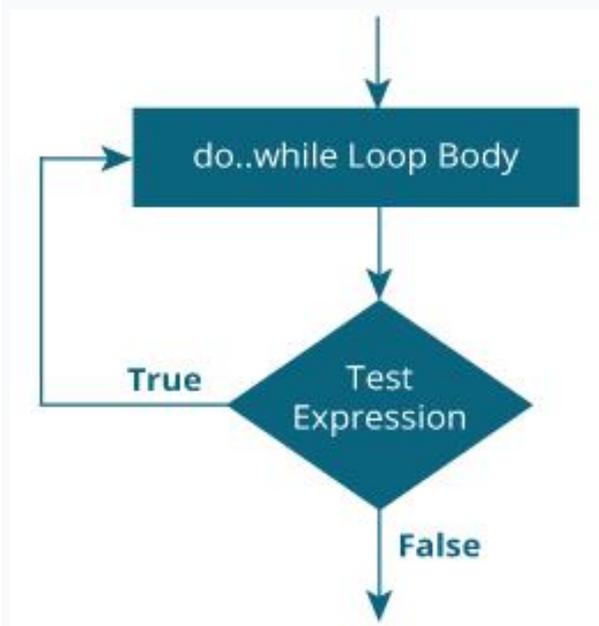
Example 2: do...while loop

```
// Program to add numbers until the user enters zero
```

```c
#include <stdio.h>
int main()
{
    double number, sum = 0;

    // the body of the loop is executed at least once
    do
    {
        printf("Enter a number: ");
        scanf("%lf", &number);
        sum += number;
    }
    while(number != 0.0);

    printf("Sum = %.2lf",sum);

    return 0;
}
```

Output

```
Enter a number: 1.5
Enter a number: 2.4
Enter a number: -3.4
Enter a number: 4.2
Enter a number: 0
Sum = 4.70
```

multiple loop variables

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

The syntax for a **nested for loop** statement in C is as follows −

```
for ( init; condition; increment ) {

   for ( init; condition; increment ) {
      statement(s);
   }
   statement(s);
}
```
The syntax for a **nested while loop** statement in C programming language is as follows −

```
while(condition) {

   while(condition) {
      statement(s);
   }
   statement(s);
}
```
The syntax for a **nested do…while loop** statement in C programming language is as follows −

```
do {
   statement(s);

   do {
      statement(s);
   }while( condition );

}while( condition );
```
A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100 − Live Demo

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int i, j;

   for(i = 2; i<100; i++) {

      for(j = 2; j <= (i/j); j++)
```

```
    if(!(i%j)) break; // if factor found, not prime
    if(j > (i/j)) printf("%d is prime\n", i);
  }

  return 0;
}
```
When the above code is compiled and executed, it produces the following result −

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

assignment operators

The following table lists the assignment operators supported by the C language −

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side	C = A + B will assign the value

	operands to left side operand	of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2

&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Example

Try the following example to understand all the assignment operators available in C –

```
#include <stdio.h>

main() {

   int a = 21;
   int c ;

   c =  a;
   printf("Line 1 - =  Operator Example, Value of c = %d\n", c );

   c +=  a;
   printf("Line 2 - += Operator Example, Value of c = %d\n", c );

   c -=  a;
   printf("Line 3 - -= Operator Example, Value of c = %d\n", c );

   c *=  a;
   printf("Line 4 - *= Operator Example, Value of c = %d\n", c );

   c /=  a;
   printf("Line 5 - /= Operator Example, Value of c = %d\n", c );

   c  = 200;
   c %=  a;
```

```
    printf("Line 6 - %= Operator Example, Value of c = %d\n", c );

   c <<=  2;
   printf("Line 7 - <<= Operator Example, Value of c = %d\n", c );

   c >>=  2;
   printf("Line 8 - >>= Operator Example, Value of c = %d\n", c );

   c &=  2;
   printf("Line 9 - &= Operator Example, Value of c = %d\n", c );

   c ^=  2;
   printf("Line 10 - ^= Operator Example, Value of c = %d\n", c );

   c |=  2;
   printf("Line 11 - |= Operator Example, Value of c = %d\n", c );
}
```

When you compile and execute the above program, it produces the following result −

Line 1 - = Operator Example, Value of c = 21
Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - %= Operator Example, Value of c = 11
Line 7 - <<= Operator Example, Value of c = 44
Line 8 - >>= Operator Example, Value of c = 11
Line 9 - &= Operator Example, Value of c = 2
Line 10 - ^= Operator Example, Value of c = 0
Line 11 - |= Operator Example, Value of c = 2

using break and continue

In this tutorial, you will learn about c programming break continue
statements. Break and continue statements are used to jump out of the loop and
continue looping. Break and continue statements in c

Till now, we have learned about the looping with which we can repeatedly execute the
code such as, for loop and while & do … while loop.

Just think what will you do when you want to jump out of the loop even if the condition is
true or continue repeated execution of code skipping some of the parts?

For this C provides break and continue statements. By the help of these statements, we can jump out of loop anytime and able continue looping by skipping some part of the code.

The break statement in C

In any loop break is used to jump out of loop skipping the code below it without caring about the test condition.

It interrupts the flow of the program by breaking the loop and continues the execution of code which is outside the loop.

The common use of break statement is in switch case where it is used to skip remaining part of the code.

How does break statement works?

```
for ( expression )
{
    statement1;

    ....

    if (condition)   true
        break;

    .....

    statement2;
}
        out of the loop
```

```
while (test condition)
{
    statement1;

    ....

    if (condition)   true
        break;

    .....

    statement2;
}
        out of the loop
```
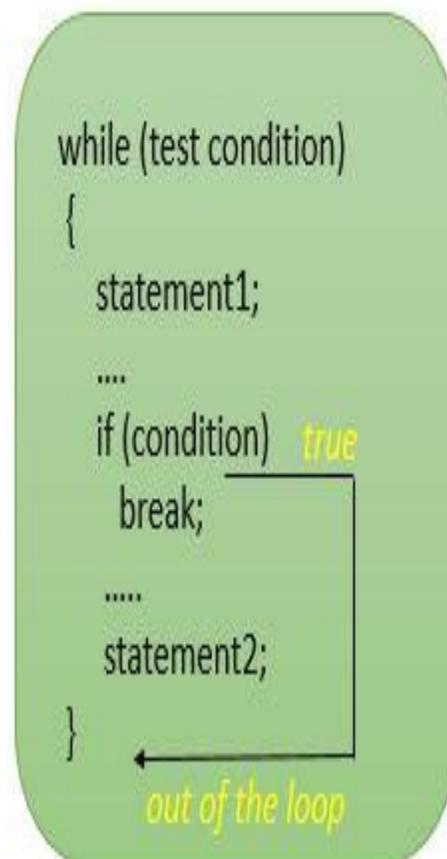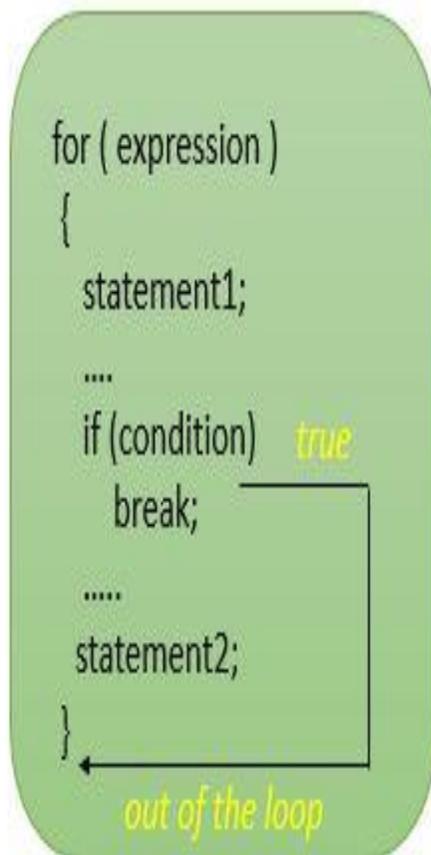
Structure of Break statement

In while loop

```
while (test_condition)

{

  statement1;

  if (condition )

    break;

  statement2;

}
```

In do…while loop

```
do

{

  statement1;

  if (condition)

    break;

  statement2;

}while (test_condition);
```

In for loop

```
for (int-exp; test-exp; update-exp)

{

  statement1;

 if (condition)

    break;

  statement2;

}
```

Now in above structure, if test_condition is true then the statement1 will be executed and again if the condition is true then the program will encounter break statement which will cause the flow of execution to jump out of loop and statement2 below if statement will be skipped.

Programming Tips

break statement is always used with if statement inside a loop and loop will be terminated whenever break statement is encountered.

Example: C program to take input from the user until he/she enters zero.

```
#include <stdio.h>

int main ()

{

 int a;

 while (1)

 {
```

```
    printf("enter the number:");

    scanf("%d", &a);

   if ( a == 0 )

      break;

  }

 return 0;

}
```

Explanation

In above program, while is an infinite loop which will be repeated forever and there is no exit from the loop.

So the program will ask for input repeatedly until the user will input 0.

When the user enters zero, the if condition will be true and the compiler will encounter the break statement which will cause the flow of execution to jump out of the loop.

The continue statement in C

Like a break statement, continue statement is also used with if condition inside the loop to alter the flow of control.

When used in while, for or do...while loop, it skips the remaining statements in the body of that loop and performs the next iteration of the loop.

Unlike break statement, continue statement when encountered doesn't terminate the loop, rather interrupts a particular iteration.

How continue statement work?

```
        for ( expression )                        while (test condition)
        {                                          {
            statement1;                                statement1;

            ....                                       ....
            if (condition)                             if (condition)
    true      continue;                        true      continue;

            .....                                      .....
            statement2;                                statement2;
        }                                          }
```

top of the loop

top of the loop
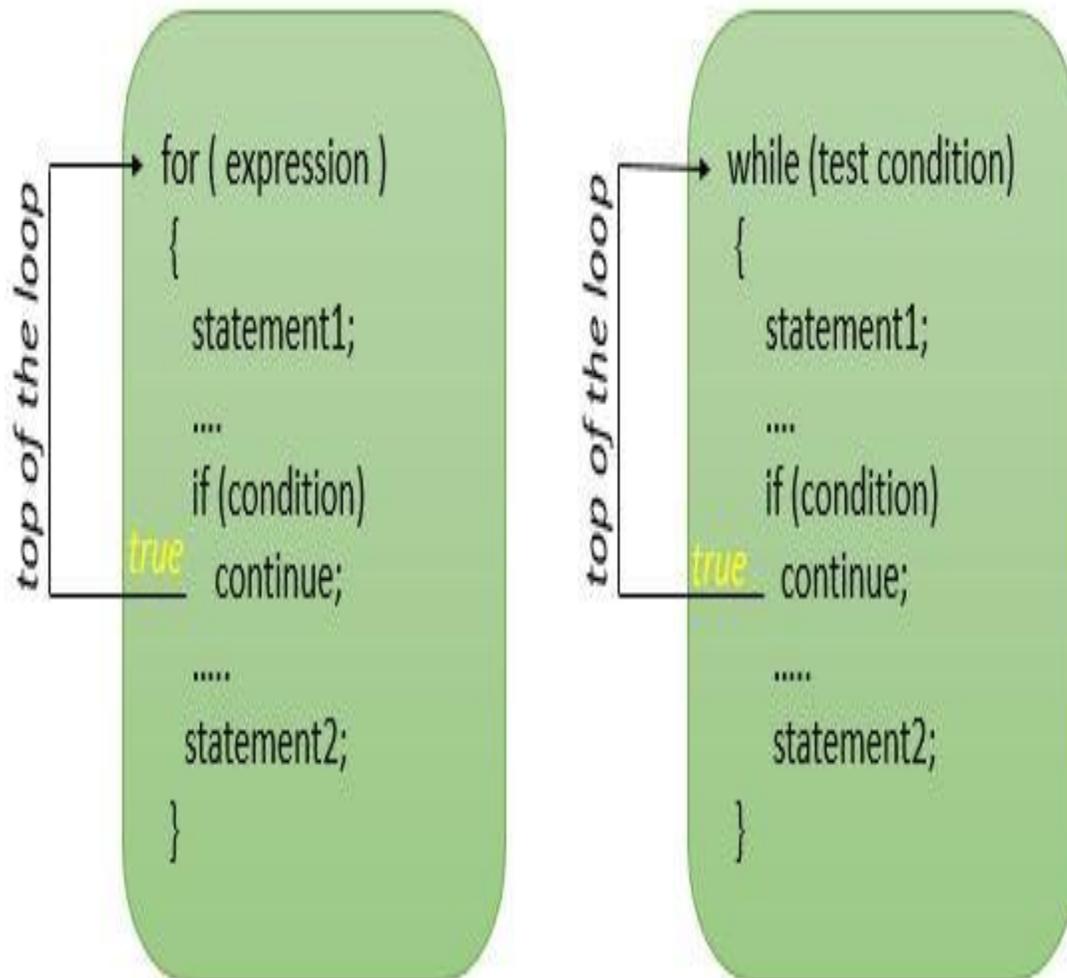
Structure of continue statement

In while loop

while (test_condition)

{

  statement1;

```
   if (condition )

      continue;

   statement2;

}
```

In do…while loop

```
do

{

  statement1;

  if (condition)

     continue;

  statement2;

}while (test_condition);
```

In for loop

```
for (int-exp; test-exp; update-exp)

{

  statement1;

  if (condition)

     continue;
```

```
  statement2;

}
```

Explanation

In above structures, if test_condition is true then the continue statement will interrupt the flow of control and block of statement2 will be skipped, however, iteration of the loop will be continued.

Example: C program to print sum of odd numbers between 0 and 10

```c
#include <stdio.h>

int main ()

{

 int a,sum = 0;

 for (a = 0; a < 10; a++)

 {


  if ( a % 2 == 0 )

    continue;

 sum = sum + a;

 }

printf("sum = %d",sum);

 return 0;
```

```
}
```

Output

```
sum = 25
```

Modular Programming:

Passing arguments by value

I will call what you are passing in a to a function the actual parameters, and where you receive them, the parameters in the function, the formal parameters. They are also called actual and formal arguments.

When passing parameters, what it is called and what happens can be confusing. It is less essential that you call it the "correct" thing than you know exactly what is happening. It is critical to have a good mental model, a valid memory picture of the process.

Recall that when you call a function, a chunk of memory called an activation record is allocated. Critical to the discussion here is that this memory holds the formal parameter values and function local variables.

By definition, pass by value means you are making a copy in memory of the actual parameter's value that is passed in, a copy of the contents of the actual parameter. Use pass by value when when you are only "using" the parameter for some computation, not changing it for the client program.

In pass by reference (also called pass by address), a copy of the address of the actual parameter is stored. Use pass by reference when you are changing the parameter passed in by the client program.

Consider a swapping function to demonstrate pass by value vs. pass by reference. This function, which swaps ints, cannot be done in Java.

```
  main() {
    int i = 10, j = 20;
    swapThemByVal(i, j);
    cout << i << " " << j << endl;    // displays 10  20
    swapThemByRef(i, j);
    cout << i << " " << j << endl;    // displays 20  10
    ...
  }
```

```
void swapThemByVal(int num1, int num2) {
    int temp = num1;
    num1 = num2;
    num2 = temp;
}

void swapThemByRef(int& num1, int& num2) {
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```
First, we show the memory picture for swapThemByVal. The activation record holds the memory for the two parameters, num1 and num2, and the local variable, temp. A copy of the values from main, in the contents of i and j, are copied. All the manipulation is done in the activation record.
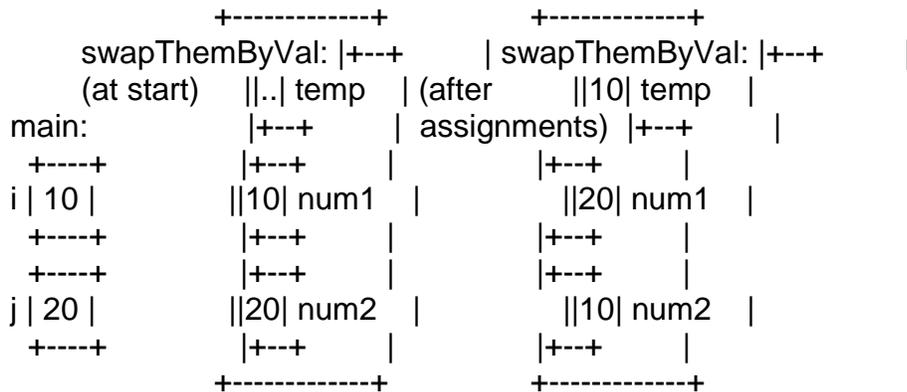
```
                  +-------------+          +-------------+
       swapThemByVal: |+--+       | swapThemByVal: |+--+       |
       (at start)   ||..| temp  | (after       ||10| temp  |
main:               |+--+       | assignments) |+--+       |
  +----+            |+--+       |              |+--+       |
i | 10 |            ||10| num1  |              ||20| num1  |
  +----+            |+--+       |              |+--+       |
  +----+            |+--+       |              |+--+       |
j | 20 |            ||20| num2  |              ||10| num2  |
  +----+            |+--+       |              |+--+       |
                  +-------------+          +-------------+
```
The contents of memory of i and j don't change. The contents of memory in the function's activation record changes, but when the function terminates, the memory is released and the changes are lost.

Contrast this with passing by reference. The addresses of i and j are passed (noted by the arrows) by reference. The compiler knows they are references so when the parameters are referred to in the function, the compiler dereferences num1 and num2 automatically so i and j of main's memory are changed.

```
                  +-------------+
       swapThemByRef: |+--+       |
                      ||..| temp  |
main:                 |+--+       |
  +----+              |+--+       |
i | 10 | <------------||--| num1  |
  +----+              |+--+       |
  +----+              |+--+       |
j | 20 | <------------||--| num2  |
  +----+              |+--+       |
                  +-------------+
```
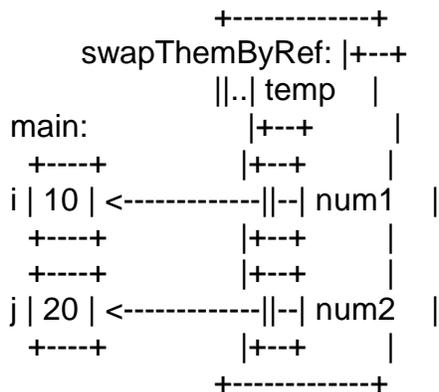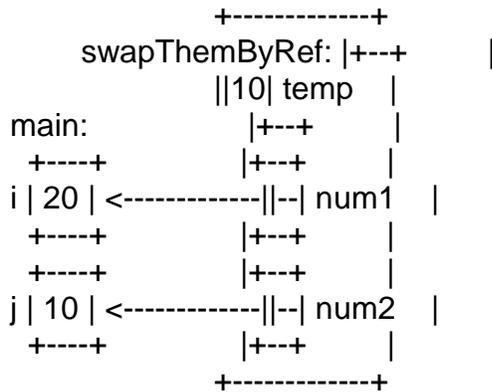
After the assignments:
```
                +-------------+
    swapThemByRef: |+--+        |
              ||10| temp    |
main:             |+--+        |
  +----+          |+--+        |
i | 20 | <-------------||--| num1    |
  +----+          |+--+        |
  +----+          |+--+        |
j | 10 | <-------------||--| num2    |
  +----+          |+--+        |
                +-------------+
```
This is the essence of pass by value vs. pass by reference. It doesn't matter if the parameters are primitive types, arrays, or objects, either a copy is made or an address is stored. As noted elsewhere, when objects are copied, the copy constructor is called to do the copying.

Typically if you aren't going to change a variable, you use pass by value. But if you are passing something in that uses a lot of memory, i.e., passing an object or passing an array, even if you aren't changing it, you use what I like to call fake pass by value.

For efficiency, you pass by reference so only the address is passed, but you put a const in front of it. This casts it to a constant for use in the function. Note that if this function passes to some other function, it is now constant object or array. For example:
```
main() {
  SomeBigClass x(100);
  // initialize and do whatever with x
  doSomething(x);
  ...
}

void doSomething(const SomeBigClass& x) {
  ...
}
```

scope rules and global variables

Scope of an identifier is the part of the program where the identifier may directly be accessible. In C, all identifiers are lexically(or statically) scoped. C scope rules can be covered under the following two categories.
There are basically 4 scope rules:

SCOPE	MEANING
File Scope	Scope of a Identifier starts at the beginning of the file and ends at the end of the file. It refers to only those Identifiers that are declared outside of all functions. The Identifiers of File scope are visible all over the file Identifiers having file scope are global
Block Scope	Scope of a Identifier begins at opening of the block / '{' and ends at the end of the block / '}'. Identifiers with block scope are local to their block
Function Prototype Scope	Identifiers declared in function prototype are visible within the prototype
Function scope	Function scope begins at the opening of the function and ends with the closing of it. Function scope is applicable to labels only. A label declared is used as a target to goto statement and both goto and label statement must be in same function

Let's discuss each scope rules with examples.

File Scope: These variables are usually declared outside of all of the functions and blocks, at the top of the program and can be accessed from any portion of the program. These are also called the global scope variables as they can be globally accessed.

Example 1:

// C program to illustrate the global scope

#include <stdio.h>

```cpp
// Global variable
int global = 5;

// global variable accessed from
// within a function
void display()
{
    printf("%d\n", global);
}

// main function
int main()
{
    printf("Before change within main: ");
    display();

    // changing value of global
    // variable from main function
    printf("After change within main: ");
    global = 10;
    display();
}
```

Output:

Before change within main: 5

After change within main: 10

separate compilation

C++ supports separate compilation, where pieces of the program can be compiled independently through the two stage approach of compilation and then linking, so changes to one class would not necessarily require the re-compilation of the other classes. The compiled pieces of code (.o or .obj files)[8] are combined through the use of the linker (in the use of Borland C++ it is **ilink32.exe**). Separate compilation allows programs to be compiled and tested one class at a time, even built into libraries for later use. It is therefore good practice to place each class in a separate source file to take full advantage of separate compilation with the C++ language.

The source code for each class is stored in two files:

- A source file (.cpp) - the implementation of the methods.

- A header file (.h) - the definition of the class.

The header file contains the declarations for the methods contained in the cpp file, allowing for these cpp files to be compiled into libraries. The cpp file will define the methods and by including the header file within the cpp file you will ensure consistency between the declarations and definitions.

So the ◾Account class would take the form of three separate files:

- **Account.h** - That stores the class declaration and definition.

- **Account.cpp** - That stores the method definitions for that class.

- **Application.cpp** - That stores the application, i.e. the ◾main() method for the application.

So the header file (Account.h) will have the form:

```
1
2
3   #include<iostream>
4   #include<string>
5
6   using std::string;  ❶  // only string is required
7
8   class Account{
9
10    protected:
11
12      int accountNumber;
13      float balance;
14      string owner;
15
16    public:
17
18      Account(string owner, float aBalance, int anAccountNumber);
19      Account(float aBalance, int anAccountNumber);
20      Account(int anAccountNumber);
21      Account(const Account &sourceAccount);
22
23      ...
24  };
25
```
❶

> You should not place using directives in header files where possible. If we were to use using namespace std; in our header file, all cpp files that include this header would also include this using directive. This would have the effect of turning off namespaces in your project (in this case for std only).

The implementation file (Account.cpp) will have the form:

```
#include "Account.h"

using namespace std;

Account::Account(string anOwner, float aBalance, int anAccNumber):
        accountNumber(anAccNumber), balance(aBalance),
        owner (anOwner) {}

Account::Account(float aBalance, int anAccNumber) :
        accountNumber(anAccNumber), balance(aBalance),
        owner ("Not Defined") {}

Account::Account(int anAccNumber):
        accountNumber(anAccNumber), balance(0.0f),
        owner ("Not Defined") {}

Account::Account(const Account &sourceAccount):
        accountNumber(sourceAccount.accountNumber + 1),
        balance(0.0f),
        owner (sourceAccount.owner) {}

...
```

And the application (Application.cpp) will have the form:

```
#include "Account.h"

int main()
{
   Account a = Account("Derek Molloy",35.00,34234324);

   ...
}
```

To compile the application, you must now specify the files to be used in the compilation. So, to compile all the files at once use: **bcc32 Application.cpp Account.cpp**, where one of the source files contains a ▪main() method. This can be seen in Figure 3.11, "Compilation, and the output from the Separately Compiled Example.".

195

Figure 3.11. Compilation, and the output from the Separately Compiled Example.

Just before we continue we need to briefly discuss preprocessor directives. Preprocessor directives are orders for the preprocessor, not for the program itself. They must be specified in a single line of code and should not end with a ; (semicolon). Some preprocessor directives are: #include (insert a header file here), #define (define a constant macro)

```
#define PI 3.14
```

#undef (removes definition), #if, #ifdef, #ifndef, #endif, #else, #elif (control directives to remove part of a program depending on the condition),

```
#ifndef MAX_WIDTH
#define MAX_WIDTH 1000
#endif
```

196

#line (allows control over compile time error messages), #error (allows us to abort compilation if required), e.g.

```
#ifndef __cplusplus
#error You need a C++ compiler for this code!
#endif
```

and #pragma (used for compiler options specific to a particular platform and compiler).

If there are multiple classes, some of which use the same parent, you can use compiler directives to prevent the re-definition of the same class, which would result in a compiler error. These directives can be placed around the class definition such as:

```
#ifndef currentAccount_h  //check not already defined
#define currentAccount_h        //if not, define!

#include "Account.h"      //include the account header

class CurrentAccount: public Account{

  protected:
    float overDraftLimit;
    ...

        public:

    CurrentAccount(int theNumber, char* theOwner, float theOverdraftLimit);
    ...
};

#endif  // currentAcount_h
```

In this case, the compiler directives simply state that if the CurrentAccount class is already defined then do not redefine it. This is determined by the currentAccount_h value, that if undefined is simply defined, and so used as a flag. This process is to ensure that we have not broken the **C++ single definition rule**: You can declare anything as many times as you want, but you can only define it once.

Fortunately, for the purpose of professional development, tools such as Integrated Development Environments(IDEs) can automatically insert the required preprocessor directives.

Linkage

Genetic linkage describes the way in which two genes that are located close to each other on a chromosome are often inherited together. In 1905, William Bateson, Edith Rebecca Saunders, and Reginald C. Punnett noted that the traits for flower color and pollen shape in sweet pea plants appeared to be linked together. A few years later, in 1911, Thomas Hunt Morgan, who was studying heredity in fruit flies, noticed that the eye color of a fly was associated with the fly's sex and hypothesized that the two traits were linked together. These observations led to the concept of genetic linkage, which describes how two genes that are closely associated on the same chromosome are frequently inherited together. In fact, the closer two genes are to one another on a chromosome, the greater their chances are of being inherited together or linked. In contrast, genes located farther away from each other on the same chromosome are more likely to be separated during recombination, the process that recombines DNA during meiosis. The strength of linkage between two genes, therefore, depends upon the distance between the genes on the chromosome.

building your own modules

This chapter describes how to create your own RADIUS module. Extending the RADIUS server is a task for advanced users with special requirements. Creating new modules requires C or C++ programming skills and an understanding of the RADIUS protocol and thread programming.

Before creating a custom module you should be familiar with:

- Understanding RADIUS Manager Modules

- Configuring RADIUS Manager

- Using the Authentication and Authorization Modules

About Creating Custom Modules

Before creating your own module, be sure that the task you wish to accomplish cannot be performed by one of the modules supplied with the RADIUS Manager.

You may write a custom module which authenticates and modifies incoming packets to handle special conditions. You also might want to modify logins to provide greater flexibility in managing requests. With custom modules, you can:

- Add or remove part of a login.

- Authenticate a request against an alternate database.

- Handle requests with identical logins.

Use **mod_example** as a template for your new module. Customize the required sections, and link your custom modules with the core functionality module libraries to produce new RADIUS server binaries.

Checklist for Creating Custom Modules

This checklist provides an overview of tasks that must be performed when you create a custom module. To create a custom module:

1. Review the Module Class Model as implemented by RADIUS Manager. See "About the Module Class Model".

2. After RADIUS Manager is installed, locate the **mod_example** template in BRM_home**/source/apps/radius**, where BRM_home is the directory in which BRM components are installed.

3. Copy the **mod_example.cpp** and **mod_example.h** files. Store the copies in the BRM_home**/source/apps/radius** directory.

4. Add the new module to the definitions file (BRM_home**/source/apps/radius/moddef.cpp**) file.

5. Add functionality to the new module, by modifying the **module.h** and **module.cpp** files.

6. Add an object for the new module to the Makefile:

   OBJS = moddef.o <mod_new>.o

7. Build your custom module and link it with the libraries to produce a new **radiusid** executable:

1. Stop the RADIUS server, if it is running.

2. Copy **pin_radiusid** from the current directory to BRM_home/**bin**.

3. Edit the RADIUS configuration file (BRM_home**/apps/radius/config**) to add the new module to the module chain.

4. Start the RADIUS server.

5. Test the new module.

Modifying the Definitions File

Modify the definitions file (BRM_home**/source/apps/radius/moddef.cpp**) file by declaring the module master creation function and adding the name of your module to the module definition table.

Add this line:

{<mod_examplename>, MDF_NONE, RadiusModuleNew_create}

To this section of the **moddef.cpp** file:

```
....

   extern RadiusModule *RadiusModuleTransform_create(string theName, string theType,

   ModuleDefFlags theFlags);


   const ModuleConfigType module_config[] = {

   { "mod_null",    MDF_NONE,  RadiusModuleNull_create },

   #ifdef __unix

   { "mod_unixpwd",  MDF_NONE,  RadiusModuleUnixpwd_create },

   //{ "mod_ipass",    MDF_NONE,  RadiusModuleIPass_create },

   #endif

   { "mod_proxy",    MDF_NONE,  RadiusModuleProxy_create },

   { "mod_text",     MDF_NONE,  RadiusModuleText_create },

   { "mod_logging",  MDF_NONE,  RadiusModuleLogging_create },

   { "mod_transform",    MDF_NONE,  RadiusModuleTransform_create },

   { 0, MDF_NONE, 0 }

   { "mod_pin",      MDF_NONE,  RadiusModulePin_create },

};
```

Adding Functionality to a Custom Module

To add functionality to a custom module you must understand the C++ API interface and the Module Class Module.

About the C++ API Interface to RADIUS Modules

To create a new module type, declare new C++ classes which inherit from a set of base classes, and then implement the module type-specific functionality. The virtual base class for module masters is **RadiusModule**. See **modbase.h** for detailed information about the **RadiusModule**.

The module masters are instantiated by a special module definition table.
See **moddef.cpp** for detailed information about the module definition table. The API support files contain specific information about using each file.

Support Header Files

The support header files (**module.h**) are self-documenting. See the following files for details:

In **source/apps/radius**:

- **moddef.h**

In **source/apps/radius/include/general**:

- **applog.h** - Application log functions
- **debuglog.h** - Debug log functions
- **ipconvert.h** - IP address translation utilities
- **list.h** - List processing functions
- **lstring.h** - String class
- **strutil.h** - String processing functions
- **usersconfig.h** - How to handle configuration files

In **source/apps/radius/include/modules**:

- **checksend.h** - Check and send support for modules
- **modbase.h** - Definitions of module classes
- **request.h** - Incoming RADIUS request
- **stdconfig.h** - Standard configuration keywords used in the configuration file
- **systemdict.h** - Accessing the RADIUS dictionary

In **source/apps/radius/include/radius**:

- **attr.h** - RADIUS attribute processing functions

- **dict.h** - Dictionary processing functions
- **packet.h** - RADIUS packet processing functions
- **password.h** - Processing the RADIUS password attribute
- **structdatatype.h** - Support for the struct data type in the dictionary
- **types.h** - Enumerated values for attributes and their types

About the Module Class Model

Before writing a custom module, you must understand the Module Class Model and the support APIs.

This section describes the C++ API interface to module masters and module workers. A virtual base class inheritance model is used. This means that creating a new module type involves declaring new C++ classes which inherit from a set of base classes and then implement the module type's specific functionality.

About Configuring Modules

The modules, which are configured in the RADIUS configuration file (BRM_home**/apps/radius/config**), form an ordered list. When a request is received from the core server, it is processed by each module in turn until one of the modules indicates that it has completely processed the request. Note that the behavior of a module is determined by its configuration, especially its type.

Each module indicates what should happen next by setting a "return value":

- MRT_CONTINUE indicates that the request should be passed to the next module.

- MRT_COMPLETE indicates that the request has been successfully processed. The response has been filled in by the module and should be sent to the requesting client (NAS).

- MRT_ERROR indicates that an error has occurred and the request should be discarded.

- MRT_DISCARD indicates that the request should be discarded. When an individual module receives a request, it performs the following tasks:

    o Checks to see that the request matches its check sections. If not, it returns MRT_CONTINUE.
    o Optionally makes changes to the incoming request by adding, deleting, or modifying attributes. (See **mod_transform** for an example).
    o Optionally makes changes to the outgoing response by setting the response type, and adding, deleting, or modifying attributes.

- Optionally adding attributes from any **send** sections to the outgoing response.
- Returns MRT_CONTINUE, MRT_COMPLETE or MRT_ERROR as appropriate.
- If the end of the module list is reached and no module has returned MRT_COMPLETE, the request is discarded.

About the Module Master

The virtual base class for module masters is RadiusModule. The salient parts are shown here from **modbase.h**. See **modbase.h** for detailed information.

```
class RadiusModule {

 protected:

   ModuleDefFlags flags;

   string name;

   string type;

   int ref_count;


   pthread_mutex_t mutex;

   ConfigEntry *shared_config;  // A copy of $CONFIG->(name)

   ConfigEntry *worker_config;  // A copy of $MODULES->(name)


   /*

   ** use()

   **

   ** Increments the reference count.

   */

   void use();
```

```cpp
public:
  /*
  ** RadiusModule()
  **
  ** Create a module master.
  ** The default operation intitialises name, type and flags from
  ** the parameters and sets the reference count to 1.
  */
  RadiusModule(string name, string type, ModuleDefFlags flags);


  /*
  ** ~RadiusModule()
  **
  ** Destroy the module master.
  ** Automatically called when the reference count reaches 0 (see use(), unuse())
  */
  virtual ~RadiusModule();


  /*
  ** unuse()
  **
  ** Decrement the reference count and if it reaches 0, destroys the object.
  */
  void unuse();
```

```
/*
** newConfig()
**
** Called when the configuration manager detects that
** new configuration information is available.
**
** Default implementation searches for the entry $MODULES->(name),
** where (name) is the name of this module entry, and stores
** a copy of that part of the configuration tree in 'worker_config'.
** It also searches for $CONFIG->(name) and stores the result (if found)
** in shared_config.
*/
virtual void newConfig(const ConfigEntry *top_config);


/*
** createWorker()
**
** User-defined function which creates a module worker.
** The worker object should take a COPY of its configuration,
** since during a reconfig, the worker may need to complete the
** current request before it dies.
**
** Must call use() to increment the reference count on this object.
*/
virtual RadiusModuleWorker *createWorker(int theThreadId) = 0;
```

```
 /*
 ** lock()
 ** unlock()
 **
 ** Synchronise access to this object.
 */
 void lock();
 void unlock();
};
```

Methods Defined in Derived Classes for the Master Module

These methods may be defined in the derived class. See the source
for **mod_example** as an example.

Constructor

You must provide a constructor in order to initialize the module master. The base class
constructor must be called.

Example:

```
RadiusModuleExample::RadiusModuleExample(string name_, string type_,
ModuleDefFlags flags_)

: RadiusModule(name_, type_, flags_)

{

}
```

Destructor

Destroys the object. Generally doesn't need to do anything.

Example:

```
RadiusModuleExample::~RadiusModuleExample()

{

}
```

newConfig()

This method is called both at startup and when a reconfiguration event occurs. It is the responsibility of the module master to extract the appropriate configuration from the configuration tree. The default implementation keeps a reference to both the module type configuration (**shared_config**) and the per-module configuration (**worker_config**). Thus, most modules do not require a special implementation of this method.

createWorker()

This method creates a new module worker and increments the reference count. Typically, this method simply creates a new module worker of the appropriate type. The module worker is responsible for taking a copy of both the per-module configuration (**worker_config**) and the module type configuration (**shared_config**), when appropriate. It must also call **use()** to ensure that the reference count is incremented.

Example:

```
RadiusModuleWorker *

RadiusModuleExample::createWorker(int thread_id)

{

assert(worker_config != 0);

use();

return(new RadiusModuleExampleWorker(this, worker_config,

   shared_config, thread_id));

}
```

About Worker Modules

The module worker is where most of the work of the module is accomplished. The virtual base class for module workers is RadiusModuleWorker. The salient parts are shown here from modbase.h. See **modbase.h** for detailed information.

Methods Defined in Derived Classes for Worker Modules

This section describes the APIs to module workers.

Constructor (required)

A constructor is required in order to initialize the module worker information. The base class constructor must be called to initialize the parent pointer and to copy the **worker_config**. Any configuration information that will be referenced must be copied. Specific configuration can also be extracted from the worker_config, the **shared_config**, or both. A **CheckSend** object is normally created here. Also, any standard options should be parsed here.

Example:

```
    RadiusModuleExampleWorker::RadiusModuleExampleWorker(RadiusModule
*parent_,

    const ConfigEntry *worker_config,

    const ConfigEntry *shared_config,

    int thread_id)

    : RadiusModuleWorker(parent_, worker_config, thread_id)

    {

    checksend = new RadiusCheckSend(&config);

    action = config.getValue(MOD_EXAMPLE_ACTION);

    if (action == "") {

    APP_LOG(("[W]%s: No action specified in config. Using action=ignore.",

    (const char *)getName()));

    action = MOD_EXAMPLE_ACTION_IGNORE;

    }

    /* No shared_config for this module type */
```

```
        }
```

Destructor

Destroys the object.

Example:

```
    RadiusModuleExampleWorker::~RadiusModuleExampleWorker()

    {

    delete checksend;

    }
```

acceptRequest()

Processes the given request. The method:

- May modify **request->input** if appropriate, by adding, modifying, or deleting attributes.
- May modify **request->output** if appropriate, by setting the type, or adding, modifying, or deleting attributes.
- Should call **checksend->check()** as the first thing, if appropriate, which should almost always be the case.
- Should call **checksend->addSendAttr()** as the last thing when sending a response.

Returns one of MRT_..., such as:

```
    RadiusModuleWorker::ModuleReturnType

    RadiusModuleExampleWorker::acceptRequest(RadiusModuleRequest *request)

    {

    // Do standard check processing

    if (checksend->check(request->input, getSystemDict()) == 0) {

    return(MRT_CONTINUE);

    }

    DEBUG_LOG(("%s: check succeeded", (const char *)getName()));
```

```
if (action == MOD_EXAMPLE_ACTION_DISCARD) {

return(MRT_DISCARD);

}


if (action == MOD_EXAMPLE_ACTION_IGNORE) {

return(MRT_CONTINUE);

}


if (action == MOD_EXAMPLE_ACTION_NAK) {

switch (request->input->getType()) {

case PW_ACCESS_REQUEST:

request->output->setType(PW_ACCESS_REJECT);

break;


case PW_ACCOUNTING_REQUEST:

request->output->setType(PW_ACCOUNTING_RESPONSE);

break;


default:

return(MRT_DISCARD);

}

}

else if (action == MOD_EXAMPLE_ACTION_ACK) {

switch (request->input->getType()) {
```

```
    case PW_ACCESS_REQUEST:

    request->output->setType(PW_ACCESS_ACCEPT);

    break;


    case PW_ACCOUNTING_REQUEST:

    request->output->setType(PW_ACCOUNTING_RESPONSE);

    break;


    default:

    return(MRT_DISCARD);

    }

    }

    else {

    APP_LOG(("%s: unknown action: %s",

    (const char *)getName(), (const char *)action));

    return(MRT_ERROR);

    }


    checksend->addSendAttr(request->output, getSystemDict(),

    RadiusCheckSend::ADD_APPEND);


    return(MRT_COMPLETE);
```

Instantiating Module Masters

The sections above describe the APIs to module masters and module workers; however, they don't explain how module masters are instantiated.This is managed by a special module definition table.The excerpt below is from **moddef.cpp**:

```
    ....

    extern RadiusModule *RadiusModuleTransform_create(string theName, string
theType,

    ModuleDefFlags theFlags);


    const ModuleConfigType module_config[] = {

    { "mod_null",    MDF_NONE,  RadiusModuleNull_create },

    #ifdef __unix

    { "mod_unixpwd",  MDF_NONE,  RadiusModuleUnixpwd_create },

    //{ "mod_ipass",    MDF_NONE,  RadiusModuleIPass_create },

    #endif

    { "mod_proxy",    MDF_NONE,  RadiusModuleProxy_create },

    { "mod_text",     MDF_NONE,  RadiusModuleText_create },

    { "mod_logging",  MDF_NONE,  RadiusModuleLogging_create },

    { "mod_transform",    MDF_NONE,  RadiusModuleTransform_create },

    { 0, MDF_NONE, 0 }

    { "mod_example", MDF_NONE,  RadiusModuleExample_create },

    { "mod_pin",     MDF_NONE,  RadiusModulePin_create },

};
```

Note:
Configure the **mod_pin** module after configuring all the other modules
because **mod_pin** prepares the RADIUS response and sends it to the client.

This table associates Module Type names with functions that know how to create a
module master of the associated class.You can modify this table to add custom
modules.

Example of a module master creation function:

```
RadiusModule *

RadiusModuleExample_create(string name, string type, ModuleDefFlags flags)

{

return(new RadiusModuleExample(name, type, flags));

}
```

This function creates an object of the appropriate type and returns it.

Sample Code for a Custom Module

This sample prints the name of the incoming RADIUS attributes and their values and appends the domain name to the user name attribute.

```
const RadiusAttr        *theAttr = NULL;

RadiusAttr * newAttr  = NULL;

/* This code prints the names of the incoming RADIUS atttributes and their valus */

DEBUG_LOG(("printing all attributes in the packet"));

while ((theAttr = request->input->getEntry ( theAttr )) != 0 ) {

string name = theAttr->printName(getSystemDict());

string value = theAttr->printValue(getSystemDict());

DEBUG_LOG(("attribute name = %s", (const char *)name));

DEBUG_LOG(("attribute value = %s", (const char *)value));


if (theAttr->getCode() <= PW_LAST_VALID_ATTR_CODE) {

newAttr  = new RadiusAttr (theAttr->getCode(),

theAttr->getBuffer(),

theAttr->getBufferLength());

request->output->addAttr ((RadiusAttr *)newAttr);

}
```

```
}


/* the following piece of code will append a domain name to the

user-name attribute i.e if the username is joe, this code

will change it to joe@myisp.com, the 'add_domain' keywords

must be defined in your .h file and specified in the config file.

*/

DEBUG_LOG(("appending domain name to the user-name attribute"));

theAttr = request->input->getEntry ( PW_USER_NAME, NULL);

string value = theAttr->printValue(getSystemDict());

if (add_domain != "") {

DEBUG_LOG(("adding domain %s to username %s", add_domain.PeekString(),
value.PeekString()));

value += add_domain;

RadiusAttr * newAttr  = new RadiusAttr (PW_USER_NAME, value);

const RadiusAttr * modAttr  = request->input->modifyAttr (theAttr, newAttr);

RadiusAttr  * outAttr  = new RadiusAttr (modAttr->getCode(), modAttr->getBuffer(),
modAttr->getBufferLength());

request->output->addAttr ((RadiusAttr *)outAttr);
```

Adding a New Module to the RADIUS Configuration File

Edit the RADIUS configuration file (BRM_home**/apps/radius/config**) to add the new module to the module chain.

Example:

```
type=<mod_new>

status=enables
```

```
<check>

<module specfic actions>

<send>

<etc.)
```

Starting and Stopping the RADIUS Daemon

When you finish modifying the RADIUS configuration file you must restart the RADIUS daemon.

Use this procedure:

1. Check to ensure that the RADIUS **config** file and dictionary file are in the directory BRM_home**/apps/radius**.

2. Run either the start or stop script, BRM_home**/bin/start_radius** or BRM_home**/bin/stop_radius**. These scripts can be run manually but you should make them part of the software initialization at startup time.

3. If you want **pin_radiusd** to start automatically when you restart the machine.

4. Run the BRM_home**/bin/install_radius** script after you install the software. The **install_radius** script puts the required entries in the **/etc/rc2.d** directory to start **pin_radiusd** automatically.

UNIT IV

Arrays:

Array notation and representation

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, …, and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and …, numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

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

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [arraySize];
This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

double balance[10];
Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

balance[4] = 50.0;
The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −

manipulating array elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];
The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays −

```
#include <stdio.h>

int main () {

   int n[ 10 ]; /* n is an array of 10 integers */
   int i,j;

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; /* set element at location i to i + 100 */
   }

   /* output each array element's value */
   for (j = 0; j < 10; j++ ) {
      printf("Element[%d] = %d\n", j, n[j] );
   }

   return 0;
}
```
When the above code is compiled and executed, it produces the following result −

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
```

Element[8] = 108
Element[9] = 109

using multidimensional arrays

Multidimensional Arrays can be defined in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

Syntax:

data_type[1st dimension][2nd dimension][]..[Nth
dimension] **array_name** = **new data_type**[size1][size2]….[sizeN];

where:

- **data_type**: Type of data to be stored in the array. For example: int, char, etc.
- **dimension**: The dimension of the array created.
  For example: 1D, 2D, etc.
- **array_name**: Name of the array
- **size1, size2, …, sizeN**: Sizes of the dimensions respectively.

Examples:

Two dimensional array:

int[][] twoD_arr = new int[10][20];

Three dimensional array:

int[][][] threeD_arr = new int[10][20][30];

Size of multidimensional arrays: The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.
For example:
The array **int[][] x = new int[10][20]** can store a total of (10*20) = 200 elements.
Similarly, array **int[][][] x = new int[5][10][20]** can store a total of (5*10*20) = 1000 elements.

Two – dimensional Array (2D-Array)

Two – dimensional array is the simplest form of a multidimensional array. A two – dimensional array can be seen as an array of one – dimensional array for easier understanding.

Indirect Method of Declaration:
- **Declaration – Syntax:**

- **data_type[][] array_name = new data_type[x][y];**
- For example: int[][] arr = new int[10][20];

- **Initialization – Syntax:**
- **array_name[row_index][column_index] = value;**
- For example: arr[0][0] = 1;

Example:
filter_none
edit
play_arrow
brightness_4

```
class GFG {
    public static void main(String[] args)
    {

        int[][] arr = new int[10][20];
        arr[0][0] = 1;

        System.out.println("arr[0][0] = " + arr[0][0]);
    }
}
```

Output:

arr[0][0] = 1

Direct Method of Declaration:

Syntax:

data_type[][] array_name = {
            {valueR1C1, valueR1C2, ...},
            {valueR2C1, valueR2C2, ...}
          };

For example: int[][] arr = {{1, 2}, {3, 4}};

Example:
filter_none
edit

play_arrow
brightness_4

```
class GFG {
    public static void main(String[] args)
    {

        int[][] arr = { { 1, 2 }, { 3, 4 } };

        for (int i = 0; i < 2; i++)
            for (int j = 0; j < 2; j++)
                System.out.println("arr[" + i + "][" + j + "] = "
                        + arr[i][j]);
    }
}
```

Output:

arr[0][0] = 1

arr[0][1] = 2

arr[1][0] = 3

arr[1][1] = 4

Accessing Elements of Two-Dimensional Arrays

Elements in two-dimensional arrays are commonly referred by **x[i][j]** where 'i' is the row number and 'j' is the column number.

Syntax:

x[row_index][column_index]

For example:

int[][] arr = new int[10][20];

arr[0][0] = 1;

The above example represents the element present in first row and first column.

Note: In arrays if size of array is N. Its index will be from 0 to N-1. Therefore, for row_index 2, actual row number is 2+1 = 3.

Example:

filter_none
edit
play_arrow
brightness_4

```
class GFG {
    public static void main(String[] args)
    {

        int[][] arr = { { 1, 2 }, { 3, 4 } };

        System.out.println("arr[0][0] = " + arr[0][0]);
    }
}
```

Output:

arr[0][0] = 1

Representation of 2D array in Tabular Format: A two – dimensional array can be seen as a table with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A two – dimensional array 'x' with 3 rows and 3 columns is shown below:

Print 2D array in tabular format:

To output all the elements of a Two-Dimensional array, use nested for loops. For this two for loops are required, One to traverse the rows and another to traverse columns.

Example:

filter_none
edit
play_arrow
brightness_4

```
class GFG {
    public static void main(String[] args)
    {

        int[][] arr = { { 1, 2 }, { 3, 4 } };

        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                System.out.print(arr[i][j] + " ");
```

```
        }

            System.out.println();
        }
    }
}
```
Output:
```
1 2

3 4
```

Three – dimensional Array (3D-Array)

Three – dimensional array is a complex form of a multidimensional array. A three – dimensional array can be seen as an array of two – dimensional array for easier understanding.

Indirect Method of Declaration:
- **Declaration – Syntax:**
- **data_type[][][] array_name = new data_type[x][y][z];**
-        For example: int[][][] arr = new int[10][20][30];

- **Initialization – Syntax:**
- **array_name[array_index][row_index][column_index] = value;**
-        For example: arr[0][0][0] = 1;

Example:
filter_none
edit
play_arrow
brightness_4

```
class GFG {
    public static void main(String[] args)
    {

        int[][][] arr = new int[10][20][30];
        arr[0][0][0] = 1;

        System.out.println("arr[0][0][0] = " + arr[0][0][0]);
    }
}
```
Output:
```
arr[0][0][0] = 1
```

Direct Method of Declaration:

Syntax:

data_type[][][] array_name = {
        {
        {valueA1R1C1, valueA1R1C2, ...},
        {valueA1R2C1, valueA1R2C2, ...}
        },
        {
        {valueA2R1C1, valueA2R1C2, ...},
        {valueA2R2C1, valueA2R2C2, ...}
        }
        };

For example: int[][][] arr = { {{1, 2}, {3, 4}}, {{5, 6}, {7, 8}} };

Example:
filter_none
edit
play_arrow
brightness_4

```java
class GFG {
   public static void main(String[] args)
   {

      int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };

      for (int i = 0; i < 2; i++)
         for (int j = 0; j < 2; j++)
            for (int z = 0; z < 2; z++)
               System.out.println("arr[" + i
                        + "]["
                        + j + "]["
                        + z + "] = "
                        + arr[i][j][z]);
   }
}
```

Output:

arr[0][0][0] = 1

arr[0][0][1] = 2

arr[0][1][0] = 3

arr[0][1][1] = 4

arr[1][0][0] = 5

arr[1][0][1] = 6

arr[1][1][0] = 7

arr[1][1][1] = 8

Accessing Elements of Three-Dimensional Arrays

Elements in three-dimensional arrays are commonly referred by **x[i][j][k]** where 'i' is the array number, 'j' is the row number and 'k' is the column number.
Syntax:
x[array_index][row_index][column_index]

For example:

int[][][] arr = new int[10][20][30];

arr[0][0][0] = 1;

The above example represents the element present in the first row and first column of the first array in the declared 3D array.

Note: In arrays if size of array is N. Its index will be from 0 to N-1. Therefore, for row_index 2, actual row number is 2+1 = 3.
Example:
filter_none
edit
play_arrow
brightness_4

```
class GFG {
    public static void main(String[] args)
    {

        int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };

        System.out.println("arr[0][0][0] = " + arr[0][0][0]);
    }
}
```
Output:

arr[0][0][0] = 1

Representation of 3D array in Tabular Format: A three – dimensional array can be seen as a tables of arrays with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A three – dimensional array with 3 array containing 3 rows and 3 columns is shown below:

Print 3D array in tabular format:
To output all the elements of a Three-Dimensional array, use nested for loops. For this three for loops are required, One to traverse the arrays, second to traverse the rows and another to traverse columns.

Example:
filter_none
edit
play_arrow
brightness_4

```
class GFG {
    public static void main(String[] args)
    {

        int[][][] arr = { { { 1, 2 }, { 3, 4 } },
                    { { 5, 6 }, { 7, 8 } } };

        for (int i = 0; i < 2; i++) {

            for (int j = 0; j < 2; j++) {

                for (int k = 0; k < 2; k++) {

                    System.out.print(arr[i][j][k] + " ");
                }

                System.out.println();
            }
            System.out.println();
        }
    }
}
```

Output:

```
1 2
3 4


5 6
7 8
```

Inserting a Multi-dimensional Array during Runtime:
This topic is forced n taking user-defined input into a multidimensional array during runtime. It is focused on the user first giving all the input to the program during runtime and after all entered input, the program will give output with respect to each input accordingly. It is useful when the user wishes to make input for multiple Test-Cases with

multiple different values first and after all those things done, program will start providing output.

As an example, let's find the total number of even and odd numbers in an input array. Here, we will use the concept of a 2-dimensional array. Here are a few points that explain the use of the various elements in the upcoming code:

- Row integer number is considered as the number of Test-Cases and Column values are considered as values in each Test-Case.

- One for() loop is used for updating Test-Case number and another for() loop is used for taking respective array values.

- As all input entry is done, again two for() loops are used in the same manner to execute the program according to the condition specified.

- The first line of input is the total number of TestCases.

- The second line shows the total number of first array values.

- The third line gives array values and so on.

Implementation:

filter_none
edit
play_arrow
brightness_4

```java
import java.util.Scanner;

public class GFGTestCase {
    public static void main(
        String[] args)
    {
        // Scanner class to take
        // values from console
        Scanner scanner = new Scanner(System.in);

        // totalTestCases = total
        // number of TestCases
        // eachTestCaseValues =
        // values in each TestCase as
        // an Array values
        int totalTestCases, eachTestCaseValues;

        // takes total number of
        // TestCases as integer number
```

```java
totalTestCases = scanner.nextInt();

// An array is formed as row
// values for total testCases
int[][] arrayMain = new int[totalTestCases][];

// for loop to take input of
// values in each TestCase
for (int i = 0; i < arrayMain.length; i++) {
    eachTestCaseValues = scanner.nextInt();
    arrayMain[i] = new int[eachTestCaseValues];
    for (int j = 0; j < arrayMain[i].length; j++) {
        arrayMain[i][j] = scanner.nextInt();
    }
} // All input entry is done.

// Start executing output
// according to condition provided
for (int i = 0; i < arrayMain.length; i++) {

    // Initialize total number of
    // even & odd numbers to zero
    int nEvenNumbers = 0, nOddNumbers = 0;

    // prints TestCase number with
    // total number of its arguments
    System.out.println(
        "TestCase " + i + " with "
        + arrayMain[i].length + " values:");
    for (int j = 0; j < arrayMain[i].length; j++) {
        System.out.print(arrayMain[i][j] + " ");

        // even & odd counter updated as
        // eligible number is found
        if (arrayMain[i][j] % 2 == 0) {
            nEvenNumbers++;
        }
        else {
            nOddNumbers++;
        }
    }
    System.out.println();

    // Prints total numbers of
    // even & odd
    System.out.println(
```

```
            "Total Even numbers: " + nEvenNumbers
            + ", Total Odd numbers: " + nOddNumbers);
        }
    }
}
```
// This code is contributed by Udayan Kamble.

Input:
2
2
1 2
3
1 2 3

Output:
TestCase 0 with 2 values:
1 2
Total Even numbers: 1, Total Odd numbers: 1
TestCase 1 with 3 values:
1 2 3
Total Even numbers: 1, Total Odd numbers: 2

Input:
3
8
1 2 3 4 5 11 55 66
5
100 101 55 35 108
6
3 80 11 2 1 5

Output:
TestCase 0 with 8 values:
1 2 3 4 5 11 55 66
Total Even numbers: 3, Total Odd numbers: 5
TestCase 1 with 5 values:
100 101 55 35 108
Total Even numbers: 2, Total Odd numbers: 3
TestCase 2 with 6 values:
3 80 11 2 1 5
Total Even numbers: 2, Total Odd numbers: 4

arrays of unknown or varying size

In C, because of the framework I use and generate though a compiler, I am required to use global variable length array. However, I can not know the size of its dimension until runtime (though argv for example). For this reason, I would like to declare a global variable length array with unknown size and then define its size.

I have done it like that :

```
int (*a)[]; //global variable length array
int main(){
  //defining it's size
  a = (int(*)[2]) malloc(sizeof(int)*2*2);

  for(int i=0;i<2; i++){
     for(int j=0;j<2; j++){
        a[i][j] = i*2 + j;
     }
  }
  return 0;
}
```

However, this does not work : I get the invalid use of array with unspecified bounds error. I suspect it is because even if its size is defined, its original type does not define the size of the larger stride.

Does someone know how to solve this issue ? Using C99 (no C++) and it should be quite standard (working on gcc and icc at least).

EDIT: I may have forget something that matters. I am required to propose an array that is usable through the "static array interface", I mean by that the multiple square bracket (one per dimension).

You cannot declare a global multi dimensional VLA, because even if you use pointers, all the dimensions except for the first one must be known at declaration time.

My best attempt would be to use a global void *. In C void * is a special pointer type that can be used to store a pointer to any type, and is often used for opaque pointers.

Here you could do:

```
void *a; // opaque (global variable length array) pointer
int main() {
    //defining it's size
    a =  malloc(sizeof(int) * 2 * 2);  // the global opaque pointer
    int(*a22)[2] = a;              // a local pointer to correct type
    for (int i = 0; i<2; i++) {
```

```
        for (int j = 0; j<2; j++) {
            a22[i][j] = i * 2 + j;
        }
    }
    return 0;
}
```
When you need to access the global VLA, you assign the value of the opaque global to a local VLA pointer, and can then use it normally. You will probably have to store the dimensions in global variables too...

Structures:

Purpose and usage of structures

Structure is a group of variables of different data types represented by a single name. Lets take an example to understand the need of a structure in C programming.

Lets say we need to store the data of students like student name, age, address, id etc. One way of doing this would be creating a different variable for each attribute, however when you need to store the data of multiple students then in that case, you would need to create these several variables again for each student. This is such a big headache to store data in this way.

We can solve this problem easily by using structure. We can create a structure that has members for name, id, address and age and then we can create the variables of this structure for each student. This may sound confusing, do not worry we will understand this with the help of example.

How to create a structure in C Programming

We use **struct** keyword to create a **structure in C**. The struct keyword is a short form of **structured data type.**

```
struct struct_name {
  DataType member1_name;
  DataType member2_name;
  DataType member3_name;
  …
};
```

Here struct_name can be anything of your choice. Members data type can be same or different. Once we have declared the structure we can use the struct name as a data type like int, float etc.

How to declare variable of a structure?

```
struct  struct_name  var_name;
or
```

```
struct struct_name {
   DataType member1_name;
   DataType member2_name;
   DataType member3_name;
   …
} var_name;
```

How to access data members of a structure using a struct variable?

```
var_name.member1_name;
var_name.member2_name;
…
```

How to assign values to structure members?

There are three ways to do this.
1) Using Dot(.) operator

```
var_name.memeber_name = value;
```

2) All members assigned in one statement

```
struct struct_name var_name =
{value for memeber1, value for memeber2 …so on for all the members}
```

3) **Designated initializers** – We will discuss this later at the end of this post.

Example of Structure in C

```
#include <stdio.h>
/* Created a structure here. The name of the structure is
 * StudentData.
 */
```

```
struct StudentData{
    char *stu_name;
    int stu_id;
    int stu_age;
};
int main()
{
    /* student is the variable of structure StudentData*/
    struct StudentData student;

    /*Assigning the values of each struct member here*/
    student.stu_name = "Steve";
    student.stu_id = 1234;
    student.stu_age = 30;

    /* Displaying the values of struct members */
    printf("Student Name is: %s", student.stu_name);
    printf("\nStudent Id is: %d", student.stu_id);
    printf("\nStudent Age is: %d", student.stu_age);
    return 0;
}
```

Output:

```
Student Name is: Steve
Student Id is: 1234
Student Age is: 30
```

Nested Structure in C: Struct inside another struct

You can use a structure inside another structure, which is fairly possible. As I explained above that once you declared a structure, the **struct struct_name** acts as a new data type so you can include it in another struct just like the data type of other data members. Sounds confusing? Don't worry. The following example will clear your doubt.

Example of Nested Structure in C Programming

Lets say we have two structure like this:

Structure 1: stu_address

```
struct stu_address
{
```

```
    int street;
    char *state;
    char *city;
    char *country;
}
```
Structure 2: stu_data

```
struct stu_data
{
    int stu_id;
    int stu_age;
    char *stu_name;
    struct stu_address stuAddress;
}
```

As you can see here that I have nested a structure inside another structure.

Assignment for struct inside struct (Nested struct)

Lets take the example of the two structure that we seen above to understand the logic

```
struct  stu_data  mydata;
mydata.stu_id = 1001;
mydata.stu_age = 30;
mydata.stuAddress.state = "UP"; //Nested struct assignment
..
```

How to access nested structure members?

Using chain of "." operator.
Suppose you want to display the city alone from nested struct –

```
printf("%s",  mydata.stuAddress.city);
```
Use of typedef in Structure

typedef makes the code short and improves readability. In the above discussion we have seen that while using structs every time we have to use the lengthy syntax, which makes the code confusing, lengthy, complex and less readable. The simple solution to this issue is use of typedef. It is like an alias of struct.

Code without typedef

```
struct home_address {
```

```
  int local_street;
  char *town;
  char *my_city;
  char *my_country;
};
...
struct home_address var;
var.town = "Agra";
```

Code using tyepdef

```
typedef struct home_address{
  int local_street;
  char *town;
  char *my_city;
  char *my_country;
}addr;
..
..
addr var1;
var.town = "Agra";
```

Instead of using the struct home_address every time you need to declare struct variable, you can simply use addr, the typedef that we have defined.

Designated initializers to set values of Structure members

We have already learned two ways to set the values of a struct member, there is another way to do the same using designated initializers. This is useful when we are doing assignment of only few members of the structure. In the following example the structure variable s2 has only one member assignment.

```
#include <stdio.h>
struct numbers
{
  int num1, num2;
};
int main()
{
  // Assignment using using designated initialization
  struct numbers s1 = {.num2 = 22, .num1 = 11};
  struct numbers s2 = {.num2 = 30};
```

```
    printf ("num1: %d, num2: %d\n", s1.num1, s1.num2);
    printf ("num1: %d", s2.num2);
    return 0;
}
```

Output:

```
num1: 11, num2: 22
num1: 30
```

declaring structures

1) Struct definition: introduces the new type struct name and defines its meaning

2) If used on a line of its own, as in struct name ;, declares but doesn't define the struct name (see forward declaration below). In other contexts, names the previously-declared struct, and attr-spec-seq is not allowed.

name	-	the name of the struct that's being defined
struct-declaration-list	-	any number of variable declarations, bit field declarations, and static assert declarations. Members of incomplete type and members of function type are not allowed (except for the flexible array member described below)
attr-spec-seq	-	(C23)optional list of attributes, applied to the struct type

Explanation

Within a struct object, addresses of its elements (and the addresses of the bit field allocation units) increase in order in which the members were defined. A pointer to a struct can be cast to a pointer to its first member (or, if the member is a bit field, to its allocation unit). Likewise, a pointer to the first member of a struct can be cast to a pointer to the enclosing struct. There may be unnamed padding between any two members of a struct or after the last member, but not before the first member. The size of a struct is at least as large as the sum of the sizes of its members.

If a struct defines at least one named member, it is allowed to additionally (since C99) declare its last member with incomplete array type. When an element of the

235

flexible array member is accessed (in an expression that uses operator . or -> with the flexible array member's name as the right-hand-side operand), then the struct behaves as if the array member had the longest size fitting in the memory allocated for this object. If no additional storage was allocated, it behaves as if an array with 1 element, except that the behavior is undefined if that element is accessed or a pointer one past that element is produced. Initialization, sizeof, and the assignment operator ignore the flexible array member. Structures with flexible array members (or unions who have a recursive-possibly structure member with flexible array member) cannot appear as array elements or as members of other structures.

```
struct s { int n; double d[]; }; // s.d is a flexible array member


    struct s t1 = { 0 };        // OK, d is as if double d[1], but UB to access

    struct s t2 = { 1, { 4.2 } }; // error: initialization ignores flexible array


    // if sizeof (double) == 8

    struct s *s1 = malloc(sizeof (struct s) + 64); // as if d was double d[8]

    struct s *s2 = malloc(sizeof (struct s) + 40); // as if d was double d[5]


    s1 = malloc(sizeof (struct s) + 10); // now as if d was double d[1]. Two
bytes excess.

    double *dp = &(s1->d[0]);    // OK

    *dp = 42;                // OK

    s1->d[1]++;                  // Undefined behavior. 2 excess bytes can't be
accessed

                    // as double.


    s2 = malloc(sizeof (struct s) + 6);  // same, but UB to access because 2
```

bytes are

// missing to complete 1 double

```
  dp = &(s2->d[0]);         //  OK, can take address just fine

  *dp = 42;                 //  undefined behavior


  *s1 = *s2; // only copies s.n, not any element of s.d

          // except those caught in sizeof (struct s)
```

Similar to union, an unnamed member of a struct whose type is a struct without name is known as anonymous struct. Every member of an anonymous struct is considered to be a member of the enclosing struct or union. This applies recursively if the enclosing struct or union is also anonymous.

```
struct v {

  union { // anonymous union

    struct { int i, j; }; // anonymous structure

    struct { long k, l; } w;

  };                                                     (since C11)

  int m;

} v1;


v1.i = 2;   // valid

v1.k = 3;   // invalid: inner structure is not anonymous

v1.w.k = 5; // valid
```

Similar to union, the behavior of the program is undefined if struct is defined without any named members (including those obtained via anonymous

nested structs or unions).

Forward declaration

A declaration of the following form

---

struct attr-spec-seq(optional) name ;

---

hides any previously declared meaning for the name name in the tag name space and declares name as a new struct name in current scope, which will be defined later. Until the definition appears, this struct name has incomplete type.

This allows structs that refer to each other:

struct y;

struct x { struct y *p; /* ... */ };

struct y { struct x *q; /* ... */ };

Note that a new struct name may also be introduced just by using a struct tag within another declaration, but if a previously declared struct with the same name exists in the tag name space, the tag would refer to that name

struct s* p = NULL; // tag naming an unknown struct declares it

struct s { int a; }; // definition for the struct pointed to by p

void g(void)

{

   struct s; // forward declaration of a new, local struct s

         // this hides global struct s until the end of this block

   struct s *p; // pointer to local struct s

         // without the forward declaration above,

         // this would point at the file-scope s

   struct s { char* p; }; // definitions of the local struct s

}

Keywords

struct

Notes

See struct initialization for the rules regarding the initializers for structs.

Because members of incomplete type are not allowed, and a struct type is not complete until the end of the definition, a struct cannot have a member of its own type. A pointer to its own type is allowed, and is commonly used to implement nodes in linked lists or trees.

Because a struct declaration does not establish scope, nested types, enumerations and enumerators introduced by declarations within struct-declaration-list are visible in the surrounding scope where the struct is defined.

Example

Run this code

```
#include <stddef.h>

#include <stdio.h>


int main(void)

{

    struct car { char *make; char *model; int year; }; // declares the struct type

    // declares and initializes an object of a previously-declared struct type

    struct car c = {.year=1923, .make="Nash", .model="48 Sports Touring Car"};

    printf("car: %d %s %s\n", c.year, c.make, c.model);


    // declares a struct type, an object of that type, and a pointer to it

    struct spaceship { char *make; char *model; char *year; }

    ship = {"Incom Corporation", "T-65 X-wing starfighter", "128 ABY"},

    *pship = &ship;
```

```c
    printf("spaceship: %s %s %s\n", ship.year, ship.make, ship.model);


    // addresses increase in order of definition
    // padding may be inserted
    struct A { char a; double b; char c;};
    printf("offset of char a = %zu\noffset of double b = %zu\noffset of char c = %zu\n"
        "sizeof(struct A) = %zu\n", offsetof(struct A, a), offsetof(struct A, b),
        offsetof(struct A, c), sizeof(struct A));
    struct B { char a; char b; double c;};
    printf("offset of char a = %zu\noffset of char b = %zu\noffset of double c = %zu\n"
        "sizeof(struct B) = %zu\n", offsetof(struct B, a), offsetof(struct B, b),
        offsetof(struct B, c), sizeof(struct B));


    // A pointer to a struct can be cast to a pointer to its first member and vice versa
    char* pmake = (char*)pship;
    pship = (struct spaceship *)pmake;
}
```

Possible output:

car: 1923 Nash 48 Sports Touring Car

spaceship: 128 ABY Incom Corporation T-65 X-wing starfighter

offset of char a = 0

offset of double b = 8

offset of char c = 16

sizeof(struct A) = 24

offset of char a = 0

offset of char b = 1

offset of double c = 8

sizeof(struct B) = 16

assigning of structures

There is a structure type defined as below:

1typedef struct __map_t {

2   int code;

3   char name[NAME_SIZE];

4   char *alias;

5}map_t;

If we want to assign map_t type variable struct2 to sturct1, we usually have below 3 ways:

1 /* Way #1: assign the members one by one */

2 struct1.code = struct2.code;

3 strncpy(struct1.name, struct2.name, NAME_SIZE);

4 struct1.alias = struct2.alias;

5

6 /* Way #2: memcpy the whole memory content of struct2 to struct1 */

7 memcpy(&struct1, &struct2, sizeof(struct1));

8

9 /* Way #3: straight assignment with '=' */

10struct1 = struct2;

Consider above ways, most of programmer won't use way #1, since it's so stupid ways compare to other twos, only if we are defining an structure assignment function.
So, what's the difference between way #2 and way #3? And what's the pitfall of the

structure assignment once there is array or pointer member existed? Coming sections maybe helpful for your understanding.

The difference between '=' straight assignment and memcpy

The struct1=struct2; notation is not only more concise, but also shorter and leaves more optimization opportunities to the compiler. The semantic meaning of = is an assignment, while memcpy just copies memory. That's a huge difference in readability as well, although memcpy does the same in this case.

Copying by straight assignment is probably best, since it's shorter, easier to read, and has a higher level of abstraction. Instead of saying (to the human reader of the code) "copy these bits from here to there", and requiring the reader to think about the size argument to the copy, you're just doing a straight assignment ("copy this value from here to here"). There can be no hesitation about whether or not the size is correct.

Consider that, above source code also has pitfall about the pointer alias, it will lead dangling pointer problem (It will be introduced below section). If we use straight structure assignment '=' in C++, we can consider to overload the operator= function, that can dissolve the problem, and the structure assignment usage does not need to do any changes, but structure memcpy does not have such opportunity.

The pitfall of structure assignment:

Beware though, that copying structs that contain pointers to heap-allocated memory can be a bit dangerous, since by doing so you're aliasing the pointer, and typically making it ambiguous who owns the pointer after the copying operation.

If the structures are of compatible types, yes, you can, with something like:

1memcpy (dest_struct, source_struct, sizeof(dest_struct));

} The only thing you need to be aware of is that this is a shallow copy. In other words, if you have a char * pointing to a specific string, both structures will point to the same string.

And changing the contents of one of those string fields (the data that the char points to, not the char itself) will change the other as well. For these situations a "deep copy" is really the only choice, and that needs to go in a function. If you want a easy copy without having to manually do each field but with the added bonus of non-shallow string copies, use strdup:

1memcpy (dest_struct, source_struct, sizeof (dest_struct));

2dest_struct->strptr = strdup(source_struct->strptr);

This will copy the entire contents of the structure, then deep-copy the string, effectively giving a separate string to each structure. And, if your C implementation doesn't have a strdup (it's not part of the ISO standard), you have to allocate new memory for dest_struct pointer member, and copy the data to memory address.

Example of trap:

```
1  #include <stdio.h>

2  #include <stdlib.h>

3  #include <string.h>

4

5  #define NAME_SIZE   16

6  typedef struct _map_t {

7      int code;

8      char name[NAME_SIZE];

9      char *alias;

10} map_t;

11

12int main()

13{

14   map_t a, b, c;

15

16   /* initialize the a's members value */

17   a.code = 1024;

18   snprintf(a.name, NAME_SIZE, "Controller SW3");

19   char *alias = "RNC&IPA";

20   a.alias = alias;
```

21

22   /* assign the value via memcpy */

23   memcpy(&b, &a, sizeof(b));

24

25   /* assign the value via '=' */

26   c = a;

27   return 0;

28}

Below diagram illustrates above source memory layout, if there is a pointer field member, either the straight assignment or memcpy, that will be alias of pointer to point same address. For example, b.alias and c.alias both points to address of a.alias. Once one of them free the pointed address, it will cause another pointer as dangling pointer. It's dangerous!!

Conclusion

Recommend use straight assignment '=' instead of memcpy.

If structure has pointer or array member, please consider the pointer alias problem, it will lead dangling pointer once incorrect use. Better way is implement structure assignment function in C, and overload the operator= function in C++.

to Objects:

Pointer and address arithmetic

Pointers variables are also known as **address data types** because they are used to store the address of another variable. The address is the memory location that is assigned to the variable. It doesn't store any value.
Hence, there are only a few operations that are allowed to perform on Pointers in C language. The operations are slightly different from the ones that we generally use for mathematical calculations. The operations are:

1. Increment/Decrement of a Pointer

2. Addition of integer to a pointer

3. Subtraction of integer to a pointer

4. Subtracting two pointers of the same type

<div align="center">

Increment/Decrement of a Pointer

</div>

Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is incremented, then it will increment by 2(**size of an int**) and the new address it will points to **1002**. While if a float type pointer is incremented then it will increment by 4(**size of a float**) and the new address will be **1004**.
Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is decremented, then it will decrement by

2(**size of an int**) and the new address it will points to **998**. While if a float type pointer is decremented then it will decrement by 4(**size of a float**) and the new address will be **996**. Below is the program to illustrate pointer increment/decrement:

filter_none
edit
play_arrow
brightness_4

```
// C program to illustrate
// pointer increment/decrement

#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores
    // the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr1 "
        "before Increment: ");
    printf("%p \n", ptr1);

    // Incrementing pointer ptr1;
    ptr1++;

    printf("Pointer ptr1 after"
        " Increment: ");
    printf("%p \n\n", ptr1);

    printf("Pointer ptr1 before"
        " Decrement: ");
    printf("%p \n", ptr1);

    // Decrementing pointer ptr1;
    ptr1--;
```

```
    printf("Pointer ptr1 after"
        " Decrement: ");
    printf("%p \n\n", ptr1);

    return 0;
}
```

Output:

Pointer ptr1 before Increment: 0x7ffcb19385e4

Pointer ptr1 after Increment: 0x7ffcb19385e8

Pointer ptr1 before Decrement: 0x7ffcb19385e8

Pointer ptr1 after Decrement: 0x7ffcb19385e4

Addition

When a pointer is added with a value, the value is first multiplied by the size of data type and then added to the pointer.

filter_none
edit
play_arrow
brightness_4

```
// C program to illustrate pointer Addition
#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Addition: ");
    printf("%p \n", ptr2);
```

```
    // Addition of 3 to ptr2
    ptr2 = ptr2 + 3;
    printf("Pointer ptr2 after Addition: ");
    printf("%p \n", ptr2);

    return 0;
}
```

Output:

Pointer ptr2 before Addition: 0x7fffffdcd984

Pointer ptr2 after Addition: 0x7fffffdcd990

Subtraction

When a pointer is subtracted with a value, the value is first multiplied by the size of the data type and then subtracted from the pointer.

Below is the program to illustrate pointer Subtraction:

filter_none
edit
play_arrow
brightness_4

```
// C program to illustrate pointer Subtraction
#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Subtraction: ");
    printf("%p \n", ptr2);

    // Subtraction of 3 to ptr2
    ptr2 = ptr2 - 3;
    printf("Pointer ptr2 after Subtraction: ");
```

```
    printf("%p \n", ptr2);

    return 0;
}
```

Output:

Pointer ptr2 before Subtraction: 0x7ffcf1221b24

Pointer ptr2 after Subtraction: 0x7ffcf1221b18

Subtraction of Two Pointers

The subtraction of two pointers is possible only when they have the same data type. The result is generated by calculating the difference between the addresses of the two pointers and calculating how many bits of data it is according to the pointer data type. The subtraction of two pointers gives the increments between the two pointers.

For Example:

Two integer pointers say **ptr1(address:1000)** and **ptr2(address:1016)** are subtracted. The difference between address is 16 bytes. Since the size of int is 2 bytes, therefore the **increment between ptr1 and ptr2** is given by **(16/2) = 8**.
Below is the implementation to illustrate the Subtraction of Two Pointers:

filter_none
edit
play_arrow
brightness_4

```
// C program to illustrate Subtraction
// of two pointers
#include <stdio.h>

// Driver Code
int main()
{
    int x;

    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
```

```c
    ptr1 = &N;
    ptr2 = &N;

    // Incrementing ptr2 by 3
    ptr2 = ptr2 + 3;

    // Subtraction of ptr2 and ptr1
    x = ptr2 - ptr1;

    // Print x to get the Increment
    // between ptr1 and ptr2
    printf("Subtraction of ptr1 "
        "& ptr2 is %d\n",
        x);

    return 0;
}
```

Output:

Subtraction of ptr1 & ptr2 is 3

Pointer Arthemetic on Arrays:

Pointers contain addresses. Adding two addresses makes no sense because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between the two addresses. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. **For Example:** if an array named arr then arr and &arr[0] can be used to reference array as a pointer.
Below is the program to illustrate the Pointer Arithmetic on arrays:

Program 1:

```c
filter_none
edit
play_arrow
brightness_4
// C program to illustrate the array
// traversal using pointers
#include <stdio.h>

// Driver Code
int main()
{

    int N = 5;
```

```c
    // An array
    int arr[] = { 1, 2, 3, 4, 5 };

    // Declare pointer variable
    int* ptr;

    // Point the pointer to first
    // element in array arr[]
    ptr = arr;

    // Traverse array using ptr
    for (int i = 0; i < N; i++) {

        // Print element at which
        // ptr points
        printf("%d ", ptr[0]);
        ptr++;
    }
}
```

Output:

1 2 3 4 5

Program 2:

filter_none
edit
play_arrow
brightness_4

```c
// C program to illustrate the array
// traversal using pointers in 2D array
#include <stdio.h>

// Function to traverse 2D array
// using pointers
void traverseArr(int* arr,
          int N, int M)
{

    int i, j;

    // Travere rows of 2D matrix
    for (i = 0; i < N; i++) {
```

```c
        // Travere columns of 2D matrix
        for (j = 0; j < M; j++) {

            // Print the element
            printf("%d ", *((arr + i * M) + j));
        }
        printf("\n");
    }
}

// Driver Code
int main()
{

    int N = 3, M = 2;

    // A 2D array
    int arr[][2] = { { 1, 2 },
                { 3, 4 },
                { 5, 6 } };

    // Function Call
    traverseArr((int*)arr, N, M);
    return 0;
}
```

Output:

```
1 2

3 4

5 6
```

pointer operations and declarations

Pointers are the special type of data types which stores memory address (reference) of another variable. Here we will learn how to declare and initialize a pointer variable with the address of another variable?

Pointer Declarations

Pointer declaration is similar to other type of variable except asterisk (*) character before pointer variable name.

Here is the syntax to declare a pointer

data_type *poiter_name;

Let's consider with following example statement

int *ptr;

Here, in this statement

- ptr is the name of pointer variable (name of the memory blocks in which address of another variable is going to be stored).

- The character asterisk (*) tells to the compiler that the identifier ptr should be declare as pointer.

- The data type int tells to the compiler that pointer ptr will store memory address of integer type variable.

Finally, ptr will be declared as integer pointer which will store address of integer type variable.

Pointer ptr is declared, but it not pointing to anything; now pointer should be initialized by the address of another integer variable.

Consider the following statement of pointer initialization

```
int x;
int *ptr;
ptr=&x;
```

Here, x is an integer variable and pointer ptr is initiating with the address of x.

Accessing address and value of x using pointer variable ptr

We can get the value of ptr which is the address of x (an integer variable)

- ptr will print the stored value (memory address of x).

- *ptr will print the value which is stored at the containing memory address in the ptr (value of variable x).

Here is the simple example to demonstrate pointer declaration, initialization and accessing address, value through pointer variable:

```
#include <stdio.h>
int main()
{
        int x=20;           //int variable
        int *ptr;  //int pointer declaration

        ptr=&x;             //initializing pointer

        printf("Memory address of x: %p\n",ptr);
        printf("Value x: %d\n",*ptr);

        return 0;
}
```

```
Memory address of x: 0x7ffe64f5c814
Value x: 20
```

using pointers as function arguments

In this tutorial, you will learn how to pass a pointer to a function as an argument. To understand this concept you must have a basic idea of Pointers and functions in C programming.

Just like any other argument, pointers can also be passed to a function as an argument. Lets take an example to understand how this is done.

Example: Passing Pointer to a Function in C Programming

In this example, we are passing a pointer to a function. When we pass a pointer as an argument instead of a variable then the address of the variable is passed instead of the value. So any change made by the function using the pointer is permanently made at the address of passed variable. This technique is known as call by reference in C.

Try this same program without pointer, you would find that the bonus amount will not reflect in the salary, this is because the change made by the function would be done to the local variables of the function. When we use pointers, the value is changed at the address of variable

```
#include <stdio.h>
void salaryhike(int  *var, int b)
{
   *var = *var+b;
}
```

```c
int main()
{
   int salary=0, bonus=0;
   printf("Enter the employee current salary:");
   scanf("%d", &salary);
   printf("Enter bonus:");
   scanf("%d", &bonus);
   salaryhike(&salary, bonus);
   printf("Final salary: %d", salary);
   return 0;
}
```

Output:

```
Enter the employee current salary:10000
Enter bonus:2000
Final salary: 12000
```

Example 2: Swapping two numbers using Pointers

This is one of the most popular example that shows how to swap numbers using call by reference.

Try this program without pointers, you would see that the numbers are not swapped. The reason is same that we have seen above in the first example.

```c
#include <stdio.h>
void swapnum(int *num1, int *num2)
{
   int tempnum;

   tempnum = *num1;
   *num1 = *num2;
   *num2 = tempnum;
}
int main( )
{
   int v1 = 11, v2 = 77 ;
   printf("Before swapping:");
   printf("\nValue of v1 is: %d", v1);
   printf("\nValue of v2 is: %d", v2);

   /*calling swap function*/
```

```
    swapnum( &v1, &v2 );

    printf("\nAfter swapping:");
    printf("\nValue of v1 is: %d", v1);
    printf("\nValue of v2 is: %d", v2);
}
```

Output:

```
Before swapping:
Value of v1 is: 11
Value of v2 is: 77
After swapping:
Value of v1 is: 77
Value of v2 is: 11
```

Dynamic memory allocation

The concept of **dynamic memory allocation in c language** enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

1. ptr=(cast-type*)malloc(byte-size)

Let's see the example of malloc() function.

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  int main(){
4.    int n,i,*ptr,sum=0;
5.      printf("Enter number of elements: ");
6.      scanf("%d",&n);
7.      ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
8.      if(ptr==NULL)
9.      {
10.        printf("Sorry! unable to allocate memory");
11.        exit(0);
12.      }
13.      printf("Enter elements of array: ");
14.      for(i=0;i<n;++i)
15.      {
16.        scanf("%d",ptr+i);
17.        sum+=*(ptr+i);
18.      }
19.      printf("Sum=%d",sum);
20.      free(ptr);
21.  return 0;
22.}
```

Output

Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30

calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

1. ptr=(cast-type*)calloc(number, byte-size)

Let's see the example of calloc() function.

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  int main(){
4.   int n,i,*ptr,sum=0;
5.      printf("Enter number of elements: ");
6.      scanf("%d",&n);
7.      ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
8.      if(ptr==NULL)
9.      {
10.        printf("Sorry! unable to allocate memory");
11.        exit(0);
12.     }
13.     printf("Enter elements of array: ");
14.     for(i=0;i<n;++i)
15.     {
16.        scanf("%d",ptr+i);
17.        sum+=*(ptr+i);
18.     }
19.     printf("Sum=%d",sum);
20.     free(ptr);
21.  return 0;
22.}
```

Output

Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

1. ptr=realloc(ptr, **new**-size)

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

1. free(ptr)

defining and using stacks and linked lists

Stack as we know is a **Last In First Out(LIFO)** data structure. It has the following operations :

- **push:** push an element into the stack

- **pop:** remove the last element added

- **top:** returns the element at top of stack
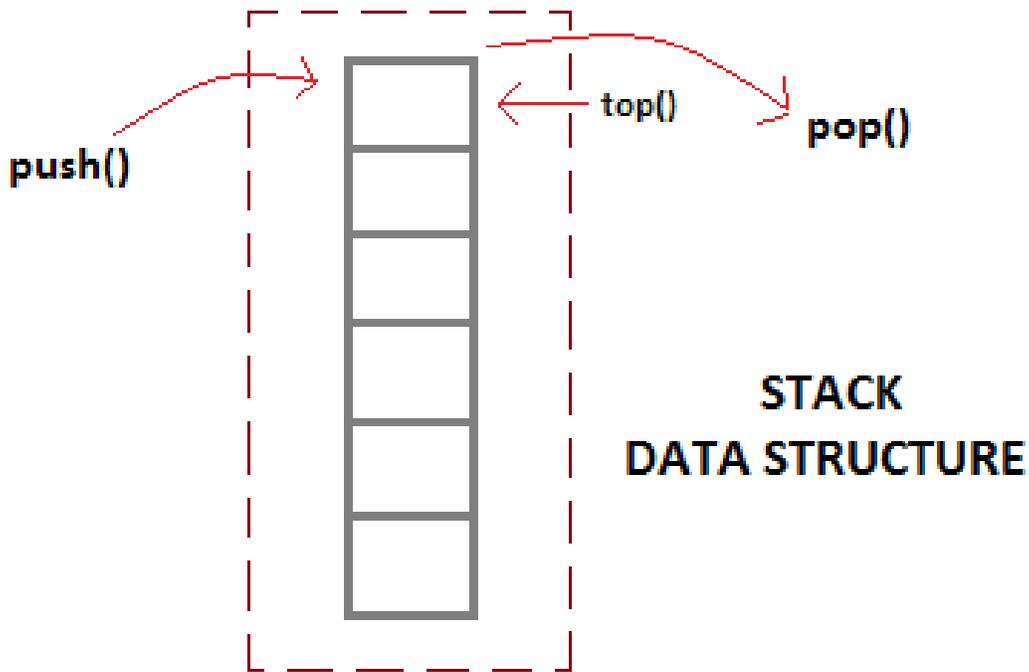
Implementation of Stack using Linked List

Stacks can be easily implemented using a linked list. Stack is a data structure to which a data can be added using the push() method and data can be removed from it using the pop() method. With Linked list, the **push** operation can be replaced by the addAtFront() method of linked list and **pop** operation can be replaced by a function which deletes the front node of the linked list.

In this way our Linked list will virtually become a Stack with push() and pop() methods.

First we create a class **node**. This is our Linked list node class which will have **data** in it and a **node pointer** to store the address of the next node element.

```
class node
{
        int data;
        node *next;
};
```

Then we define our stack class,

```
class Stack

{

        node *front;  // points to the head of list

        public:

        Stack()

        {

                front = NULL;

        }

        // push method to add data element

        void push(int);

        // pop method to remove data element

        void pop();

        // top method to return top data element

        int top();

};
```

Inserting Data in Stack (Linked List)

In order to insert an element into the stack, we will create a node and place it in front of the list.

```
void Stack :: push(int d)

{

        // creating a new node

        node *temp;

        temp = new node();
```

```
        // setting data to it

        temp->data = d;



        // add the node in front of list

        if(front == NULL)

        {

                temp->next = NULL;

        }

        else

        {

                temp->next = front;

        }

        front = temp;

}
```

Now whenever we will call the push() function a new node will get added to our list in the front, which is exactly how a stack behaves.

Removing Element from Stack (Linked List)

In order to do this, we will simply delete the first node, and make the second node, the head of the list.

```
void Stack :: pop()

{

        // if empty

        if(front == NULL)

                cout << "UNDERFLOW\n";


        // delete the first element
```

```
        else

        {

                node *temp = front;

                front = front->next;

                delete(temp);

        }

}
```

Return Top of Stack (Linked List)

In this, we simply return the data stored in the head of the list.

```
int Stack :: top()

{

        return front->data;

}
```

Conclusion

When we say "implementing Stack using Linked List", we mean how we can make a Linked List behave like a Stack, after all they are all logical entities. So for any data structure to act as a Stack, it should have push() method to add data on top and pop() method to remove data from top. Which is exactly what we did and hence accomplished to make a Linked List behave as a Stack.

UNIT V

Sequential search

When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship. Each data item is stored in a position relative to the others. In Python lists, these relative positions are the index values of the individual items. Since these index values are ordered, it is possible for us to visit them in sequence. This process gives rise to our first searching technique, the **sequential search**.

The diagram below shows how this search works. Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items. If we run out of items, we have discovered that the item we were searching for was not present.
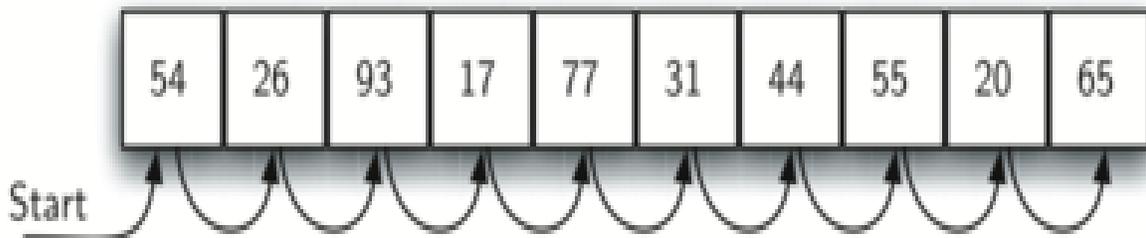
Sequential search of a list of integers

The Python implementation for this algorithm is shown below. The function needs the list and the item we are looking for and returns a boolean value as to whether it is present. Remember in practice we would use the Python in operator for this purpose, so you can think of the below algorithm as what we would do if in were not provided for us.

```python
def sequential_search(alist, item):
    position = 0

    while position < len(alist):
        if alist[position] == item:
            return True
        position = position + 1

    return False

testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]

sequential_search(testlist, 3)  # => False
```

sequential_search(testlist, 13) # => True

Analysis of Sequential Search

To analyze searching algorithms, we need to decide on a basic unit of computation. Recall that this is typically the common step that must be repeated in order to solve the problem. For searching, it makes sense to count the number of comparisons performed. Each comparison may or may not discover the item we are looking for. In addition, we make another assumption here. The list of items is not ordered in any way. The items have been placed randomly into the list. In other words, the probability that the item we are looking for is in any particular position is exactly the same for each position of the list.

If the item is not in the list, the only way to know it is to compare it against every item present. If there are n items, then the sequential search requires n comparisons to discover that the item is not there. In the case where the item is in the list, the analysis is not so straightforward. There are actually three different scenarios that can occur. In the best case we will find the item in the first place we look, at the beginning of the list. We will need only one comparison. In the worst case, we will not discover the item until the very last comparison, the nth comparison.

What about the average case? On average, we will find the item about halfway into the list; that is, we will compare against $\frac{n}{2}$ items. Recall, however, that as n gets large, the coefficients, no matter what they are, become insignificant in our approximation, so the complexity of the sequential search, is $O(n)$:

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

We assumed earlier that the items in our collection had been randomly placed so that there is no relative order between the items. What would happen to the sequential search if the items were ordered in some way? Would we be able to gain any efficiency in our search technique?

Assume that the list of items was constructed so that the items were in ascending order, from low to high. If the item we are looking for is present in the list, the chance of it being in any one of the n positions is still the same as before. We will still have the same number of comparisons to find the item. However, if the item is not present there is a slight advantage. The diagram below shows this process as the algorithm looks for the item 50. Notice that items are still compared in sequence until 54. At this point,

however, we know something extra. Not only is 54 not the item we are looking for, but no other elements beyond 54 can work either since the list is sorted.
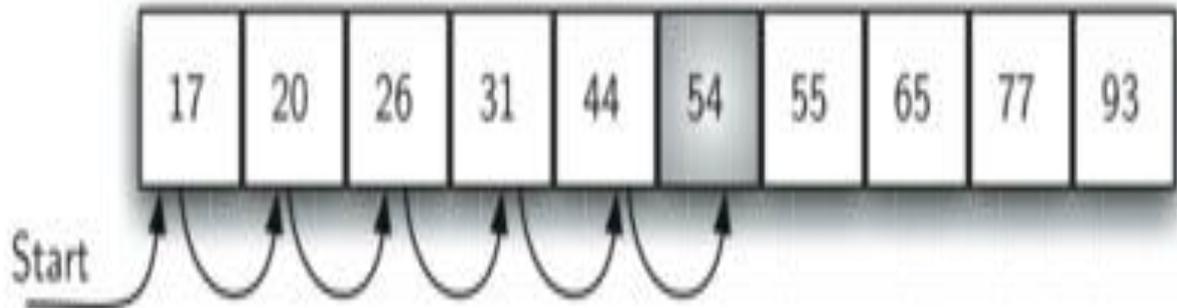
Sequential search of an ordered list of integers

In this case, the algorithm does not have to continue looking through all of the items to report that the item was not found. It can stop immediately. The code below shows this variation of the sequential search function.

```
def ordered_sequential_search(alist, item):
    position = 0

    while position < len(alist):
        if alist[position] == item:
            return True

        if alist[position] > item:
            return False

        position = position + 1

    return False
```

```
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
ordered_sequential_search(testlist, 3)  # => False
ordered_sequential_search(testlist, 13)  # => True
```

The table below summarizes these results. Note that in the best case we might discover that the item is not in the list by looking at only one item. On average, we will know after looking through only \frac {n}{2}2n items. However, this technique is still O(n)O(n). In summary, a sequential search is improved by ordering the list only in the case where we do not find the item.

Case	Best Case	Worst Case	Average Case
item is present	11	nn	$\frac{n}{2}2n$
item is not present	nn	nn	$\frac{n}{2}2n$

Sorting arrays

sort() method is a java.util.Arrays class method.
Syntax:

public static void **sort**(int[] arr, int from_Index, int to_Index)

arr - the array to be sorted
from_Index - the index of the first element, inclusive, to be sorted
to_Index - the index of the last element, exclusive, to be sorted

This method doesn't return any value.

A Java program to **sort an array of integers in ascending order**.

* Java
  filter_none
  edit
  play_arrow
  brightness_4

```java
// A sample Java program to sort an array of integers
// using Arrays.sort(). It by default sorts in
// ascending order
import java.util.Arrays;

public class SortExample
{
    public static void main(String[] args)
    {
        // Our arr contains 8 elements
        int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};

        Arrays.sort(arr);
```

```
        System.out.printf("Modified arr[] : %s",
                Arrays.toString(arr));
    }
}
```
Output:

Modified arr[] : [6, 7, 9, 13, 21, 45, 101, 102]

We can also use sort() to sort a subarray of arr[]

- Java
  filter_none
  edit
  play_arrow
  brightness_4

```
// A sample Java program to sort a subarray
// using Arrays.sort().
import java.util.Arrays;

public class SortExample
{
    public static void main(String[] args)
    {
        // Our arr contains 8 elements
        int[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

        // Sort subarray from index 1 to 4, i.e.,
        // only sort subarray {7, 6, 45, 21} and
        // keep other elements as it is.
        Arrays.sort(arr, 1, 5);

        System.out.printf("Modified arr[] : %s",
                Arrays.toString(arr));
    }
}
```
Output:

Modified arr[] : [13, 6, 7, 21, 45, 9, 2, 100]

We can also sort in descending order.

- Java
  filter_none
  edit
  play_arrow
  brightness_4

```java
// A sample Java program to sort a subarray
// in descending order using Arrays.sort().
import java.util.Arrays;
import java.util.Collections;

public class SortExample
{
    public static void main(String[] args)
    {
        // Note that we have Integer here instead of
        // int[] as Collections.reverseOrder doesn't
        // work for primitive types.
        Integer[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

        // Sorts arr[] in descending order
        Arrays.sort(arr, Collections.reverseOrder());

        System.out.printf("Modified arr[] : %s",
                    Arrays.toString(arr));
    }
}
```
  Output:

Modified arr[] : [100, 45, 21, 13, 9, 7, 6, 2]

We can also sort strings in alphabetical order.

- Java
  filter_none
  edit
  play_arrow
  brightness_4

```java
// A sample Java program to sort an array of strings
// in ascending and descending orders using Arrays.sort().
import java.util.Arrays;
import java.util.Collections;

public class SortExample
{
```

```java
    public static void main(String[] args)
    {
        String arr[] = {"practice.geeksforgeeks.org",
                    "quiz.geeksforgeeks.org",
                    "code.geeksforgeeks.org"
                    };

        // Sorts arr[] in ascending order
        Arrays.sort(arr);
        System.out.printf("Modified arr[] : \n%s\n\n",
                    Arrays.toString(arr));

        // Sorts arr[] in descending order
        Arrays.sort(arr, Collections.reverseOrder());

        System.out.printf("Modified arr[] : \n%s\n\n",
                    Arrays.toString(arr));
    }
}
```
Output:

Modified arr[] :

Modified arr[] :

[quiz.geeksforgeeks.org, practice.geeksforgeeks.org, code.geeksforgeeks.org]

We can also sort an array according to user defined criteria.
We use Comparator interface for this purpose. Below is an example.

- Java
  filter_none
  edit
  play_arrow
  brightness_4

```java
// Java program to demonstrate working of Comparator
// interface
import java.util.*;
import java.lang.*;
import java.io.*;
```

```java
// A class to represent a student.
class Student
{
    int rollno;
    String name, address;

    // Constructor
    public Student(int rollno, String name,
                        String address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name +
                    " " + this.address;
    }
}

class Sortbyroll implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}

// Driver class
class Main
{
    public static void main (String[] args)
    {
        Student [] arr = {new Student(111, "bbbb", "london"),
                    new Student(131, "aaaa", "nyc"),
                    new Student(121, "cccc", "jaipur")};

        System.out.println("Unsorted");
        for (int i=0; i<arr.length; i++)
            System.out.println(arr[i]);
```

```
        Arrays.sort(arr, new Sortbyroll());

        System.out.println("\nSorted by rollno");
        for (int i=0; i<arr.length; i++)
            System.out.println(arr[i]);
    }
}
```
Output:

Unsorted

111 bbbb london

131 aaaa nyc

121 cccc jaipur

Sorted by rollno

111 bbbb london

121 cccc jaipur

131 aaaa nyc

Strings

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C/C++ −

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string −

Live Demo

```
#include <stdio.h>

int main () {

   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   printf("Greeting message: %s\n", greeting );
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings −

Sr.No.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);**

	Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions –

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

int main () {

   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   int  len ;

   /* copy str1 into str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );

   /* total lenghth of str1 after concatenation */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );

   return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result −

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

Text files

A text file is a computer file that only contains text and has no special formatting such as bold text, italic text, images, etc. With Microsoft Windows computers text files are identified with the .txt file extension, as shown in the example picture.

An example of a text file and ASCII art is shown in the Kirk text file. You can click this link to open the .txt file in your browser or right-click the file to save the text file to your computer.

How to open a text file?

A text file can be opened in any text editor or word processor. For example, in Microsoft Windows, you can use the Notepad program to open, view, and edit text files.

The Standard C Preprocessor:

Defining and calling macros

In a C program, all lines that start with # are processed by preprocessor which is a special program invoked by the compiler. In a very basic term, preprocessor takes a C program and produces another C program without any #.
The following are some interesting facts about preprocessors in C.

1) When we use include directive, the contents of included header file (after preprocessing) are copied to the current file.
Angular brackets < and > instruct the preprocessor to look in the standard folder where all header files are held. Double quotes " and " instruct the preprocessor to look into the current folder (current directory).

2) When we use define for a constant, the preprocessor produces a C program where the defined constant is searched and matching tokens are replaced with the given expression. For example in the following program max is defined as 100.

- C

filter_none
edit
play_arrow
brightness_4

```c
#include<stdio.h>
#define max 100
int main()
{
    printf("max is %d", max);
    return 0;
}
```
Output:

max is 100

3) The macros can take function like arguments, the arguments are not checked for data type. For example, the following macro INCREMENT(x) can be used for x of any data type.

- C

filter_none
edit
play_arrow
brightness_4

```c
#include <stdio.h>
#define INCREMENT(x) ++x
int main()
{
    char *ptr = "GeeksQuiz";
    int x = 10;
```

```c
    printf("%s  ", INCREMENT(ptr));
    printf("%d", INCREMENT(x));
    return 0;
}
```

Output:

eeksQuiz 11

4) The macro arguments are not evaluated before macro expansion. For example, consider the following program

- C
filter_none
edit
play_arrow
brightness_4

```c
#include <stdio.h>
#define MULTIPLY(a, b) a*b
int main()
{
    // The macro is expanded as 2 + 3 * 3 + 5, not as 5*8
    printf("%d", MULTIPLY(2+3, 3+5));
    return 0;
}
// Output: 16
```

Output:

16
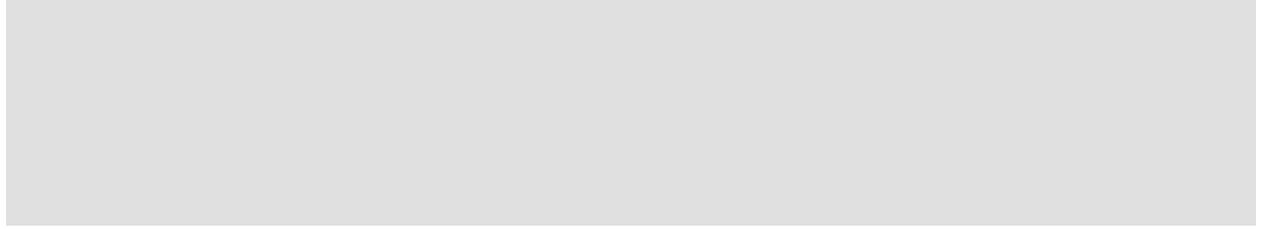
The previous problem can be solved using following program

- C
filter_none
edit
play_arrow
brightness_4

```
#include <stdio.h>
//here, instead of writing a*a we write (a)*(b)
#define MULTIPLY(a, b) (a)*(b)
int main()
{
    // The macro is expanded as (2 + 3) * (3 + 5), as 5*8
    printf("%d", MULTIPLY(2+3, 3+5));
    return 0;
}
// This code is contributed by Santanu
```
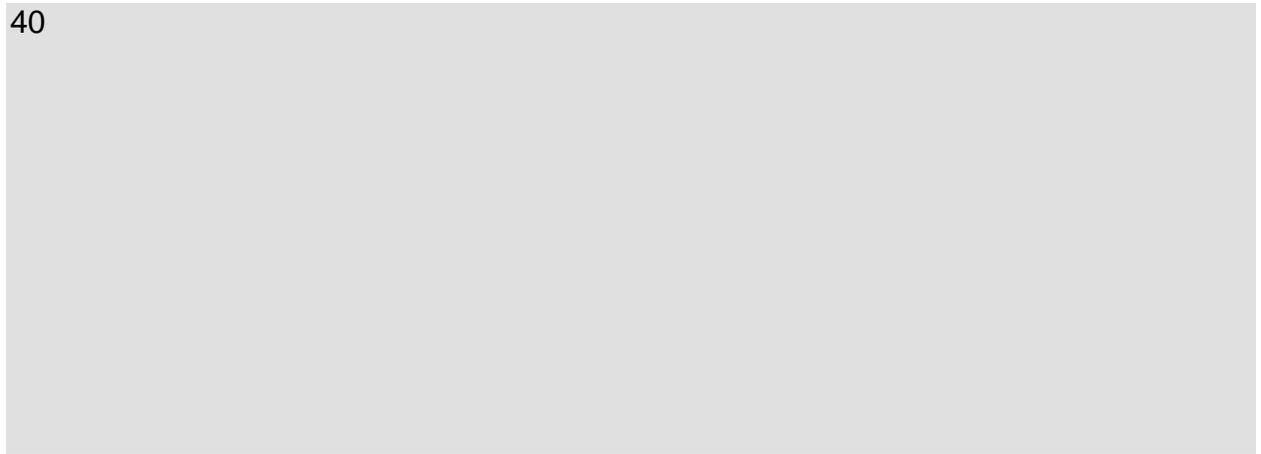
Output:

```
40
```

5) The tokens passed to macros can be concatenated using operator ## called Token-Pasting operator.

- C

filter_none
edit
play_arrow
brightness_4

```c
#include <stdio.h>
#define merge(a, b) a##b
int main()
{
    printf("%d ", merge(12, 34));
}
```
Output:

```
1234
```

6) A token passed to macro can be converted to a string literal by using # before it.

- C
filter_none
edit
play_arrow
brightness_4

```c
#include <stdio.h>
#define get(a) #a
int main()
{
    // GeeksQuiz is changed to "GeeksQuiz"
    printf("%s", get(GeeksQuiz));
}
```

Output:

GeeksQuiz

7) The macros can be written in multiple lines using '\'. The last line doesn't need to have '\'.

- C
filter_none
edit
play_arrow
brightness_4

```c
#include <stdio.h>
#define PRINT(i, limit) while (i < limit) \
                        { \
                            printf("GeeksQuiz "); \
                            i++; \
                        }
int main()
{
    int i = 0;
    PRINT(i, 3);
    return 0;
}
```

Output:

GeeksQuiz GeeksQuiz GeeksQuiz

8) The macros with arguments should be avoided as they cause problems sometimes. And Inline functions should be preferred as there is type checking parameter evaluation in inline functions. From C99 onward, inline functions are supported by C language also. For example consider the following program. From first look the output seems to be 1, but it produces 36 as output.

- C
filter_none
edit
play_arrow
brightness_4

```c
#include <stdio.h>

#define square(x) x*x
int main()
{
    // Expanded as 36/6*6
    int x = 36/square(6);
    printf("%d", x);
    return 0;
}
```

Output:

36

If we use inline functions, we get the expected output. Also, the program given in point 4 above can be corrected using inline functions.

- C
  filter_none
  edit
  play_arrow
  brightness_4

  ```c
  #include <stdio.h>

  static inline int square(int x) { return x*x; }
  int main()
  {
  int x = 36/square(6);
  printf("%d", x);
  return 0;
  }
  ```

  Output:

  ```
  1
  ```

9) Preprocessors also support if-else directives which are typically used for conditional compilation.

- C
  filter_none
  edit
  play_arrow
  brightness_4

  ```c
  int main()
  ```

```
{
#if VERBOSE >= 2
  printf("Trace Message");
#endif
}
```

Output:

No Output

10) A header file may be included more than one time directly or indirectly, this leads to problems of redeclaration of same variables/functions. To avoid this problem, directives like defined, ifdef and ifndef are used.
11) There are some standard macros which can be used to print program file (__FILE__), Date of compilation (__DATE__), Time of compilation (__TIME__) and Line Number in C code (__LINE__)

- C
filter_none
edit
play_arrow
brightness_4

```c
#include <stdio.h>

int main()
{
  printf("Current File :%s\n", __FILE__ );
  printf("Current Date :%s\n", __DATE__ );
  printf("Current Time :%s\n", __TIME__ );
  printf("Line Number :%d\n", __LINE__ );
  return 0;
}
```
Output:

Current File
:/usr/share/IDE_PROGRAMS/C/other/081c548d50135ed88cfa0296159b05ca/081c548d
50135ed88cfa0296159b05ca.c

Current Date :Sep 4 2019

Current Time :10:17:43

Line Number :8

12) We can remove already defined macros using :
#undef MACRO_NAME

- C
  filter_none
  edit
  play_arrow
  brightness_4

```c
#include <stdio.h>
#define LIMIT 100
int main()
{
   printf("%d",LIMIT);
   //removing defined macro LIMIT
   #undef LIMIT
   //Next line causes error as LIMIT is not defined
   printf("%d",LIMIT);
   return 0;
}
```
//This code is contributed by Santanu
Following program is executed correctly as we have declared LIMIT as an integer
variable after removing previously defined macro LIMIT

- C
  filter_none
  edit
  play_arrow
  brightness_4

```c
#include <stdio.h>
#define LIMIT 1000
int main()
{
   printf("%d",LIMIT);
   //removing defined macro LIMIT
   #undef LIMIT
   //Declare LIMIT as integer again
   int LIMIT=1001;
   printf("\n%d",LIMIT);
   return 0;
}
```

Output:

```
1000
1001
```

Another interesting fact about macro using (#undef)

- C
  filter_none
  edit
  play_arrow
  brightness_4

```c
#include <stdio.h>
//div function prototype
```

```c
float div(float, float);
#define div(x, y) x/y

int main()
{
//use of macro div
//Note: %0.2f for taking two decimal value after point
printf("%0.2f",div(10.0,5.0));
//removing defined macro div
#undef div
//function div is called as macro definition is removed
printf("\n%0.2f",div(10.0,5.0));
return 0;
}

//div function definition
float div(float x, float y){
return y/x;
}
//This code is contributed by Santanu
```

Output:

```
2.00
0.50
```

utilizing conditional compilation

The last preprocessor directive we're going to look at is #ifdef. If you have the sequence

```
#ifdef name
program text
#else
```

```
     more program text
     #endif
```

in your program, the code that gets compiled depends on whether a preprocessor macro by that name is defined or not. If it is (that is, if there has been a #define line for a macro called name), then ``program text'' is compiled and ``more program text'' is ignored. If the macro is not defined, ``more program text'' is compiled and ``program text'' is ignored. This looks a lot like an if statement, but it behaves completely differently: an if statement controls which statements of your program are executed at run time, but #ifdef controls which parts of your program actually get compiled.

Just as for the if statement, the #else in an #ifdef is optional. There is a companion directive #ifndef, which compiles code if the macro is not defined (although the ``#else clause'' of an #ifndef directive will then be compiled if the macro is defined). There is also an #if directive which compiles code depending on whether a compile-time expression is true or false. (The expressions which are allowed in an #if directive are somewhat restricted, however, so we won't talk much about #if here.)

Conditional compilation is useful in two general classes of situations:

- You are trying to write a portable program, but the way you do something is different depending on what compiler, operating system, or computer you're using. You place different versions of your code, one for each situation, between suitable #ifdef directives, and when you compile the progam in a particular environment, you arrange to have the macro names defined which select the variants you need in that environment. (For this reason, compilers usually have ways of letting you define macros from the invocation command line or in a configuration file, and many also predefine certain macro names related to the operating system, processor, or compiler in use. That way, you don't have to change the code to change the #define lines each time you compile it in a different environment.)

  For example, in ANSI C, the function to delete a file is remove. On older Unix systems, however, the function was called unlink. So if filename is a variable containing the name of a file you want to delete, and if you want to be able to compile the program under these older Unix systems, you might write

- #ifdef unix
-    unlink(filename);
- #else
-    remove(filename);
- #endif

Then, you could place the line

   #define unix

at the top of the file when compiling under an old Unix system. (Since all you're using the macro unix for is to control the #ifdef, you don't need to give it any replacement text at all. Any definition for a macro, even if the replacement text is empty, causes an #ifdef to succeed.)

(In fact, in this example, you wouldn't even need to define the macro unix at all, because C compilers on old Unix systems tend to predefine it for you, precisely so you can make tests like these.)

- You want to compile several different versions of your program, with different features present in the different versions. You bracket the code for each feature with #ifdef directives, and (as for the previous case) arrange to have the right macros defined or not to build the version you want to build at any given time. This way, you can build the several different versions from the same source code. (One common example is whether you turn debugging statements on or off. You can bracket each debugging printout with #ifdef DEBUG and #endif, and then turn on debugging only when you need it.)

   For example, you might use lines like this:

- #ifdef DEBUG
- printf("x is %d\n", x);
- #endif

   to print out the value of the variable x at some point in your program to see if it's what you expect. To enable debugging printouts, you insert the line

   #define DEBUG

   at the top of the file, and to turn them off, you delete that line, but the debugging printouts quietly remain in your code, temporarily deactivated, but ready to reactivate if you find yourself needing them again later. (Also, instead of inserting and deleting the #define line, you might use a compiler flag such as -DDEBUG to define the macro DEBUG from the compiler invocatin line.)

Conditional compilation can be very handy, but it can also get out of hand. When large chunks of the program are completely different depending on, say, what operating system the program is being compiled for, it's often better to place the different versions in separate source files, and then only use one of the files (corresponding to one of the versions) to build the program on any given system. Also, if you are using an ANSI Standard compiler and you are writing ANSI-compatible code, you usually won't need

so much conditional compilation, because the Standard specifies exactly how the compiler must do certain things, and exactly which library functions it much provide, so you don't have to work so hard to accommodate the old variations among compilers and libraries.

passing values to the compiler

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case A is called the **"caller function"** and B is called the **"called function or callee function"**. Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Terminology

- **Formal Parameter :** A variable and its type as they appear in the prototype of the function or method.

- **Actual Parameter :** The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

- **Modes:**

    - **IN:** Passes info from caller to calle.
    - **OUT:** Callee writes values in caller.
    - **IN/OUT:** Caller tells callee value of variable, which may be updated by callee.

Important methods of Parameter Passing

1. **Pass By Value :** This method uses in-mode semantics. Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as **call by value**.
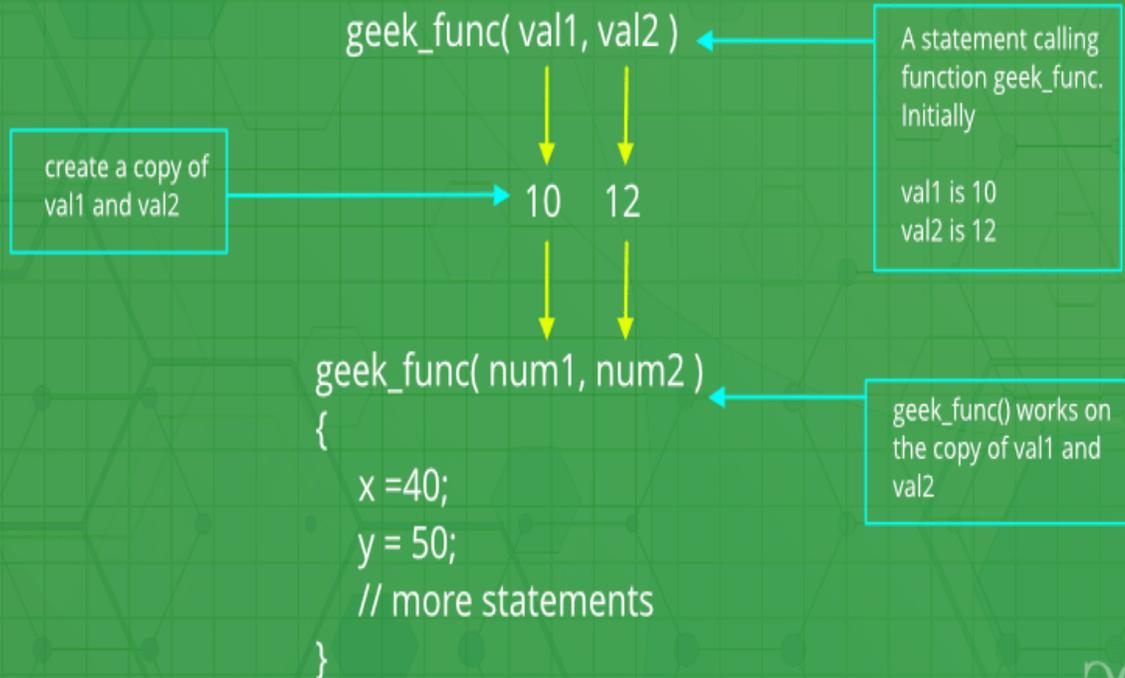
filter_none
edit
play_arrow
brightness_4

```c
// C program to illustrate
// call by value
#include <stdio.h>

void func(int a, int b)
{
    a += b;
    printf("In func, a = %d b = %d\n", a, b);
}
int main(void)
{
    int x = 5, y = 7;

    // Passing parameters
    func(x, y);
```

```
    printf("In main, x = %d y = %d\n", x, y);
    return 0;
}
```
Output:

In func, a = 12 b = 7

In main, x = 5 y = 7

Languages like C, C++, Java support this type of parameter passing. Java in fact is strictly call by value.

Shortcomings:

- Inefficiency in storage allocation
- For objects and arrays, the copy semantics are costly

2. **Pass by reference(aliasing) :** This technique uses in/out-mode semantics. Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as **<em>call by reference**. This method is efficient in both time and space.
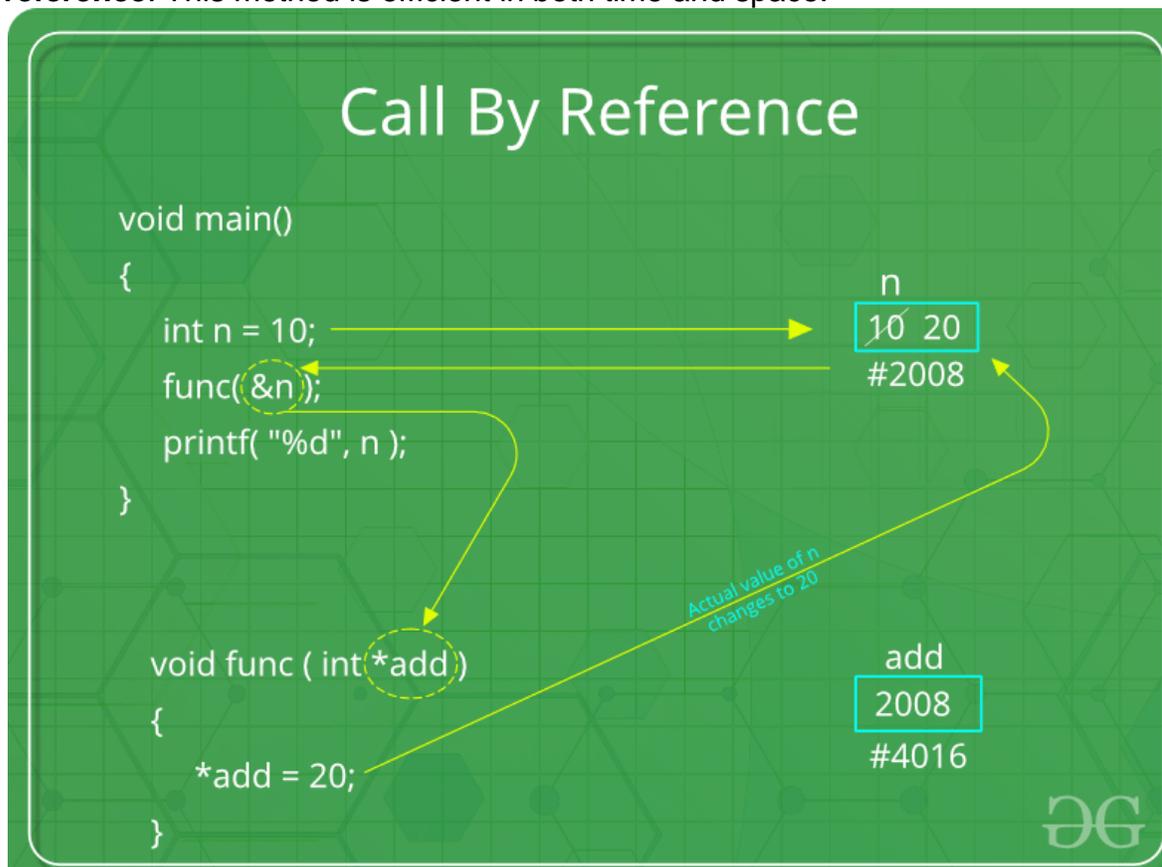
```
filter_none
edit
play_arrow
brightness_4

// C program to illustrate
// call by reference
#include <stdio.h>

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

int main(void)
{
    int a = 10, b = 20;

    // passing parameters
    swapnum(&a, &b);

    printf("a is %d and b is %d\n", a, b);
    return 0;
}
```
Output:

a is 20 and b is 10

C and C++ both support call by value as well as call by reference whereas Java does'nt support call by reference.

Shortcomings:

- Many potential scenarios can occur
- Programs are difficult to understand sometimes

Other methods of Parameter Passing

These techniques are older and were used in earlier programming languages like Pascal, Algol and Fortran. These techniques are not applicable in high level languages.

1. **Pass by Result :** This method uses out-mode semantics. Just before control is transfered back to the caller, the value of the formal parameter is transmitted back to the actual parameter.T his method is sometimes called call by result. In general, pass by result technique is implemented by copy.

2. **Pass by Value-Result :** This method uses in/out-mode semantics. It is a combination of Pass-by-Value and Pass-by-result. Just before the control is transferred back to the caller, the value of the formal parameter is transmitted back to the actual parameter. This method is sometimes called as call by value-result

3. **Pass by name :** This technique is used in programming language such as **Algol**. In this technique, symbolic "name" of a variable is passed, which allows it both to be accessed and update.
   Example:
   To double the value of C[j], you can pass its name (not its value) into the following procedure.

4. procedure double(x);

5.   real x;

6. begin

7.   x:=x*2

8. end;

In general, the effect of pass-by-name is to textually substitute the argument in a procedure call for the corresponding parameter in the body of the procedure. Implications of Pass-by-Name mechanism:

- The argument expression is re-evaluated each time the formal parameter is passed.

- The procedure can change the values of variables used in the argument expression and hence change the expression's value.

The Standard C Library:

Input/Output

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

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files 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 pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

The getchar() and 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 character 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 the 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() and 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 (End of File).

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

NOTE: Though it has been deprecated to use gets() function, Instead of using gets, you want to use fgets().

```
#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 the 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 the input from the standard input stream **stdin** and scans that input according to the **format** provided.

The **int printf(const char *format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the 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. Let us now proceed with a simple example to understand the concepts better –

```
#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() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

The last chapter explained the standard input and output devices handled by C programming language. This chapter cover how C programmers can create, open, close text or binary files for their data storage.

A file represents a sequence of bytes, regardless of it being a text file or a 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 the important calls for 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. The prototype of this function call is as follows −

FILE *fopen(const char * filename, const char * mode);

Here, **filename** is a string literal, which you will use to name your file, and access **mode** can have one of the following values –

Sr.No.	Mode & Description
1	**r** Opens an existing text file for reading purpose.
2	**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.
3	**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.
4	**r+** Opens a text file for both reading and writing.
5	**w+** Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
6	**a+** Opens a text file for both reading and writing. 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 following access modes instead of the above mentioned ones −

"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 provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

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 it is 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 the next section.

Reading a File

Given below 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 function allows 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 the 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 encountering the first space character.

```
#include <stdio.h>

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 the 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 in detail about what happened here. First, **fscanf()** read just **This** because after that, it encountered a space, second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely.

Binary I/O Functions

There are two functions, that 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.

Fopen

The C library function FILE *fopen(const char *filename, const char *mode) opens the filename pointed to, by filename using the given mode.

Declaration

Following is the declaration for fopen() function.

```
FILE *fopen(const char *filename, const char *mode)
```

Parameters

- **filename** − This is the C string containing the name of the file to be opened.

- **mode** − This is the C string containing a file access mode. It includes −

Sr.No.	Mode & Description
1	**"r"** Opens a file for reading. The file must exist.

2	**"w"**	
	Creates an empty file for writing. If a file with the same name already exists, its content is erased and the file is considered as a new empty file.	
3	**"a"**	
	Appends to a file. Writing operations, append data at the end of the file. The file is created if it does not exist.	
4	**"r+"**	
	Opens a file to update both reading and writing. The file must exist.	
5	**"w+"**	
	Creates an empty file for both reading and writing.	
6	**"a+"**	
	Opens a file for reading and appending.	

Return Value

This function returns a FILE pointer. Otherwise, NULL is returned and the global variable errno is set to indicate the error.

Example

The following example shows the usage of fopen() function.

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

int main () {
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);

   fclose(fp);

   return(0);
```

```
}
```

Let us compile and run the above program that will create a file **file.txt** with the following content −

We are in 2012

Now let us see the content of the above file using the following program −

```c
#include <stdio.h>

int main () {
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1) {
      c = fgetc(fp);
      if( feof(fp) ) {
         break ;
      }
      printf("%c", c);
   }
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program to produce the following result −

We are in 2012

Fread

The C library function size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) reads data from the given stream into the array pointed to, by ptr.

Declaration

Following is the declaration for fread() function.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Parameters

- **ptr** − This is the pointer to a block of memory with a minimum size of size*nmemb bytes.
- **size** − This is the size in bytes of each element to be read.

- **nmemb** − This is the number of elements, each one with a size of **size** bytes.

- **stream** − This is the pointer to a FILE object that specifies an input stream.

Return Value

The total number of elements successfully read are returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, then either an error had occurred or the End Of File was reached.

Example

The following example shows the usage of fread() function.

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

int main () {
   FILE *fp;
   char c[] = "this is tutorialspoint";
   char buffer[100];

   /* Open file for both reading and writing */
   fp = fopen("file.txt", "w+");

   /* Write data to the file */
   fwrite(c, strlen(c) + 1, 1, fp);

   /* Seek to the beginning of the file */
   fseek(fp, 0, SEEK_SET);

   /* Read and display data */
   fread(buffer, strlen(c)+1, 1, fp);
   printf("%s\n", buffer);
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** and write a content this is tutorialspoint. After that, we use **fseek()** function to reset writing pointer to the beginning of the file and prepare the file content which is as follows −

this is tutorialspoint

string handling functions

String is an array of characters. In this guide, we learn how to declare strings, how to work with strings in C programming and how to use the pre-defined string handling functions.

We will see how to compare two strings, concatenate strings, copy one string to another & perform various string manipulation operations. We can perform such operations using the pre-defined functions of "string.h" header file. In order to use these string functions you must include string.h file in your C program.

String Declaration

Method 1:

char address[]={'T', 'E', 'X', 'A', 'S', '\0'};
Method 2: The above string can also be defined as –

char address[]="TEXAS";
In the above declaration NULL character (\0) will automatically be inserted at the end of the string.

What is NULL Char "\0"?
'\0' represents the end of the string. It is also referred as String terminator & Null Character.

String I/O in C programming

String I/O
1) printf and scanf
2) puts and gets
Syntax of above functions - Assume string as str1

```
printf("%s", str1);
puts(str1);  --%s not require here.
scanf("%s", &str1);
gets(str1); --%s not require
```

Read & write Strings in C using Printf() and Scanf() functions

```c
#include <stdio.h>
#include <string.h>
int main()
{
    /* String Declaration*/
    char nickname[20];

    printf("Enter your Nick name:");

    /* I am reading the input string and storing it in nickname
     * Array name alone works as a base address of array so
     * we can use nickname instead of &nickname here
     */
    scanf("%s", nickname);

    /*Displaying String*/
    printf("%s",nickname);

    return 0;
}
```
Output:

```
Enter your Nick name:Negan
Negan
```
Note: %s format specifier is used for strings input/output

Read & Write Strings in C using gets() and puts() functions

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

```
#include <string.h>
int main()
{
    /* String Declaration*/
    char nickname[20];

    /* Console display using puts */
    puts("Enter your Nick name:");

    /*Input using gets*/
    gets(nickname);

    puts(nickname);

    return 0;
}
```
C – String functions

strlen - Finds out the length of a string
strlwr - It converts a string to lowercase
strupr - It converts a string to uppercase
strcat - It appends one string at the end of another
strncat - It appends first n characters of a string at the end of another.
strcpy - Use it for Copying a string into another
strncpy - It copies first n characters of one string into another
strcmp - It compares two strings
strncmp - It compares first n characters of two strings
strcmpi - It compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp - It compares two strings without regard to case (identical to strcmpi)
strnicmp - It compares first n characters of two strings, Its not case sensitive
strdup - Used for Duplicating a string
strchr - Finds out first occurrence of a given character in a string
strrchr - Finds out last occurrence of a given character in a string
strstr - Finds first occurrence of a given string in another string
strset - It sets all characters of string to a given character
strnset - It sets first n characters of a string to a given character
strrev - It Reverses a string

C String function – strlen

Syntax:

```
size_t strlen(const char *str)
```
size_t represents unsigned short
It returns the length of the string without including end character **(terminating char '\0')**.

Example of strlen:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[20] = "BeginnersBook";
    printf("Length of string str1: %d", strlen(str1));
    return 0;
}
```
Output:

Length of string str1: 13
strlen vs sizeof
strlen returns you the length of the string stored in array, however sizeof returns the total allocated size assigned to the array. So if I consider the above example again then the following statements would return the below values.

strlen(str1) returned value 13.
sizeof(str1) would return value 20 as the array size is 20 (see the first statement in main function).

C String function – strnlen

Syntax:

```
size_t strnlen(const char *str, size_t maxlen)
```
size_t represents unsigned short
It returns length of the string if it is less than the value specified for maxlen (maximum length) otherwise it returns maxlen value.

Example of strnlen:

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

```
int main()
{
    char str1[20] = "BeginnersBook";
    printf("Length of string str1 when maxlen is 30: %d", strnlen(str1, 30));
    printf("Length of string str1 when maxlen is 10: %d", strnlen(str1, 10));
    return 0;
}
```
Output:
Length of string str1 when maxlen is 30: 13
Length of string str1 when maxlen is 10: 10

Have you noticed the output of second printf statement, even though the string length was 13 it returned only 10 because the maxlen was 10.

C String function – strcmp

```
int strcmp(const char *str1, const char *str2)
```
It compares the two strings and returns an integer value. If both the strings are same (equal) then this function would return 0 otherwise it may return a negative or positive value based on the comparison.

If string1 < string2 OR string1 is a substring of string2 then it would result in a negative value. If string1 > string2 then it would return positive value.
If string1 == string2 then you would get 0(zero) when you use this function for compare strings.

Example of strcmp:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[20] = "BeginnersBook";
    char s2[20] = "BeginnersBook.COM";
    if (strcmp(s1, s2) ==0)
    {
      printf("string 1 and string 2 are equal");
    }else
     {
       printf("string 1 and 2 are different");
     }
    return 0;
}
```
Output:

string 1 and 2 are different
C String function – strncmp

int strncmp(const char *str1, const char *str2, size_t n)
size_t is for unassigned short
It compares both the string till n characters or in other words it compares first n characters of both the strings.

Example of strncmp:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[20] = "BeginnersBook";
    char s2[20] = "BeginnersBook.COM";
    /* below it is comparing first 8 characters of s1 and s2*/
    if (strncmp(s1, s2, 8) ==0)
    {
       printf("string 1 and string 2 are equal");
    }else
    {
       printf("string 1 and 2 are different");
    }
    return 0;
}
```
Output:

string1 and string 2 are equal
C String function – strcat

char *strcat(char *str1, char *str2)
It concatenates two strings and returns the concatenated string.

Example of strcat:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[10] = "Hello";
    char s2[10] = "World";
    strcat(s1,s2);
    printf("Output string after concatenation: %s", s1);
    return 0;
```

```
}
```
Output:

Output string after concatenation: HelloWorld
C String function – strncat

```
char *strncat(char *str1, char *str2, int n)
```
It concatenates n characters of str2 to string str1. A terminator char ('\0') will always be appended at the end of the concatenated string.

Example of strncat:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[10] = "Hello";
    char s2[10] = "World";
    strncat(s1,s2, 3);
    printf("Concatenation using strncat: %s", s1);
    return 0;
}
```
Output:

Concatenation using strncat: HelloWor
C String function – strcpy

```
char *strcpy( char *str1, char *str2)
```
It copies the string str2 into string str1, including the end character (terminator char '\0').

Example of strcpy:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[30] = "string 1";
    char s2[30] = "string 2 : I'm gonna copied into s1";
    /* this function has copied s2 into s1*/
    strcpy(s1,s2);
    printf("String s1 is: %s", s1);
    return 0;
}
```
Output:

String s1 is: string 2: I'm gonna copied into s1
C String function – strncpy

char *strncpy(char *str1, char *str2, size_t n)
size_t is unassigned short and n is a number.
Case1: If length of str2 > n then it just copies first n characters of str2 into str1.
Case2: If length of str2 < n then it copies all the characters of str2 into str1 and appends several terminator chars('\0') to accumulate the length of str1 to make it n.

Example of strncpy:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char first[30] = "string 1";
    char second[30] = "string 2: I'm using strncpy now";
    /* this function has copied first 10 chars of s2 into s1*/
    strncpy(s1,s2, 12);
    printf("String s1 is: %s", s1);
    return 0;
}
```
Output:

String s1 is: string 2: I'm
C String function – strchr

char *strchr(char *str, int ch)
It searches string str for character ch (you may be wondering that in above definition I have given data type of ch as int, don't worry I didn't make any mistake it should be int only. The thing is when we give any character while using strchr then it internally gets converted into integer for better searching.

Example of strchr:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char mystr[30] = "I'm an example of function strchr";
    printf ("%s", strchr(mystr, 'f'));
    return 0;
}
```
Output:

f function strchr

C String function – Strrchr

char *strrchr(char *str, int ch)
It is similar to the function strchr, the only difference is that it searches the string in reverse order, now you would have understood why we have extra r in strrchr, yes you guessed it correct, it is for reverse only.

Now let's take the same above example:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char mystr[30] = "I'm an example of function strchr";
    printf ("%s", strrchr(mystr, 'f'));
    return 0;
}
```
Output:

function strchr
Why output is different than strchr? It is because it started searching from the end of the string and found the first 'f' in function instead of 'of'.

C String function – strstr

char *strstr(char *str, char *srch_term)
It is similar to strchr, except that it searches for string srch_term instead of a single char.

Example of strstr:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char inputstr[70] = "String Function in C at BeginnersBook.COM";
    printf ("Output string is: %s", strstr(inputstr, 'Begi'));
    return 0;
}
```
Output:

Output string is: BeginnersBook.COM
You can also use this function in place of strchr as you are allowed to give single char also in place of search_term string.

Math functions :

log, sin, alike Other Standard C functions

C sin() cos() tan() exp() log() function:

- sin( ), cos( ) and tan( ) functions in C are used to calculate sine, cosine and tangent values.

- sinh( ), cosh( ) and tanh( ) functions are used to calculate hyperbolic sine, cosine and tangent values.

- exp( ) function is used to calculate the exponential "e" to the xth power. log( ) function is used to calculates natural logarithm and log10( ) function is used to calculates base 10 logarithm.

- "math.h" header file supports all these functions in C language.

EXAMPLE PROGRAM FOR SIN(), COS(), TAN(), EXP() AND LOG() IN C:
C

```
1   #include <stdio.h>

2

3   #include <math.h>

4

5   int main()

6

7   {

8        float i = 0.314;

9        float j = 0.25;

10       float k = 6.25;

11       float sin_value = sin(i);
```

```c
12      float cos_value = cos(i);

13      float tan_value = tan(i);

14      float sinh_value = sinh(j);

15      float cosh_value = cosh(j);

16      float tanh_value = tanh(j);

17      float log_value = log(k);

18      float log10_value = log10(k);

19      float exp_value = exp(k);

20

21      printf("The value of sin(%f) : %f \n", i, sin_value);

22      printf("The value of cos(%f) : %f \n", i, cos_value);

23      printf("The value of tan(%f) : %f \n", i, tan_value);

24      printf("The value of sinh(%f) : %f \n", j, sinh_value);

25      printf("The value of cosh(%f) : %f \n", j, cosh_value);

26      printf("The value of tanh(%f) : %f \n", j, tanh_value);

27      printf("The value of log(%f) : %f \n", k, log_value);

28      printf("The value of log10(%f) : %f \n",k,log10_value);

29      printf("The value of exp(%f) : %f \n",k, exp_value);

30      return 0;

31 }
```

OUTPUT:

The value of sin(0.314000) : 0.308866
The value of cos(0.314000) : 0.951106
The value of tan(0.314000) : 0.324744

The value of sinh(0.250000) : 0.252612

The value of cosh(0.250000) : 1.031413

The value of tanh(0.250000) : 0.244919

The value of log(6.250000) : 1.832582

The value of log10(6.250000) : 0.795880

The value of exp(6.250000) : 518.012817

OTHER INBUILT ARITHMETIC FUNCTIONS IN C:
- "math.h" and "stdlib.h" header files support all the arithmetic functions in C language. All the arithmetic functions used in C language are given below.
- Click on each function name below for detail description and example programs.

Function	Description
abs ( )	This function returns the absolute value of an integer. The absolute value of a number is always positive. Only integer values are supported in C.
floor ( )	This function returns the nearest integer which is less than or equal to the argument passed to this function.
round ( )	This function returns the nearest integer value of the float/double/long double argument passed to this function. If decimal value is from ".1 to .5", it returns integer value less than the argument. If decimal value is from ".6 to .9", it returns the integer value greater than the argument.
ceil ( )	This function returns nearest integer value which is greater than or equal to the argument passed to this function.
sin ( )	This function is used to calculate sine value.
cos ( )	This function is used to calculate cosine.

cosh ( )	This function is used to calculate hyperbolic cosine.
exp ( )	This function is used to calculate the exponential "e" to the xth power.
tan ( )	This function is used to calculate tangent.
tanh ( )	This function is used to calculate hyperbolic tangent.
sinh ( )	This function is used to calculate hyperbolic sine.
log ( )	This function is used to calculates natural logarithm.
log10.(.)	This function is used to calculates base 10 logarithm.
sqrt ( )	This function is used to find square root of the argument passed to this function.
pow ( )	This is used to find the power of the given number.
trunc.(.)	This function truncates the decimal value from floating point value and returns integer value.

Standard C functions

In this tutorial, you will be introduced to functions (both user-defined and standard library functions) in C programming. Also, you will learn why functions are used in programming.

A function is a block of code that performs a specific task.

Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- create a circle function

- create a color function

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Types of function

There are two types of function in C programming:

- Standard library functions
- User-defined functions

Standard library functions

The standard library functions are built-in functions in C programming.

These functions are defined in header files. For example,

- The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the stdio.h header file.
  Hence, to use the printf()function, we need to include the stdio.h header file using #include <stdio.h>.
- The sqrt() function calculates the square root of a number. The function is defined in the math.h header file.

User-defined function

You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

How user-defined function works?

```
#include <stdio.h>

void functionName()

{

    ... .. ...

    ... .. ...

}



int main()

{
```

```
    ... .. ...

    ... .. ...


    functionName();


    ... .. ...

    ... .. ...

}
```

The execution of a C program begins from the main() function.
When the compiler encounters functionName();, control of the program jumps to

```
 void functionName()
```

Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.

2. Reusable codes that can be used in other programs

3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.