Programming Expertise In C

Student Reference Book

Contents

C Programming Introduction

In this lecture you will understand:

- Why one should learn C
- What is meant by constants, variables and keywords
- Rules for building variables and constants
- How many keywords are there in C
- What happens in memory when a variable is created

Why C?

There are at least four good reasons to learn C:

(a) C is simple.

(b) C is easy to understand since it is extremely small and contains only 32 keywords.

(c) Programs written in C execute faster than the programs written in most other languages.

(d) Languages like C++, C# or Java use C syntax.

(e) Operating System like Unix, Linux or Windows are written in C.

(f) Moreover, gaming frameworks and even mobile devices use C.

(g) Communication with the hardware is easily possible using C, which was earlier possible only using assembly language.

Where C Stands

Languages can be categorized as Low-Level Languages (LLL) and High-Level Languages (HLL). The LLLs (Assembly Languages) offer better machine efficiency since they understand the machine better and are able to exploit its capabilities more effectively. Also, these languages have all the features and instructions to interact with the hardware. The HLLs offer better programming efficiency since they have been designed to make programming easier. Moreover, all high-level languages don't have facilities and instructions to interact with the hardware. C is a good mix of both. Being a high-level language it offers better efficiency as well as has all the features to interact with the hardware. Hence, C is also called a Middle-Level Language.

In The Beginning…

Learning C language is similar to learning English language. Instead of words and numbers of English, in C we build **constants**, **variables** and **keywords**. These in turn are used to form a **statement** or an **instruction**. A group of such instructions forms a **program**.

Alphabets, Digits, Special Symbols

Let us begin our study of C with the **alphabets**, **digits** and **special symbols** that it permits us to use. Alphabets include the uppercase alphabets (A - Z) and lowercase alphabets (a - z). Digits include the digits (0 - 9). In addition to these, C permits use of 32 special symbols like +, <, >, (,), {, }, %, ^, &, *, etc.

Constants and Variables

Constants are the entities that don't change, whereas, **variables** are entities that can change. In the slide **3**, **2**, and **20** are constants as they are themselves values and hence cannot be changed. On the other hand **x** and **y** are variables as their values can be changed.

C Constants

There are two types of constants viz. **Primary** and **Secondary**. Each of them further contains several types as shown in the slide. In this lecture we would focus our attention only on Primary constants.

Integer Constants

Integer constants are numbers without decimal point or fractional part in them. While building an integer constant the rules shown in the slide should be followed. The valid range of integer constants is –32768 to 32768. If we are to program an application which needs to tackle constants bigger/smaller than the range offered by integer constant we should use real constants.

Real Constants

The real constant must contain a decimal point. It can be positive or negative. No comma or spaces are allowed and the valid range is -3.4×10^{38} to $+3.4 \times 10^{38}$.

Forms of Real Constants

There are two ways to represent real constants viz. **Fractional** and **Exponential** form. Of these the fractional form is more popularly used. The exponential form is used to represent very small or very large real constants. The exponential form has the following format:

mantissa »exponent

Or

mantissa Ûexponent

Character Constants

Character constants are the constants enclosed within single quotes pointed to left. Character constants are single character wide. More than one character is not allowed within single quotes.

C Variables

All computer languages follow one cardinal rule: A particular type of constant can be stored in the same type of variable. Hence those many types of variables exist, as the number of types of constants.

Variables

Since variables are containers that hold constants in them the number of types of variables must be the same as the number of types of constants.

It is necessary to identify the types of variables so that the computer can decide what value and operations can be allowed on that variable.

What Happens in Memory

Like human memory, computer's memory also consists of millions of cells. When we say **x = 3**, one of these cells get chosen, a value **3** gets stored in it and a name **x** is given to that cell. Here onwards whenever we use **x**, the value stored under this location will get used.

How to Identify Types

Though 3, 3.0 and '3' are value-wise same C treats them as different types of constants. Looking at the constant we can easily identify its type. The slide shows their types. However, the same cannot be said about the variables.

We cannot identify the types of variables by simply looking at them. So in C, we need to declare or define the variables before using them, for example:

```
int a
float b
char c
```

Rules for Building Variable Names

While building variable names the rules mentioned in the slide should be followed. Since C is case-sensitive language, variables written in lowercase are different than the ones in uppercase. Also notice the difference between the minus sign '-' and the underscore '_'. Although variable name can contain digits and underscores, the first character of the variable name must be an alphabet.

C Keywords

In C there are 32 keywords. Keywords are words having predefined meaning in the language. For example, when we say **int i** it means that we are trying to define **i** to be an integer type. In this statement, **int** is a keyword and its meaning stands predefined. The terms **Keywords** and **Reserved words** are used interchangeably. **int**, **char** and **float** are keywords. The keywords should never be used as variable names. So, the statement **int float** is invalid.

However, integer, character and real are not keywords. Hence they cannot be used to declare variable types and can be used as variable names.

The First C Program

In this lecture you will understand:

* What is a comment and how to add the same
* How to create a simple C program
* The general form of **printf( )**

Where Do We Stand

This slide shows what we have learnt so far. In the first step, we saw what are alphabets, digits and special symbols in C's point of view. Then we learnt what are constants, variables and keywords and saw how to form C statements and instructions.

By combining instructions or statements we can create a program. How, we will see in the next slide.

The First C Program

Here we intend to calculate the simple interest for 1000.50 as the principal amount, 3 as the number of years and 15.5% per annum. In the slide we have taken variable **p** for principal amount, **n** for years and **r** for rate of interest and **si** for simple interest. We have stored the respective values in these variables and then calculated the interest amount using the arithmetic statement **si = p * n * r /100**. To let the compiler know what type of data we are going to store in these variables we must declare them at the beginning of the program. Since **p** and **r** are storing real values they are declared as **float**. Since **n** is holding an integer value it is declared to be of the type **int**. The simple interest may be a real number. Hence **si** has also been declared as **float**.

If variables are of same type we can declare them in one statement separating hem by comma (,).

Printing Values…

This slide shows memory locations containing the values 1000.50, 3 and 15.5. These memory locations are identified by the names **p**, **n** and **r**. When we calculate simple interest its value would be stored in another location and a name **si** would be given to it. However, just storing the value in memory location is not enough. This value should get printed on the screen. This printing is done using **printf( )**. The **%f** enclosed within double quotes forms the format string. **%f** indicates that we are trying to print a **float** value.

In C, **( )** are called parenthesis, **{ }** are called braces and **[ ]** are called square brackets. They have different meanings. So you should familiarize yourselves with the name of each.

General Form of *printf( )*

The general form of **printf( )** is shown in the slide. The format string can contain **format specifiers**, which specify the type of the value to be printed. Here we should use **%i** for printing **integer**, **%f** for printing **float** and **%c** for printing **character**.

The variable list contains a list of variables whose values we wish to print. The variables in the list should be separated using commas. There should be a one-to-one correspondence between the variables in the list and the format specifiers in the format string.

Any character other than format specifiers in the format string is printed as it is. We can also write the **printf( )** statement as follows:

```
printf ( "p = %f, n = %i, r = %f, si = %f, p, n, r, si )
```

Here the output would be

```
p = 1000.5, n = 3, r = 15.5, si = 465.2325
```

Statement Terminators

The way all English sentences end with a full-stop (.),all C statements end with a semicolon (;). Because of such syntax, we can write multiple statements in one line. This makes C a free form language.

What To Execute

Instead of instructing the machine to execute first the **float** declaration, then the **int** declaration, and so on, C permits us to give a collective name to the set of statements in a program. This collective name is **main( )**.

To indicate how many statements belong to **main( )** the statements must be enclosed within a pair of braces. The pair of braces is called scope delimiters. Execution of a program in C starts from **main( )**.

Comments Are Useful

A comment is as a note or a remark. It is a good practice to mention a comment at the beginning of the program indicating the purpose of the program. The comment must be enclosed within /* */.

Note that a comment is given only for our understanding and the computer always ignores it during execution of the program. Hence we can write English sentences in comments in capital or small case.

Tips About Comments

We can add comments anywhere in the program. Using /* */ we can comment out multiple lines. Nested comments are not allowed i.e. a comment should not be written inside another comment.

A More General Program

If we execute the simple interest program we would always get the same value of **si**, since the values of **p**, **n** and **r** are fixed. If we wish to calculate **si** for some other set of values we would have to make a change in the program. This should never happen. Our program should be general and flexible enough to work for any type of data.

If we are to write a general program then we should receive the values of **p**, **n** and **r** from the user through keyboard when program is running using the **scanf( )** statement as shown in the slide.

The general form of **scanf( )** is similar to **printf( )** except that the variables in the list are preceded by the 'Address Of' (**&**)operator. The actual purpose of **&** operator would be discussed later.

One More Program

The slide shows one more program that accepts three integers and calculates the average. The average is then printed on the screen using **printf( )**.

C Instructions Part-I

In this lecture you will understand:

* What are editors, compilers, linkers, preprocessor, etc.
* What do we mean by Integrated Development Environment
* Various editing commands

C Compilers

Various compiler manufacturers have designed different types of C Compilers, which offers better development environment called Integrated Development Environment that provides easy debugging, compiling and execution of programs. The slide lists various C Compilers.

Integrated Development Environment

An Integrated Development Environment consists of:

(a) Editor

(b) Compiler

(c) Preprocessor

(d) Linker

(e) Debugger

Editor helps to type or edit program. Editor provides easy editing commands so that writing programs becomes easy.

Compiler is a translator, which translates a C language program to machine language program, which a processor can understand.

A preprocessor is used to include different files in one file and expand macros if any in the program.

A linker is used to link the function calls from the program with the library that contains the function body.

A debugger is used to search for bugs (problems) in the code.

Editing Commands

Editing commands consists of some Cursor movement and Deletion commands.

Cursor movement keys consists of

(a) Arrow Keys

- Up Arrow – moves cursor to one line up
- Down Arrow – moves cursor to one line down
- Right Arrow – moves cursor to one place right
- Left Arrow – moves cursor to one place left

(b) Home – moves cursor to the beginning of the line

(c) End – move cursor to the end of the line

(d) PgUp (Page Up) – moves cursor to the beginning of the previous page

(e) PgDn (Page Down) – moves cursor to the beginning of the next page

(f) Ctrl+Home – moves cursor to the beginning of the screen

(g) Ctrl+End – moves cursor to the end of the screen

(h) Ctrl+PgUp – moves cursor to the beginning of the file

(i) Ctrl+PgDn – moves cursor to the end of the file

Deletion Commands consists of

(a) Del – to delete a character

(b) Backspace – to delete a character left to the cursor

(c) Ctrl+T – to delete a word (Place cursor to the beginning of the word to be deleted)

(d) Ctrl+Y – to delete a line (Place cursor on the line to be deleted)

Some More Commands

File Menu consists of some Opening and Closing file commands,

(a) New – to open a new file

(b) Save – to save the file

(c) Save as – to save the file with different name

(d) Open – to open an existing file

(e) Exit – to quit out of the Integrated Development Environment (IDE)

(f) DOS Shell – to temporarily exit IDE and to execute DOS commands

Miscellaneous Commands

(a) F2 – to save the current file

(b) Ctrl+F9 – to compile and execute the program

(c) Alt+F5 – to view the output

(d) Alt+F3 – to close the current window or file

Always give relevant names to program so that they can be easily identified with their names.

Interchanging Contents of Two Variables

In this slide we are attempting to write a program to interchange the contents of two variables. Initially we have declared two variables **c** and **d**. We have taken the input through the **scanf( )** statement. Assume that 5 and 10 are entered through the keyboard for **c** and **d**. Now to interchange values, **c** is assigned the value of **d** (**c**'s original value has been lost forever) hence **c** and **d** both contains value 10. Again the value of **c** is assigned to **d** hence no change in **d**, **c** and **d** still remains as 10 and 10.

Hence when printed using **printf( )** statement the values of **c** and **d** are printed as 10 and 10 and it is confirmed that values are not interchanged yet.

Interchanging Contents of Two Variables

This slide shows the same program with few modifications. Another temporary integer variable **t** is declared, which initially doesn't contain anything.

Firstly the value of **c** is assigned to **t**, then the value of **d** is assigned to **c**. Here **c** will loose its original value. But that value is stored in **t**. Since **d** should contain that we have assigned the value of **t** to **d**. Now the values have been interchanged. You can trace these assignments in the slide.

One More Way

The slide shows one more way to interchange the values of the variables, which do not use third variable **t**. The program accepts the values of **c** and **d**, add them and store in **c**, subtract **d** from **c** and store it in **d** (which is value of **c**) again, subtract **d** from **c** again and store it in **c** (which is value of **d**). Finally the values have been interchanged.

Sum of Digits

Now we intend to calculate the sum of individual digits of a number. Consider a number 26913. The sum of individual digits: 2, 6, 9, 1, and 3 needs to be calculated. In the program shown in the slide the number is taken in **n** using **scanf( )**.

To separate the individual digits of a number we need to apply the Modulus (%) operator on the number and use 10 as the divisor. This would give us the last digit of the number because Modulus returns the remainder. If we use Division (/) operator with 10 then we would get the number except the last digit. This would be required to extract the second last digit and then subsequent digits.

The Whole Picture

Once the number is accepted from the user and stored in **n**, firstly we have extracted the last digit of the number and stored it in **d5**. We have the reduced the number by dividing by 10. Suppose the number is 26913. The number would now become 2691. This number has been stored in **n** again. Hence now **n** would contain a new number i.e 2691. Since second last digit has now become the last one we have applied the modulus operator on this number again. This extracted the last digit again. We have stored it in **d4**. Likewise we have extracted the digits 3 more times and stored them in **d3**, **d2** and **d1**. We have then added the values of **d1**, **d2**, **d3**, **d4** and **d5** to obtain the sum of individual digits and stored the answer in **s**. Finally the sum is printed through **printf( )**.

Is % (modulus) Really Useful

The modulus (%) operator is used in various applications shown in the slide.

In case of leap year, year has to be divided by 4 and if the remainder is zero then year is considered as leap year.

Also to find odd or even number, if a number on division by 2 results into 0, then it is even, otherwise odd.

In case of prime number too, we need to divide and check the remainder, if it is 0 then the number cannot be prime and if there is no factor until one less than the number, the number is prime.

C Instructions Part - II

In this lecture you will understand:

* What are type declaration instructions
* Various arithmetic instructions
* What do we mean by associativity
* How many numbering systems can be used
* Various escape sequences and their usage

Where Are We…

We have seen simple example of how to define variables. Now we will see more ways to define variables. Then we will see types of arithmetic instructions, input/output instructions and control instructions.

Type Declaration Instruction

Variables in C should be declared using type declaration statement. This has been discussed while writing the first program. The type declaration statements consist of two things, one the type and second the list of variable names separated through commas.

Initialization can also be done at the time of declaration. In the slide the declaration statement **int a ;** and assignment statement **a = 5 ;** is grouped together as **int a = 5 ;**. First declaration would be done and then 5 would be assigned to **a**.

Also, more than one variable can be declared as well as initialized in one type declaration statement as follows:

```
int a = 5, b = 10 ;
```

An expression can also be used to assign the answer to the variable being declared, which is as follows:

```
int a = 5, b = 10, c = a + b * 5 % 2 ;
```

But for this the variables used in the expression must be declared prior to the variable to which the expression is used. Hence the order of declaration is important and the order is left to right.

We can assign multiple variables in the single statement. For example,

```
int a, b, c, d ;
a = b = c = d = 5 ;
```

For this to work all the variables must be declared prior to assignment. Hence this works but the following statement doesn't:

```
int a = b = c = d = 5 ;
```

Arithmetic Instruction

Having understood the type declaration instructions, we now move on to Arithmetic Instructions. Arithmetic Instructions are used while calculating a value. Arithmetic instructions use the arithmetic operators shown in the slide. In C there is no exponentiation operator. To calculate power we can use **pow( )** function for which we need to add **#include "math.h"** before **main( )**.

The left hand side of the assignment operator must always be a variable. It cannot be a constant or an expression. This is because expressions get evaluated to constants.

Unlike algebra, in C, in arithmetic statements no operator is assumed, for example:

a = b (c + d) ;

Here **b** is being multiplied by **(c + d)**, hence ***** is necessary.

Types of Arithmetic Instruction

There are three types of Arithmetic Instructions as shown in the slide. When all operands are integers it is called integer mode arithmetic operation. When all operands are real it is called real mode arithmetic operation. When some operands are integers and some real then it is called mixed mode arithmetic operation.

In the example shown in the slide **a**, **b**, and **c** are declared as **int**. Hence the result of the arithmetic instruction evaluates to an integer value.

The first set of operators shown in the slide refer that they are arithmetic operators. The second indicates an assignment operator. The third is the statement terminator. The fourth group indicates that **c**, **a**, **b** are variables, whereas, **5**, **6**, **14** are constants and can be used as operands for the operators.

Legal Arithmetic Operations

The slides shows what would be result if operands are either only integers or only reals or both.

Try This

It is important to know that if an arithmetic operation were a mixed mode operation then it would always be carried out in the type that is more powerful. Between **int** and **float**, **float** is more powerful. Also, if it were a real mode arithmetic operation the result would be in real mode. Suppose some evaluation gives 5.5 as a result. If this result were initialized to an integer variable the number would be truncated to 5, means only the integer part would get stored.

In the left half of the slide, that variable **a** has been declared as **int** hence all the results of the statements are only integer values. Even when mixed mode arithmetic operation is performed the result is **int** itself.

In the right half of the slide, **a** has been declared as **float** hence real values get stored in **a**.

Which is Correct

The result of the first expression in the slide is **0** since both **5** and **9** are integers. Hence, in the next statement **9** is changed to **9.0**, which is a real constant and hence the expression becomes mixed mode operation, which would always be done in real mode. The parenthesized expression is evaluated first. Hence, parenthesis must be given for the part of expression, which is to be evaluated first. So, in the last statement the expression **(f – 32)** is evaluated first.

Hierarchy/Priority/Precedence

Similar to algebraic BODMAS rule, which specifies the order of operations to be performed, C has also some rules of operator evaluation called hierarchy or priority or precedence of operators. In the slide the numbers indicates the order of arithmetic operations. The arrow in the figure moves from the operator at the lowest level hierarchy to highest-level hierarchy.

Associativity

Consider the expression given in the slide. Here there is a tie between operators of same priority, that is between / and *. This tie is settled using the associativity of / and *. But both enjoy Left to Right associativity. The slide shows for each operator which operand is unambiguous and which is not.

Since both / and * have L to R associativity and only / has unambiguous left operand (necessary condition for L to R associativity) it is performed earlier.

Associativity

Consider one more expression given in the slide. Here both assignmetnt operators have the same priority and same associativity (Right to Left). The slide shows for each operator which operand is unambiguous and which is not. Since both = have R to L associativity and only the second = has unambiguous right operand (necessary condition for R to L associativity) the second = is preferred earlier.

Input/Output Functions

This slide revises the syntax of **printf( )** and **scanf( )** that we have seen earlier. For output we use the **printf( )** statement and for input we use the **scanf( )** statement.

printf( )

The general format of **printf( )** consists of two things viz., the format string and list of variables.

The list of variables is optional, whereas, the format string consists of any message, format specifiers and escape sequences.

The variables in the list of variables must be separated with commas. Also, the number of variables in the list must be equal to the number of format specifiers in the format string.

Numbering Systems

In the slide four Number Systems are explained. In Decimal number system, 10 symbols (0 – 9) can be used to form a decimal number. In Octal, 8 symbols (0 – 7) can be used to form an octal number. In Hexadecimal, 16 symbols (0 – 9, A – F) can be used to form a hexadecimal number. C allows us to use any of the three while using a constant. Binary is not allowed in C. But all the numbers ultimately get converted to binary before they are actually processed by the processor.

Conversions

The slide shows how to convert from hexadecimal to decimal and octal to decimal. To convert from hexadecimal to decimal we need to multiply each digit of the number with base of the system i.e 16 raise to **n**, where **n** is the place of the digit. The right-most digit has a place **0**. The second right-most digit has **1** and so on. Likewise, to convert an octal to decimal we need to multiple each digit of the number with 8 (base) raise to place number. Lastly, the results of all multiplications need to be added to obtain the decimal number.

The slide also shows conversion from decimal to binary, hexadecimal to binary and octal to binary. To convert to binary we have to divide the number with its base. The division must go on until it becomes **0** and store the remainder in each step. Then starting from the bottom all the remainders should be combined to form the actual binary number.

printf( ) Makes it Handy

If we want to print the hexadecimal and octal equivalent of a decimal number we can write the following statement:

```
printf ( "%d %o %x %X", 10, 10, 10, 10 ) ;
```

Here we have used **%d**, **%o**, **%x** and **%X** format specifiers to print the decimal, octal and hexadecimal equivalent of decimal 10. The difference between **%x** and **%X** is that **%X** would print the characters (A-F) in hex number in capital, whereas, %x would print in small case.

To represent an octal constant we need to write **0** (zero) before the number. For example, 077. To represent a hexadecimal constant we need to add 0x or 0X before the number. For example 0xA, 0XA.

Note, that real constants cannot be converted to hexadecimal or octal equivalents using **%o** and **%x**.

Escape Sequences

Escape sequences are non-printable character constants having more than one character within single quotes. The '\' is used as escape character to be understood by the C compiler that the character followed by '\' represents non-printable character such as newline ('\n'), tab ('\t'), etc. The newline

takes the cursor to the beginning of the next line. The tab character takes the cursor to the next tab character on the screen.

Any Other Characters

The slide shows what all a **printf( )** statement can contain.

scanf( )

The **scanf( )** statement have similar format as that of **printf( )** except the variables in list of variables precede with ampersand (**&**), which specifies the address of variables where the values accepted through the keyboard to be stored. As shown in the slide, an input for octal and hexadecimal variables can be supplied as **022** and **0abc**, as well as **22** and **abc** respectively.

Decision Control Instructions Part-I

In this lecture you will understand:

* What are control instructions
* What are decision control instructions

C Instructions

The first two types of instructions were covered in the last lecture. In this lecture we wish to explore the Control Instructions.

Control Instructions

Control Instructions control the sequence of execution of instructions in a program. The different types of control instructions are:

(a) Sequence

(b) Decision

(c) Repetition

(d) Case

Normal C Program

Unless explicitly specified the instructions in a program get executed one after the other, or sequentially. All programs that we have developed so far used the Sequence control instruction. This in fact is the default control instruction.

Decision Control Instruction

Let's now take a look at the Decision Control Instruction. Suppose we intend to calculate the total expenses incurred on purchasing a quantity for a particular price. We wish to offer a 10 % discount if the quantity purchased is more than or equal to 1000. So we cannot calculate total expenses unless we check whether the quantity supplied is more than 1000 or not. And to check this we need to use a Decision control instruction. The usage of this instruction is shown in the next slide.

Slide Number 5

The quantity accepted through the **scanf( )** statement in **qty** is checked using the **if** statement. The **if** statement always includes a condition within the pair of parenthesis. Here it is checked whether the value of **qty** is greater than 1000 or not. If the value of **qty** is found to be greater than 1000 then the **dis**, which represents discount, is assigned the value 10, else **dis** is assigned the value 0.

Here the operator >= checks whether the left-side value is greater than or equal to the right-side value. This operator is called relation operator.

Next, we have calculated the total expense and displayed it.

Tips

The **if** and **else** are keywords. The general form of **if-else** is shown in the slide.

For building the condition we can use the following relational operators:

(a) < Less than

(b) > Greater than

(c) <= Less than or equal to

(d) >= Greater than or equal to

(e) == Relational equal to

(f) != Not equal to

While comparing two entities for equality we need to use == (relational equal to), whereas, for assigning one entity to another we need to use = (assignment operator).

Note that relational operators always evaluate to either 1 or 0. If the condition is true then it evaluates to 1 otherwise to 0.

Would This Work

We are allowed to use expressions in **printf( )**. The expression can be an arithmetic expression like **a + b** or a conditional expression like **a <= b**. On evaluating a conditional expression it is replaced by 1 if it evaluates to truth and by 0 if it evaluates to false.

Is it Monday or Tuesday

Until a variable is initialized with some value, it holds a garbage value. A garbage value is any unpredictable value that is present in the memory cell when that cell gets reserved for a variable. In the slide, since we have not initialised **a** with any value a garbage value gets printed.

Slide Number 9

In this program, we have initialised **dis** with **0** at the time of declaration. If the quantity entered by the user is greater than 1000, we have assigned **10** to **dis** otherwise we have directly calculated the total expense, with **dis** equal to **0**. Conclusion is writing **else** block is optional.

Slide Number 10

In this program, in the **if-else** block, we want that three statements should get executed. However, out of three statements, only the first statement after **if** and **else** belong to them. The second and third statements after **if** are treated as normal statements that are not belonging to **if**. So, on compiling this program we would get an error saying 'misplaced else'. This error occurs if the compiler finds an **else** block without an **if** block.

Slide Number 11

The error 'misplaced else' can be removed by enclosing the statements in pair of braces. We have done the same in this slide.

One More Form

The slide shows the general form of **if-else** block when multiple statements are to be executed in them.

Leap Year or Not

Let's now try a program to determine whether a year is leap or not. It is wrong to believe that if the year is divisible by 4 it is leap, otherwise not. The correct logic to find out whether the year is leap or not is as follows:

(a) A century year divisible by 400.

(b) A non-century year divisible by 4.

Slide Number 14

In the slide firstly we have checked whether the year is a century year or not using the condition **y % 100 == 0**. If the year is a century year we have checked whether it is divisible by 400. If the year is a non-century year then we have checked whether it is divisible by 4.

Decision Control Instructions Part-II

In this lecture you will understand:

* The working of **sizeof( )** operator
* What are relational operators
* How to create a program. to find out whether the year is leap or not
* The working of **if-else** statement

First Day of Any Year

Having dealt with the logic to determine a leap year, we now attempt to find the first day (Monday, Tuesday…) of any year. In this slide we have accepted the year.

The Logic Behind It

1/1/1, i.e. 1st January; 1 was Monday according to the Gregorian calendar. To find out the day on 29/1/1 we need to carry out three steps:

(1) Find out the number of days that have passed from 1/1/1 to a day prior to 29/1/1, i.e. upto 28/1/1.

(2) Divide this number of days by 7.

(3) If the remainder is 0, then coming day, i.e. 29/1/1 is Monday. If remainder is 1, then 29/1/1 is Tuesday and so on.

Slide Number 3

The logic discussed for 29/1/1 in the previous slide can be extended to find out the day on first day of the year entered through the keyboard. Suppose the year entered through the keyboard is 1998, we need to find out the number of days that have passed from 1/1/1 to 31/12/1997.

This calculation has been split into two parts:

(a) The **normaldays** variable accounts for 365 days that occurred in every year from year 1 to year 1997.

(b) The **leapdays** variable accounts for one extra day that occurred in every leap year from year 1 to year 1997.

Summation of the two would give total number of days between 1/1/1 and 31/12/1997. Once this has been obtained, modulus by 7 is carried out. If the remainder turns out to be 0 then 1/1/1998 is Monday, if it is 1 then 1/1/1998 is Tuesday and so on.

Syntax error occurs when there is a grammatical error in the syntax of the statement. For instance in the example,

printf ("%d , 23) ;

the quotation marks are not completed, hence a Syntax error. Semantic errors are logical errors, like in the example **if (a = 2)**, we wish to check if a contains value 2, however, here instead of comparing the values, 2 would get assigned to **a** as = is used in place of ==.

Slide Number 4

The expression **(y - 1) * 365** may yield a value greater than the range of an **int**. Hence to store such value we need to define **normaldays** and **totdays** as **long int**. The **long int** is a data type that has range of –2147483648 to 2147483647.

What's Wrong Here?

Just declaring **normaldays** and **totdays** as **long int**s is not enough. This is because **(y - 1) * 365** operation is being carried out in integer mode as **y** is an integer variable and 1 & 365 are integer constants. If the whole operation were in integer the result would also be in integer. Hence even **normaldays** is declared as **long int** the result that would be assigned is integer. There are two solutions to it. Either make 1 or 365 as **long int** constant by writing 1L or 365L or define **y** as **long int**. Doing either of this would make **(y - 1) * 365** a mixed mode operation.

To print or take **long int** as input we need to specify **%ld** as the format specifier.

The *sizeof( )* Operator

The **sizeof( )** is an operator in C. It gives the size of a variable or a type in bytes. Every **int** variable occupies 2 bytes and every **long int** variable occupies 4 bytes in memory. The **sizeof** is also a keyword.

Are You Sure

The upper half part of the slide shows arithmetic statements. While doing modulus division the remainder would always take the sign of the numerator.

In the lower half of the slide execution of conditional statement would give 'Hi' as output. First **a** == **b** is compared, which results in 1. Then 1 is compared with the value of **c**, which results in 0. Hence the **else** block gets executed.

More Decision Control Instructions Part-I

In this lecture you will understand:

* What are logical operators
* When to use logical operators
* How logical operators work
* Hierarchy of operators

What Will Be The Output

The output of the program shown in the slide is 30. Carefully observe the **if** statement. It is terminated by semicolon. The statement **b = 30 ;** doesn't belong to the **if** block even if it is supposed to. The semicolon after the **if** statement forms the null statement, which is executed when the condition is true. Since **b = 30 ;** lies outside the **if** block it is executed irrespective of whether the condition is true or false.

What Will Be The Output

In the program shown in the slide we have used = in the condition written in **if** statement. Being assignment operator value 5 gets assigned to **a**. Then **if (a)** is executed. The value 5 is non-zero and so, the condition is treaded as true. Note that not only 1 any non-zero (including a negative number) value in C is treated as true and zero as false. Hence the output of the program is **Hi**.

Slide Number 3

This program uses nested **if-else** statements. Its working is easy to understand but it has three problems:

(a) The indentation of the statements will increase with the number of conditions

(b) Needs care for matching **if**s and **else**s

(c) Braces should be matched

Slide Number 4

The problems mentioned in the previous slide seem to have been avoided in this program. But if you look carefully there are a lot of other problems.

Suppose percentage marks turns out to be 70. For this value the first three conditions turn out to be true. Hence the student ends up passing in all three divisions. Something that is totally wrong.

Slide Number 5

In this program we have combined conditions using a 'logical AND' operator (**&&**). When combined using && the expression is considered to be true only if both the conditions turn out to be true. However, there is still a minor flaw here. If the student gets 70 percent marks then the condition **if (per >= 40 && per < 50)** fails. As a result, control goes to the else block and prints out fail. This is wrong. It can be rectified by explicitly writing all conditions as shown in the next slide.

Slide Number 6

Finally, all the problems have been solved. So is this a better program as compared to the one that used nested **if-else**s? Yes and no. Yes because the three problems of indentation, too many **if-else** to be matched and too many { } to be matched have been avoided.

However, the performance suffers. If the first condition gets satisfied the rest of the conditions still get unnecessarily checked. This consumes time thereby degrading the performance.

One More Way

The solution shown in the slide would work the best.

Logical Operators

The **&&**, ||, and **!** are logical operators. Here && is Logical AND, || is Logical OR and ! is Logical NOT operators. They are generally used for checking ranges and for yes/no problems. yes/no problem are problems, which on evaluating a complicated set of conditions ultimately result into true or false. For example, after checking several conditions if the result is either the student has passed or failed then this becomes a yes/no problem.

Slide Number 9

The program determines whether a person should be insured or not. It accepts age, sex and martial status of a person. The criteria for insuring a person are:

(a) If the person is married

(b) If the person is unmarried, is male and is above 30 years of age

(c) If the person is unmarried, is female and is above 25 years of age

(d) In all other cases the person is not insured.

The program checks these conditions in the same order and reports whether the person should be insured or not.

Note that the user of the program has to enter **m** for male, **f** for female and **m** for married marital status, **u** for unmarried marital status.

Using Logical Operators

This slide shows the previous program using logical operators. First we have checked the marital status using the condition **ms == 'm'**. This condition is combined with other conditions using ||. Hence if this condition is evaluated to true then other conditions would not be checked and this is what we wanted. If this condition evaluates to false then the next condition where we have checked for unmarried marital status and male and above 30 years of age would be evaluated. If this evaluates to false then the last condition would be checked. If all conditions evaluate to false then **else** block would be executed otherwise the **if** block would be executed.

Working of && And ||

The working of the logical operators **&&** and || are given in the slide.

(a) The output of **&&** is true if both the operands are true, otherwise false.

(b) The output of || is false if both the operands are false, otherwise true.

Hierarchy of Operators

The slide shows hierarchy of all the operators that we have used so far. The unary operators have a higher priority over the binary operators. Unary operators need only one operand, whereas, a binary operator needs two operands.

Consider the expression

a = -3 + b –5 ;

Here the unary operator –3 has the highest priority, the + and – that follows in the expression after –3 has the second and third priority (L to R). The **&** operator used in the **scanf( )** statement has the higher priority as it is a unary operator. Other unary operators that can be used in an expression are also shown in the slide.

More Decision Control Instructions Part-II

In this lecture you will understand:

* What are conditional operators
* How are they different?

Exchanging Blocks

The **if** or **else** block can be exchanged by reversing the condition in the **if** which is shown in the slide.

One More Way

One more way of reversing the condition is by using 'not' (**!**)operator. How to use this operator is shown in the slide. But the condition given in the slide would not work properly and wrong output i.e. **B** would get displayed. This is because, since the **!** operator enjoys highest priority firstly **!a** would get evaluated resulting into 0 (not of non-zero value is zero), which is then compared with 4.

The Correct Way

The program has been modified, by putting the condition in the pair of parenthesis. Now firstly **a > b** is checked and then the result of this condition gets negated.

Yet Another Way

Another way of writing **a > b** is **! (a <= b)**.

What Would be The Output

We know that logical operators always evaluate to either true or false. Hence applying Logical NOT to a variable would result in either true if the value of the variable is zero and to false if the value of the variable is non-zero. Since true in C is 1 and false is 0 if we print the result we would either get 1 or 0.

Point Out The Error

When you execute the program given in the slide you will get an 'Lvalue Required' error message for the statement **c = !a || b = 7 ;**. Here because of higher priority of || with respect to = (equal to). **!a || b** would be executed before b = 7. The result of **!a || b** may turn out to be 1 or 0. And 7 cannot be assigned to either of them, hence the error.

In the **printf( )** statement shown in a block in the slide, the terminating quotes of the format string are missing as a result, compiler flashes error as 'Unterminated string or character constant'.

What Would Be The Output

The statement causing error in the previous slide can be modified as shown in this slide. The parentheses are used for giving higher priority for the assignment operation.

Yet Another Way

The program given in the first half of the slide simply checks whether **a** is less than or equal to **b** and prints **A** or **B** accordingly. This program can also be done using **conditional** operators.

The **?** and **:** are conditional operators. These are used as a replacement of **if-else**. Their general form is:

condition ? statement1 : statement2

If the condition gets satisfied then statement1 gets executed and if the condition turns out to be false the statement2 gets executed.

However, while using **? :** more than one statement cannot be given in the true or false part of the conditional operators.

How Would You Convert

If the **:** and the statement following it are dropped it would result into an error.

Remove The Error

We may attempt to rectify the error by writing a null statement after the colon. However, the error doesn't vanish because now the semicolon acts as a terminator.

No Errors At Last

If we put one more semicolon believing that it would act as the null statement the error still persists, because first semicolon acts as a terminator and second semicolon acts as a null statement.

This problem can be resolved putting some dummy statement as shown in the slide. These do not serve any useful purpose except for possibly satisfying the ego of the conditional operators.

Moral of The Story

A few tips...

- If we use **?** we must use **:**. They always go hand in hand.
- In the **?** and **:** there must at least be one statement. A dummy statement would also do.
- It is not a replacement of the **if-else** statement as only single statements are allowed in the **?** and **:** parts.

Some More Different Ways

Some more forms of the conditional operator are shown in the slide. In the first form parentheses are necessary otherwise the error 'Lvalue Required' would be flashed because of operator precedence.

The second form shows that **? :** can be used for assigning the value evaluated through the conditional operator.

They can also be used in **printf( )** statement as shown in the slide.

Conversions

In this lecture you will understand:

* How to convert from one numbering system to another
* How to print graphics characters

Conversions

In this slide we will see simple examples of **int** to **float** conversions. The first **printf( )** prints the values of variables as expected.

In the second **printf( )**, **float** variable **d** is printed with **%d** which is an integer format specification. Since **float** to **int** conversion is unpredictable it prints correct values for **a**, **b** and **c** but prints **0** for **d**.

In the third **printf( )** the integer **a** is printed using **%f**. This produces a wrong output. Since the first conversion goes wrong it messes all the conversions of subsequent variables. Hence this time **printf( )** would print unpredictable output for **a** as well as all the subsequent variables..

Conversions Continued...

Let us now see conversion between an **int** and **char**. The first **printf( )** uses the usual format specifiers for **int** and **char** and prints the expected output.

In the second **printf( )** the **int** variables **i** and **j** are printed using **%c**. This outputs A and Z.

If the **char** variables **ch** and **dh** are printed with **%d** format specifier the output is 65 and 90.

This indicates that there is some relation between 65 and A and 90 and Z. In the next few slides we would try to understand this relationship.

Representation

Let us now see how an **int**, **char** and **float** are stored in memory. Each gets converted to its binary equivalent and then stored in memory. Conversion of **int** or **float** to binary is done through division/multiplication by 2. However, such division/multiplication would not be possible on the character 'A'. Hence ASCII values (American Standard Code for Information Interchange) are used for storing a character.

From the slide it can be noted that binary equivalent for the number 65 and binary representation of character 'A', whose ASCII value is 65, are same.

Methods Are Different

Consider the binary representation of 65 and 'A'. Though their binary representations are same they are obtained differently. 65 is converted by carrying out repeated divisions by 2 and writing remainders in reverse order. As against this, binary of A is represented using the ASCII code. When we print **i** using **%c** we are asking it to print character equivalent of the values stored in **i**. This turns out to be 'A'. Similarly, when we ask it to print **ch** using **%d** we are asking it to print decimal equivalent of the value stored in **ch**. This turns out to be 65.

What About More Than 255

If **int** variables are printed using **%c** (character format specification) then only the binary representation of the lower byte is used for the character conversion as shown in the slide. Hence if we print 300 using **%c** character equivalent of **00101100** would get printed.

Why Unreliable Conversions

As we have seen earlier **float** to **int** or **float** to **char** conversions are unreliable. Let us now understand why this is so. When a **float** variable is declared, 4 bytes get reserved in memory and binary representation of floating point value is stored. Now when printed through **%c** only the value present in lower 1 byte gets printed since **char** is of 1 byte. Similarly, when printed using **%d** the value present in lower 2 bytes gets printed since **int** is of 2 bytes. Hence the unreliable result.

If an **int** variable is defined 2 bytes get reserved in memory and the binary representation of the value gets stored in them. If the value is printed with **%f**, along with the 2 bytes of the variables the adjacent 2 bytes are used to print the **float** value. Since we do not know what these two bytes contain the result becomes unpredictable.

Surprised?

Consider the first program shown in the slide. The condition given in **if( )** should evaluate to **false** and give output as **B**. However, we get an output as **A**. Considering the result of first program, if we execute the second program the output comes out to be **B**. The reason for this surprising output would be discussed in the next few slides.

Appearances Are Misleading

By default a constant without a decimal point is considered to be an **int**. By default a constant with decimal point is treated as **double** and not as **float**. Hence the size of 0.7 is reported as 8 (i.e. the size of a **double**).

Binary of A Float

If 0.7 is converted to binary, the conversion goes into unending process, which we call recurring. Value wise the 4-bytes recurring binary equivalent of 0.7 is smaller than the 8-bytes binary equivalent of 0.7. If we convert 5.375 to a binary it is accommodated in 8 bytes as **5.375** is non-recurring number.

This clarifies why in the programs given in slide number 8 gave unexpected results. In the first program the condition turns out to be true resulting in **A** getting printed out. (As a comparison you can imagine that 1.3333 is less than 1.3333333, even though both are representations of 4/3). In the second program value wise 8-byte **5.375** turns out to be same as 4-byte **5.375.** Hence the condition fails and **B** gets printed.

Safety

By default a constant without a decimal point is an integer. If we want it to be a **long** integer we have to add a suffix **L**. Similarly, a constant with a decimal point is a **double**. If we want it to be a **float** constant we need to add a suffix **f**. In the **if** statement in the first program (top-left corner) since **a** and **0.7f** both are **float**s the condition evaluates to false resulting in **B** getting printed out. In the second program (bottom-left corner) the variable **a** of type **double** is initialized with **0.7**. Then we have compared **a** with the **double** constant **0.7**. As both 64-bit recurring binary equivalent are same, we get **B** as the output.

In the third program (top-right corner) **a** is of type **float** having value **5.375**. This **a** is then compared with **float** constant **5.375**. As both are same the condition evaluates to false resulting into **B** getting printed. Similarly in the fourth program (bottom-left corner) **a** of type **double** with value 5.375 is compared with **double** constant 5.375. As this is a non-recurring double value the condition evaluates to false resulting in **B** getting printed out.

ASCII Code

This slide shows the ASCII codes for the alphabets.

Characters to be Represented

This slide shows the characters that we would wish to represent. There are total 256 characters whose ASCII values are from 0 to 255 i.e., starting from 00000000 to 11111111 in binary.

Graphics Characters

The editor window of Turbo C is shown in the slide. T in window is built out of a rectangle. We can also construct such a window using graphical characters present in the ASCII character set.

To type the ASCII characters in the Turbo C editor, use Alt + ASCII value of the character. For example, to print a vertical line press Alt + 179. The ASCII equivalent of various graphical characters is shown in the slide.

ASCII Values

This slide shows the range of ASCII values representing each group.

While Part-I

In this lecture you will understand:

* What are repetition control instructions
* What are incrementation & decrementation operators

Control Instructions

We are already through with the sequence control instructions and decision control instructions. Let us now start with the repetition or loop control instructions.

To Begin With...

To begin with consider the program that calculates the simple interest. Suppose we wish to calculate the simple interest for different principal values with varying interest rates. To achieve this we have to execute the program again and again. Instead it would be better if the language itself provides features for repeating a set of instructions. This indeed is the job of the loop control instruction.

Repetition

To repeat the statements we can use **while** statement as shown in the slide. The **while** is a reserved word or a keyword.

Using the **while** statement we can repeat a set of statements. However, we must specify how many times the statements should get repeated. Hence in the **while** statement we must specify a condition. As long as this condition is true the statements would be repeated. In this program the statements written below the **while** statement will get repeated till the value of **i** is less than or equal to **10**.

We saw that using a **while** loop we can repeat a set of statements. However, if we run the program without initializing the variable **i** the statements in **while** loop may not get executed at all. This is because a variable (**i**), if not initialized would contain a garbage value which can be more than 10. For the condition to be satisfied we must first initialize **i**. Hence in the slide the variable **i** is initialized with 1.

Now the variable **i** stands initialized with a value 1. The control would enter **while** loop because the condition is true. As long as this condition is true the **printf( )** statement would be repeated. To break the loop the condition should become false means **i** should become greater than 10. And since we have not incremented the value of **i** or changed the value of **i** the loop will run infinite number of times.

Let's do all the things that are necessary to make **while** loop run properly. Firstly we have initialized **i** a value 1. By doing so the condition satisfies and the control enters **while** loop. Note that we have enclosed multiple statements within the pair of braces. This is because we want multiple statements to get executed repeatedly. Like the **if** condition the default scope of **while** loop is only one statement if braces are not used.

In the last statement we have incremented the value of **i** by 1. In every repetition (iteration) the value of **i** would be incremented by 1 and as it becomes 11 the loop would break.

Logic Doesn't Matter

The loop counter can be of type **int**, **long**, **float** or **char**. Also the loop counter can be incremented or decremented by a suitable value as required. For example, in the program that we saw earlier we have incremented the value of **i** by **1**. We could have incremented it by **2**. In this case the loop would have executed for **5** times. Similarly, we can initialize the value of **i** as **10** and decremented it by **1**. In this case our condition would be **while (i >= 1)**. If the loop counter variable is of type **float** then the incrementation and decrementation can be in steps of fractional numbers.

Another Program

The program shown in the slide is similar to the one in the previous slide. There is only one difference. The statement **i = i + 1** has been replaced by **i++**. The ++ is called as incrementation

operator which increments the value of the operand by one. The expression **i**++ works same as **i** = **i** + **1**.

Incrementation/Decrementation Operators

On similar lines we have one more operator called decrementation operator that decrements the value of an operand by one. Thus the expression **i**-- is same as **i** = **i** - **1**. Remember the way we use operators ++ and --, we cannot use operators such as **, //, %%.

For incrementing the value of a variable by 1 we can write **i** = **i** + **1** or **i**++. Can we use +++ operator to increment the value of operand by **2**? If yes, then to increment the operand by 10, should we repeat + 10 times? This doesn't make any sense. We can only use ++**i** or **i**++. There is no +++ operator in C.

Would This Work

Let's see some subtleties of increment or decrement operators. The first example tells that increment or decrement operator can be used in expressions. Next it is shown that writing incrementation/decrementation operator before the variable is valid. But it cannot be used with constants as in slide increment/decrement is used with 3. The third example on the left hand side is also invalid. This is because the expression **(j + k)** would evaluate to a constant, and to that constant we are trying to apply the incrementation operator ++, hence incorrect. The statement **i** = **j**----**2** ; is valid as it is treated as **i** = **j**-- - -**2** which means that we are trying to decrement the value of **j** by one and then subtract –**2** from **j**. The statement **i** = **j**----**2** **;** is not valid as it is treated as **i** = **j**-- --**2** where again decrementation operator is applied to a constant **2** which is invalid.

While Part-II

In this lecture you will understand:

* What is pre & post incrementation/decrementation
* The loop control instructions
* The working of **while** loop
* How to generate prime numbers

Is i++ Same As ++i

Let us check whether **i++** is same as **++i**. Both the expressions increment the value of **i** by 1. But there is slight difference in the way the expressions get evaluated. Let's first see how **i++** works. When we write **i++** in **printf( )** the value of **i** would get printed and then the value of **i** would get incremented by one. Hence, in every iteration, the value of **i** would get printed first and then would get incremented by 1.

If we use the expression **++i** in place of **i++**, then first the value of **i** would get incremented by one and then the incremented value would get printed. For the program shown in the slide, we expect the output as 1, 2, 3....10, but the output would be 2, 3, 4 ... 11. This is because the initial value of **i** is 1. As mentioned earlier, **i** would get incremented to 2 and then it will be printed. Thus, when the value of **i** would reach 10, it will get printed, and again the condition being true, **i** would get incremented to 11 and would then get printed. The correct way to get the desired output is shown in the next half of the slide.

To get the output as **1 2 3 4 .....9 10** the program in the previous slide can be modified, by initializing **i** with 0 and changing the condition to **i < 10** as shown in the slide. If you now run through the statements you would find that the program correctly prints values from 1 to 10.

Thus, both **pre** and **post** incrementation operators can be used to get the same output, but with a slight modification in the initialization of variables and the condition.

Two More Ways

We can print numbers from 1 to 10 in two more ways as shown in the slide.

Look at the conditions in the **while** statement. In case of the condition **i++ < 10** of the first program, first the condition is tested with the value of **i** and then **i** is incremented. As against this, in case of the condition **++i < 10** of the second program, **i** gets incremented first and then the condition is tested with the incremented value of **i**.

Compare

Let us compare the different ways in which we can print numbers from 1 to 10. In the program on the left-hand side of the upper half, first the value of **i** would get printed and then the value of **i** would get incremented. Whereas, in the program on the right-hand side of the upper half, first the value of **i** would be incremented and then the value of **i** would be printed.

Two more ways are shown in the lower half of the slide. In the condition **i++ < 10** the condition is tested with the value of **i** and then **i** is incremented, whereas, in the condition **++i <= 10**, **i** is incremented first and then the condition is tested with the incremented value of **i**.

Print Nos. From 1 To 10

Let us consider the same program to print numbers from 1 to 10. Note the change in the incrementation statement. The statement **i += 1** is equivalent to **i++**, **++i**, **i = i + 1**.

If we write

```
i += 5 ;
```

then the value of **i** would be incremented by 5 which is equivalent to **i = i + 5**. Similar to +=, we have -=, *=, /=, **%**=. As these operators work together with the assignment operator they are called as compound assignment operators. The examples of *= and **%**= are shown in the slide.

One More Way

Let us see one more way to print the numbers from 1 to 10. The program shown in the slide would go on printing numbers infinite number of times. This is because, the non-zero value **1** in place of condition in the **while** statement is treated as true value and this will remain constant. Thus, the condition being true for all iterations the statements in the **while** block would get executed infinite number of times.

To break the **while** loop the **break** statement can be used as shown in the slide. The **break** statement is generally associated with a condition. It transfers the control to the first statement after the loop. The **break** statement is used to terminate the loop. If we wish to terminate the execution of the program itself we have a function **exit( )**. Its usage is shown in the slide.

Calculate

Let us see how to calculate the factorial value of a given number, sum of first **n** natural numbers and the value of one number raised to the power of another. The factorial can be calculated by multiplying the numbers from **1 to n**. The summation can be found out by accumulating the result of addition of numbers that vary from **1 to n**. The power can be calculated by multiplying a number **n** number of times. As in each of these cases a number is varying for **n** number of times, looping technique can be used to solve the problem.

Running Sum And Products

We have first declared and initialized a few variables. Variables **s, p, pr** would be used to store the sum, product and power respectively. The variable **i** is used as a counter and **n** to store the number entered by the user. The condition **i <= n** would cause **n** number of iterations. In every iteration, the statements enclosed within **while** block would get executed thereby calculating running sum, product and power. The result is printed after the loop is over. The corresponding values of **i** and **s** after each iteration are shown in the slide where value for **n** is considered as 5.

Slide Number 8

Now let us try to find out the sum of first ten terms of the series shown in the slide. The pseudo code for the program is shown in the slide. The value of the **x** is to be accepted through the keyboard. Initialize the variables that are required as a counter and to store sum. Now through the loop calculate the numerator part and the denominator part of the term. Then calculate the term by dividing numerator by denominator. Then either add or subtract the term from the sum **s** depending on the value of the counter. To repeat the **while** loop 10 times the condition **i <= 10** is used. Lastly print the sum **s**.

Slide Number 9

The program is shown in the slide. Initially the variables that are required for the program are declared. The value of **x** is accepted through the keyboard. The sum variable **s** is initialized to 0 and the counter **i** variable is initialized to 1. The statement **j = 2 * i - 1 ;** in the outer **while** loop would generate numbers like 1, 3, 5...etc. as shown in the slide. Now to calculate the numerator and denominator part of the term, we have used one more **while** loop, which is a nested loop that gets repeated **j** times. After calculating the numerator and denominator the term **t** is calculated. Finally after checking whether **i** is even or odd using the conditional operator the sum is calculated. The sum **s** has been printed after the loop is over.

Note that the statement **(s = s + t)** must put in parenthesis, otherwise it would result into an error.

Prime No.

Let us now write a program to find out whether a given number is prime or not. First a number **n**, is accepted through keyboard. Say the number 13 is entered. Now how do we find out whether this number is prime or not? A number is said to be prime if it is divisibly by 1 or the number itself. Thus, to find whether 13 is prime or not we need to carry out divisibility test for 13 by dividing it with numbers from 2 to 12.

In **while** loop, in every iteration it is checked whether **n** (i.e. 13) is divisible by **i** (the loop counter), if yes then the message 'Not a prime number' is printed and the loop is terminated otherwise **i** is incremented by 1. We have used the **break** statement, which would terminate the loop the moment divisibility test is satisfied. The control is transferred to the next statement following the **while** block. However for a given number, if the divisibility test does not get satisfied, then after the loop is over the value of the counter would be equal to the number **n**, which means the number is a prime number. In our case 13 happens to be a prime number.

Output?

Let us find out the output of the program shown in the slide. The value of **i** being 5, i.e., non-zero and **j** being 4 again a non-zero, on **AND**ing **i** with negation of **j**, the result would be 0 as the condition 1 **AND** 0 would evaluate to 0.

Next, consider the expression whose answer is assigned to **b**. The slide shows the two ways (marked by red arrows) in which the statement **b = ++i && ++j || ++k ;** may get interpreted. As the **&&** operator enjoys a higher priority than || the compiler would interpret the statement as shown below:

```
b = ( ++i && ++j ) || ++k ;
```

Here the **&&** operator would get evaluated first and then the || operator. But before the logical **&&** operator get evaluated, the unary prefix increment ++ operators would get evaluated. Thus, in this case first due to incrementation operator the value of **i**, and **j** would become **5** and **6** respectively. As these are non-zero values the result of **5 && 6** would be 1. Next, this result should get **OR**ed with **++k**, but as the result of the first condition is 1, there is no necessity to evaluate the second expression. Hence the expression **++k** would never get evaluated and the result of || operation is 1. So the output for **k** would remain –1. On similar lines you can evaluate the expressions shown at the top of the slide. In the expression **cond1 && cond2** given at the bottom of the slide, the expression in **cond2** would be evaluated if expression in **cond1** evaluates to true. Similalry, in the expression **cond1 || cond2** the expression in **cond2** would be evaluated if expression in **cond1** evaluates to false.

For

In this lecture you will understand:

* The working of **for** loop
* Differences between **while** and **for** loops

Loop Control Instructions

So far we have seen the **while** loop, now lets switch to the next looping statement, i.e. the **for** statement.

The *while* Loop

Before switching to the next looping statement **for**, let us summarize how **while** works.

After initializing the loop counter the condition in the **while** loop is checked. Thus initialization is done only once, whereas, in the loop the Test, Incrementation part keeps getting repeated till the time the condition remains true.

The *for* Loop

As against a **while** loop, in a **for** loop the initialisation, testing and incrementation are written in the same line. On execution, firstly the initialization statement gets executed, then the condition is tested and after executing the body of the **for** loop, the incrementation goes to work. The control flow of the **for** loop is shown in the slide. Note that this control flow is same as what we saw for a **while** loop in the previous slide.

The semicolons given in the **for** statement are necessary to separate initialization, condition and incrementation/decerementation part.

Comparision - I

In this slide the comparison of **for** loop with **while** loop is explained. If the braces are omitted then the **while** loop would get executed infinite number of times. On the other hand the **for** loop would get repeated for finite number of times.

The scope of **while** statement is the first statement immediately after **while**. This statement keeps getting repeated since the condition remains true all the time. The statement **i**++ never gets a chance to get executed.

Whereas, in case of **for** loop **statement1** would be repeated 10 times. Since the incremenation would always be done the **for** loop runs for finite number of times.

Comparision - II

Let's compare the **while** loop with the **for** loop in one more way. The programs shown in the slide contains a semicolon after the **while** and the **for** statement. This semicolon acts like a null statement. This null statement now forms the scope of the two loops. As a result, the **while** loop becomes an infinite loop as the statements within the pair of braces do not get a chance to ever get executed. This so happens because on satisfying the condition the null statement gets executed and the control again goes back to the condition in the **while** statement.

On the other hand the **for** loop is a finite loop as the incrementation is executed after executing the null statement. After executing the null statement 10 times, the statements within the braces would get executed sequentially.

Print Nos. From 1 To 10

The program to print numbers from 1 to 10 looks more elegant with the use of **for** loop as compared to a corresponding **while** loop.

Dropping Initialisation

We are allowed to drop the initialisation part in the **for** statement (we cannot drop the semicolon, however). But for the loop to work the loop counter must be initialized hence it has been initialized at the time of declaration.

Dropping Incrementation

We can omit the incrementation, but for the loop to terminate or work finite number of times, it should be present within the body of the **for** loop. Once again we cannot drop the semicolon after the condition.

Dropping Condition

We can drop the condition along with the initialization and incrementation expressions in the **for** loop. However, two semicolons cannot be omitted. But these omitted expressions must be explicitly mentioned in the block wherever necessary as shown in the slide. To minimize the number of statements within the block either of the two methods shown can be adopted.

Comparision – III

Let's compare the **while** and the **for** loop in one more way. If the condition in the **while** statement is omitted then an error would be flashed, whereas, in case of **for** if all the expressions are omitted and the separators are given then the **for** loop would get executed for infinite number of times. This is because if we drop the condition from the **for** it is treated as true always.

Drawing Boxes

A screen consists of 25 rows and 80 columns, numbered from 0 to 24 and 0 to 79 respectively. Let us attempt to draw a rectangle from 10^{th} row, 20^{th} column to 23^{rd} row, 70^{th} column as shown in the slide.

Slide Number 24

The ASCII equivalents for the other corner characters are 217 (bottom-right), 191 (top-right), 192 (bottom-left). To print these characters at appropriate positions we have to place the cursor at the required position. This is achieved using **gotorc( )** function that places the cursor at the desired row-column position. We also need to print the vertical and horizontal characters that would join the corners of the rectangle. The ASCII equivalents for these characters are 179 and 196. We have drawn the vertical and horizontal lines in **for** loops. The slide shows how to draw vertical lines. Try to draw horizontal lines on your own.

Note that we have **#include**d the "goto.c" file. The contents of this file are given below:

```
# include "dos.h"
union REGS i, o ;
gotorc ( int x, int y )
{
    i.h.ah = 2 ;
    i.h.dl = y ;
    i.h.dh = x ;
    i.h.bh = 0 ;
    int86 ( 0x10, &i, &i ) ;
}
```

You can create the "goto.c" file and write this function in it.

Do While

In this lecture you will understand:

* How to do multiple initializations in the **for** statement
* How to do multiple incrementations in the for statement
* How to implement the logic using **do** – **while** loop
* Difference between **for**, **while** and **do-while** loops
* Where to use **break** and **continue** statements

Combinations

The slide shows all possible combinations of digits 1, 2, 3. We intend to write a program that would generate these combinations.

Starting Off

Let us try to write a program that would generate the combinations shown in the previous slide. In the program we have declared three integer variables namely **i, j** and **k** each initialized with 1. Next, through a **for** loop we have printed the values of **i**, **j** and **k**, where **k** is varying from 1 to 3. Thus the program would generate only the first three combinations.

Adding One or More Loop

Now add one more **for** loop to the same program. The outer **for** loop would run three times with **j** varying from 1 to 3. For each value of **j** the inner **for** loop would run three times with **k** varying from 1 to 3. On execution the program would generate the 9 combinations shown in the slide. But still this is not the desired output.

Finishing Off

This time we vary **i** too from 1 to 3. For each value of **i**, j takes three values, and for each value of **j**, **k** takes three values. This results into generation of 27 combinations.

Unique Combinations

The combinations of digits shown in the earlier slide were not unique. By the term unique, we mean that while printing the values of variables, no two variables should have the same value. To get such unique combinations, before printing the values, we must check that **i** should not be equal to **j, j** should not be equal to **k** and again **k** should not be equal to **i**.

The condition **if (i != j && j != k && k != i)** would print the values of variables **i**, **j** and **k** only when their values are not matching. The condition shown in the rectangular box in the slide is wrong because it checks the result of **i != j** with **k**.

One More Way

One more way to generate unique combinations of 1, 2, 3 is shown in the slide. Here instead of checking for inequality of the variables we are checking the equality. Thus, when **i** is equal to **j**, or **j** is equal to **k**, or **k** is equal to **i**, **break** would simply terminate the **k** loop and continue with the next value of **j**. This appears to be alright, but on execution this program generates only 2 combinations. The culprit here is the **break** statement. Ideally when a non-unique combination is encountered instead of terminating the **k** loop the next value of **k** should be generated. The next slide shows how this can be achieved.

The Correct Way

Instead of terminating the loop with **break** statement we have used the **continue** statement. As a result, on executing the **continue** statement the controls is transferred to the **k**++ part of the k-loop. The values of variables **i**, **j** and **k** get printed only when they are unique.

Instead of **continue** we could as well have used a null statement (i.e. a ;) to produce the same results. However, do not form an impression that a null statement is a replacement for **continue**. This can be verified from the loop shown in the box in the slide. Here when **continue** goes to work "Hi" doesn't get printed as control reaches **k**++. If **continue** is replaced by a null statement, then on execution of the null statement the control is bound to reach **printf( )**.

Multiple Initializations

Here **i** and **j** are being initialised at one place, whereas, **k** is being initialised at another. Instead we can do multiple initializations in the **for** statement itself as shown in the next slide.

When we do multiple initializations in the **for** statement the initializations must be separated using a comma. Also multiple conditions can be grouped together using logical operators **&&** and ||. If we wish we can drop incrementation/decrementation of any/all variables.

Types of Loops

So far we have seen **while** and **for** loops. Now let us see another type of loop, the **do-while** loop.

The *do-while* Loop

Let us now write a program that generates numbers from 1 to 10 using the **do-while** loop.

In case of **do-while** loop the initialization of the counter variable must be done before the loop begins. The incrementation expression for the counter variable should be present inside the **do-while** block. The condition in the **do-while** loop is tested at the end of the loop. As no testing is done when the loop begins, the statements inside the loop are executed at least once even if the condition fails for the first time.

The statements in the loop must be enclosed within a pair of braces even if there is a single statement in the loop. Also, the semicolon after the **while( )** is a must. Make it a convention to write **while** following the closing braces of the **do-while** loop so as to avoid confusion between **do-while** and **while**.

Compare

We have now three methods two generate a loop within a program. Is there any difference between these three techniques? Let us find it out.

The **do-while** loop would give output as 'Hi', whereas, the other two loops would not output anything. This is because in case of **for** and **while** the condition is checked before the control enters the loop. And since the condition is false the **printf( )** is not executed even once.

As against this, in the **do-while** the testing of the condition is done at end of the loop. So, the **printf( )** statement would print 'Hi'. Next, the condition being false the loop would get terminated.

Effects of *break & continue*

Now let us see the effect of **break** and **continue** statements on **while**, **for**, and **do-while** loops. The slide shows where the control would reach on execution of **break** and **continue** in each of the three loops. Note that **break** or **continue** statement is associated with **if** and can be can be used in conjunction with loops (**for/while/do-while**) only. They cannot be used with if( ) only as shown in the slide.

Unknown No. of Times

Now consider the program shown in the slide where the loop has to be repeated for an unknown number of times.

In the program we have initialized variable **ch** with value **'y'**. To begin with the condition is true. The users choice, accepted through the keyboard would get stored in **ch**. If user presses **n** the loop would get terminated. Thus, at the time of compiling the program, how many times the loop would get repeated was not known.

Now, the problem with this code is that, even when the user supplies uppercase **Y** as an input, the loop would get terminated. This is because as given in the condition the choice for **ch** should only be lowercase **y**.

To allow our program to accept both lowercase and uppercase character any of the three conditions shown in the slide can be used. The function **toupper( )** converts lowercase alphabet to an uppercase. Whereas the function **tolower( )** converts an uppercase alphabet to lowercase. Do not forget to include "ctype.h" header file in your program so as to make use of these functions.

Switch

In this lecture you will understand:

* Where to use **break** and **continue** statements
* How to use **switch-case** control instruction

Control Instructions

Till now we have covered sequence, decision and repetition control instructions. In repetition control instructions we discussed the working of loop forming statements **while**, **for** and **do-while**. We also studied the effect of **break**, **continue** statements as well as that of incrementation, decrementation and compound assignment operators. Now let us focus our attention on 'Case control' instructions.

Slide Number 2

The program shown in the slide accepts a number between 1 and 3 as a choice and according to the number entered by the user it prints the appropriate message. If the user enters number greater than 3, then the program would print a message as 'Wrong choice'.

Though looks very easy, this program has three problems. Suppose the maximum number that can be entered is 10. More the number of alternatives to make a choice, more would be the number of statements. Secondly, the number of nested **if-else** would also increase. The statements would slide towards right due to indentation. Thirdly, tracking of a logical error would become difficult.

We have two alternatives. Either use logical operators or case control instruction. Logical operators would not be of any use, as we do not want to check a number that falls in a range. Rather we want to check for the number constants. The best alternative, when we want to check for numeric constants, is to use the case control instruction.

Slide Number 3

The program written earlier using **if-else** statements is now written using the **switch-case** statement as shown in the slide. The control statement, which allows us to make a decision from the number of choices is called a **switch-case**. The integer expression following the keyword **switch** is any C expression that will yield an integer value. Let us see how the **switch-case** instructions get executed.

First, the number **n** entered by the user is passed to **switch**. The value is then matched, one by one, against the constant values that follow the **case** statement. When a match is found, the program executes the statements following that **case**, not only that but all the subsequent **case** statements would also get executed. How to handle a situation when the user enters wrong choice? It is incorrect to write else for the wrong choice. **else** is not allowed in the **switch-case** statement.

Slide Number 4

Instead of the **else** statement we have to use **default** label as shown in the slide. When the number is not matched with any of the given **case** statements then the control is transferred to **default** case.

Now, suppose user enters a choice as **2** then the output 'You entered 2' is expected. The actual output is shown in next slide.

Slide Number 5

As mentioned earlier, when a match is found, the program executes the statements following that **case**, not only that but all the subsequent **case** statements would also get executed. Thus, instead of one line, there would be three lines of output as shown in the slide. The solution to overcome this problem is discussed in the next slide.

The Solution

You can place the statement or group of statements following each of the **case** within a pair of braces. But, the braces are optional. Even if you include braces for multiple statements, the subsequent **case** statements would get executed sequentially. To get desired output we will have to make use of **break**

statement with every **case**. When a particular **case** is satisfied, **break** would transfer the control to the next statement that comes immediately after the **switch-case** statement.

What If *continue*

We are wrong if you think that the way **break** works **continue** statement shall also work with **switch-case**. As you know, **continue** statement is associated with a condition hence it is illegal to use it with **switch-case**.

Slide Number 8

As shown in the slide, the **case**s in a **switch** can be written in any order. The order is not important. Secondly, even if there are multiple statements to be executed in each **case** there is no need to enclose these within a pair of braces.

Slide Number 9

Moreover, it is not necessary to have **default** label at the end of the **switch**. It can be placed at the beginning before all **case** statements.

Slide Number 10

Even the **default** case is optional. It can be omitted if not required. In absence of **default** case, the program simply falls through the entire switch and continues with the next instruction that follows the control structure.

Slide Number 11

As shown in the slide, like integer constants, the character constants can also be checked using **switch-case** statement. In the program shown in the slide, we have given **case** statements that check for an uppercase alphabet. Can we not check for the lowercase alphabets? Yes, we can.

Slide Number 12

One can think of using logical operators to check both, the lowercase and uppercase constants. When we compare character constants their ASCII values are taken into consideration. When these ASCII values (non-zero values) are grouped together with the logical operators would evaluate to true value (1). As a result, all the **case** statements shown in the program would become 1. This tells that more than one constant cannot be specified with a **case** label.

Slide Number 13

The solution for the problem we have seen in the previous slide is to write separate **case** labels for each constant value that is to be tested. Group together those **case** labels for which you want a particular statement or group of statement to be executed.

In the program shown in the slide, we have written separate **case** statements for a lowercase and uppercase constant which are then followed by the statements that are to be executed when an alphabet is entered in either of the case (upper or lower).

Suppose user enters an alphabet as **a**, then the first **case** would get satisfied. As there are no statements to be executed in this **case** the control would automatically reach to the next **case 'A'** and would execute all the statements in this **case**. This happens because, when a **case** is satisfied the control simply falls through the **case** till it doesn't encounter a **break** statement. Note that here the order in which the cases are written is important. Library function **toupper( )** or **tolower( )** can also be used to convert from small case to capital case character or from capital to small case character respectively. But this function cannot be used for numbers.

The program given in the slide can be modified by using **switch (toupper (ch))** in place of **switch (ch)**. By doing so we can omit the cases for lowercase alphabets.

Would This Work

The program shown in the slide shows that the variables cannot be used with the **case**. Only constants (**char** or **int**) are allowed with **case**.

General Form

Having gone through the subtleties of the **switch-case** statements, let us form some general rules for **switch-case** statement.

The expressions that contain constants or variables can be given as an argument with **switch**. The expression would get evaluated and the result would be passed to **switch**. With the **case** an expression with only constant values should be used.

Checking Switch

We can check only integer and character constants with **switch-case**. Real constants are not allowed.

Now the question arises as can we use **switch** as a better replacement for **if**? Yes and no. Yes, because it offers a better way of writing program as compared to **if**, and no because in certain situations we are left with no choice but **if**. The situations where one wants to compare floats, number range and even expressions that include variables can be better handled by **if-else**.

Next question that arises is which works faster? **Switch-case** works faster then **if-else**. This is because the compiler generates a jump table for **switch** during compilation. As a result, during execution it simply refers the jump table to decide which **case** is satisfied. As against this **if-else** are slower because they are evaluated at execution time. A **switch** with 10 **cases** would work faster than an equivalent **if-else** ladder.

Menu

In this lecture you will understand:

* How to create menu driven program
* How to define a function
* How to declare a function
* How make function calls

Menu Management

We would try to develop a menu-driven program. The menu would contain four items as shown in the slide. The first item in the menu should appear in the 10th row and 20th column. Correspondingly the second item would appear in 11th row, 20th column, third item in 12th row and 20th column and so on.

Slide Number 2

Let's attempt to write the program. The first step is to display menu. The code shown in the slide displays the menu. **gotorc( )** function would position the cursor at suitable row and column. The **printf( )** following the **gotorc( )** would print a menu item at this cursor position. The **gotorc( )** function has been defined in a file called **goto.c**.

Slide Number 3

For managing the menu, users' choice is accepted as soon as menu is displayed. The actions to be taken can be decided with the use of **switch** statement. Each **case** handles the corresponding choices given in the menu. In the **default** case i.e. for the choice other than those listed in the menu, a beep is sounded as a warning to the user that he has made a wrong choice. The beep can be sounded by sending an escape sequence '\a' to the **printf( )**. '\a' stands for alert.

If more than one beep character is specified in the **printf( )** statement then a long beep would occur since the time span between the first beep ending and second beep starting is very small.

No *while*, No Menu

The menu should continue to appear till the user doesn't select 0 from the menu. To ensure this a **while** loop has been inserted to run indefinitely. But now to terminate the program, the **exit( )** statement is used in place of **break** statement in **case 0**, which terminates the program.

For any menu to work it would be necessary to use an infinite loop and the **switch** statement inside it.

Break From The Outermost Loop

Suppose there are three nested loops as shown in the slide and we want to break out of the innermost loop. The inner **for** loop has got a condition with a **break** statement. If this condition becomes true **break** statement would not transfer control out of all the loops, rather it would merely transfer the control to the statement that comes immediately after the inner loop(in our program from **k** loop). To terminate the subsequent outer loops we need to again specify the same condition and the **break** statement. This would be tedious if the number of loops is more.

In such a case **goto** statement can be used.

Better Way

If the condition in the inner loop is satisfied then we want the control to go outside the outermost loop. For this **goto** statement can be used as shown in the slide.

The **goto** statement can transfer the control anywhere in the program. Thus, on satisfying the condition the statement **goto out**, would transfer control to a place in the program where **out** label is written. The label is a word followed by a **:** (colon).

A null statement has been given, as we do not want to perform any other action once control reaches outside the outermost loop.

Never use **goto** in any other situation except breaking the control outside the outermost loop from within the innermost loop.

Where Am I

Instead of going out of the loop we can also transfer control inside the loop as shown in the slide. If you use several **goto**s within a program soon it would become difficult to trace out how the control is flowing.

Control Instructions

We are now through with the control instructions available in C. We have discussed all the control instructions listed in the slide.

Functions

Let us now move on to another feature of C called functions. Functions minimize the complexity of the program by breaking the code and writing it in separate units known as functions.

The function familiar to us is **main( )** from which the execution begins. A function is identified by the identifier name followed by a pair of parentheses. The functions that we have used in our programs so far are **printf( )**, **scanf( )**, etc. Note that **for( )**, **while( )**, **if( )**, **switch( )** are control statements, and not functions even though they are followed by a pair of parentheses.

The program in the slide consists of three functions, **main( )**, **bombay( )** and **kanpur( )**. The output of this program is as shown in the slide as the functions **bombay( )** and **kanpur( )** do not get executed at all.

Calling Functions

Whenever a function is to be executed, a call to that function should be made. The function name followed by a pair of parentheses and terminated by **;** (Semicolon) identifies a call to the function. The function name followed by block is called definition of the function.

In this program firstly the **printf( )** statement in **main( )** would get executed. Now on calling **bombay( ),** the control is transferred to the body of the function **bombay( )**. After executing the statements in this function, the control would returns to the statement from where the call to **bombay( )** is made. Since that statement only contains the call, the next statement is executed. Now, the function **kanpur( )** would get called similarly. Hence the output as shown in the slide would get generated.

Functions

Functions are categorized as standard library functions and user-defined functions. Standard library functions are those functions, which come ready-made with the compiler, whereas, user-defined functions are the functions that are defined by the user. The **printf( )**, **scanf( )**, etc., functions are library functions, whereas, **kanpur( )**, **bombay( )**, **gotorc( )**, which are defined by us are called user-defined functions. Even **main( )** is a user-defined functions as we define it every time when a new program is written. But the compiler generates a call to main.

Tips

Go through the tips shown in the slide.

Functions

In this lecture you will understand:

* How to pass values to the functions
* How to returning a value from the function

Order, Order

The functions can be written in any order. Functions get executed whenever they are called. Call to the functions decides when to execute them, and not the order in which they are defined.

Whatever be the order, execution begins from **main( )**. The program shown in the slide proves this by printing **I am in main**. And then the function **bombay( )** is called, which prints **I am in Bombay**.

More Calls, More Bills

We can call the same function any number of times. But if there were more calls to a function it would slow the execution of the program. This is because, whenever we call a function the control is transferred to the function and after executing the body of the function the control returns to the statement from where the call is made. Hence if more calls are made, time is wasted in passing control and returning control.

Nobody Is Nobody's Boss

Any function can be called from any other function. The program shown in the slide illustrates this point. Through **main( )**, function **bombay( )** has been called, which calls function **kanpur( )** and function **kanpur( )** in turn calls the function **bombay( )**.

Local v/s STD v/s ISD Calls

The way we have different types of phone calls, like Local, STD or an ISD call, **C** also categories function call as simple or recursive call to a function. Thus, when a function calls itself, we say that a recursive call to a function has been made. The process is known as Recursion.

In the program, function **main( )** has called itself, hence **main( )** here is a recursive function.

Communication

Let us now see how a program communicates with the function it has called. In the program shown in the slide, we have defined a function **calsum( )**. In **main( )** variables **a**, **b**, **c** and **s** are declared. Of these variables **a**, **b** and **c** are initialized with some integers. The variable **s** has not been initialized so it will hold a garbage value. Next, a call to **calsum( )** function has been made. In this function also we have declared same variables **a**, **b**, **c** and **s**. This is a different set of variables than the ones declared in **main( )**. As variables **a**, **b**, **c** are not initialized they will hold garbage values and so **s** will hold sum of these garbage values. After printing the value of **s** control will return to **main( )** and **printf( )** would be executed. Again a garbage value would get printed, as **s** of **main( )** too holds garbage value.

Note that, the garbage values printed by function **calsum( )** and **main( )** would be different. This clarifies that variables declared in **main( )** and those declared in **calsum( )** have no connection.

Passing Values

Since **a**, **b**, **c** declared in **main( )** are not available to **calsum( )** we need to pass their values to the **calsum( )** function. This can be done by enclosing these variables within a pair of parentheses while calling the **calsum( )** function. These values are called actual arguments or parameters.

The values passed are collected in the corresponding variables present within the parentheses of the called function. These variables are called formal arguments.

In the program the values of variables are passed to the function **calsum( )**, by the statement,

```
calsum ( a, b, c ) ;
```

The values of **a**, **b** and **c** are collected in the variables x, y, z. As these variables are new to the function, they must be declared. While declaring formal arguments they must match in number, order and type. By number we mean that if 3 actual arguments are passed to the function then it must receive them in three formal arguments. By order and type we mean that the formal arguments must be in same sequence and with same data type as that of the actual arguments.

Next, the sum is calculated and stored in **s**. But in **main( )** when we print the value of **s** it turns out to be the garbage value. This is because the value 60 that was calculated in the **calsum( )** function is neither returned and nor collected in **main( )**.

Returning Values

Note the modifications done in the program to get the desired output. In the function **calsum( )**, we have added **return** statement that returns the sum to **main( )**.

In **main( )**, the sum returned has been collected in variable **s** through the statement,

```
s = calsum ( a, b, c ) ;
```

The **return** statement returns a value. We can either write a constant, a variable or an expression after **return**. However **return ;** simply returns the control to the calling function. **return (ss) ;** returns the value of variable **ss**. **return (60) ;** returns 60 to the calling function. **return (x + y + z) ;** returns the result after evaluating the expression.

Are These Calls OK?

All the calls to the **calsum( )** function shown in the slide are valid. As illustrated in the slide, the first three calls to function **calsum( )** are not collecting the returned values. This proves that the returned values can be ignored.

In the third statement within a call, a call to same function **calsum( )** has been made. Here the inner **calsum( )** would get executed first. The value returned by this call would be used as second parameter for the outer call to **calsum( )**. Thus nested calls are legal.

The last statement shows that an expression can have function calls. The value returned by the function would replace the function call.

Returning More Than One Value

How to return more than one value? In an attempt to return more than one value the way shown in the program is incorrect as a function can return only one value at a time.

One More Try

Let's try some other way to return more than one value. In the program two separate calls have been made. In the first call the value returned is received in the variable **s**. In the next call the value returned is received in the variable **p**.

Now in the function **sumprod( )** two return statements are added, that would to return the sum **ss** and product **pp**. In **main( )**, we have printed the values received through **sumprod( )**.

The expected answer is **60** and **6000**. However, the answer would be **60** and **60**. How?

The reason is, the **return** statement immediately returns the control to the calling function, hence with every call made in the **main( )**, only the first **return** statement would return the value, and second would never be executed. This confirms that a function can return only one value at a time.

The Only Way Out

If we wish that **sumprod( )** should return sum and product we should pass one more variable to it. Depending on the value of this variable the function should return either the sum or the product. If **1** is passed **sumprod( )** would return sum and if 2 is passed **sumprod( )** would return the product.

The same idea can be extended to **return** average, variance, standard deviation, etc.

A Better Way

In the earlier slide the value of **code** is checked through the **if** statement. Instead, we can use conditional operators. However **return** statement cannot work in **? :**. Hence the most compact way to write the function is to use **? :** within the **return** statement as shown in the slide.

Advanced Features

In this lecture you will understand:

* How to returning non-integer values from the functions
* What are Pointers
* How to print address of variables

ANSI V/s K&R

For declaring the formal arguments we have two notations: ANSI, K & R (Kernighan and Ritchie). In ANSI notation all the variables are declared within the parenthesis following the function name. Every variable needs to be declared independently. This becomes tedious if there are a large number of variables of same type. In K & R notation there can be a common declaration for variables of the same type. Most of the compilers today use the ANSI notation.

Roman Equivalent

We wish to now write a program that converts a given year into its roman equivalent.

In **main( )**, a call to user defined function **romanize( )** has been made. To this function we have passed the value of **y**, which represents year, as a parameter. In function **romanize( )**, only the equivalent for 1000 is printed. So, our task still remains incomplete, as the entire year has not been represented so far.

A More General Call

Now, in the call to function **romanize( )** we have passed three parameters, the year **y**, decimal number **1000** and its roman equivalent **'m'**.

In function **romanize( )**, we have found out how many 1000s are present in **yy** and then we have printed those many 'm's. Lastly we have returned that part of **yy**, which is not divisible by 1000.

Slide Number 4

Here **romanize( )** has been called several times. If the year supplied to **scanf( )** is 1998, then in the first call it determines how many 1000s are present in 1998 and print out those many 'm's. In the second call it determines how many 500s are present in 1998 and prints out those many 'd's and so on.

In the last call to **romanize( )**, the value returned is not collected, as we do not need the value that is returned.

Advanced Features of Functions

Let us see some of the advanced features of C as listed in the slide. We shall study them one by one. To begin with, let us discuss how to return a non-integer value from a function.

Returning A Non-Int Value

The program calls **square( )** to determine the squares of 2.0, 2.5 and 1.5. On printing the returned values we get 4.0 6.0 2.0. This is obviously wrong. However, when the square values are printed in the **square( )** function they turn out to be 4.0, 6.25 and 2.25. This means that when **square( )** tried to return a **float** value only the integer part of it got returned.

To return a non-integer value, the definition of **square( )** must be preceded by **float**. This would tell the compiler that the function is going to return a **float** value. We also need to specify the function prototype in **main( )** indicating that **square( )** is a function, which is going to receive a **float** and is going to return a **float**.

What's Wrong?

Since prototype of function **square( )** is not present in function **f( )** we cannot call **square( )** from **f( )**. The program on compilation gives an error.

The two ways to avoid this error is to mention the prototype of **square( )** in **f( )** or by declaring it above the **main( )** as indicated by the arrows in the slide.

What Would Be The Output

It seems that the second **printf( )** should print **10** as **f( )** is not returning any value. However, on execution it prints a garbage value. This means that by default any function returns a garbage integer value.

Solution 1

If we want that the second **printf( )** should also print 10 then do not collect the value returned by **f( )**. This function would continue to return the garbage value. Only thing is now we are not collecting the value anywhere.

Solution 2

Another way is to ensure that no value ever goes back from **f( )**. This is achieved by declaring its return type as **void**. **void** prevents a function from returning any value. Moreover, if we try to collect the value now, the compiler would flash an error.

Advanced Features of Functions

Let us now look at another advanced feature of functions—Call by Value/Call by Reference. To be able to understand this we first need to understand a concept called Pointers.

Pointers The Biggest Hurdle!!

In this lecture you will understand:

* What can we do with address of operator (&)
* How to use value at the address operator (*) to access values
* How to access VDU memory using pointers

Things Are Simple

Declaring a variable **i** of type **int** reserve space in memory to hold the integer value. The name **i** is associated with this memory location (say 4080) and **10** is stored at this location. This is shown in the memory map.

These locations in memory are termed as an address, reference, memory location or cell number but conventionally address is preferred.

The value of **i** is printed as **10**. **&i** prints **4080**, the address of **i**. It is clear from the memory map that it is the location where the value of **i** is stored. **&** is known as 'address of' operator as it retrieves the address.

***(&i)** prints value **10** stored at memory address of **i**. The operator * is called 'value at address' or 'indirection' operator.

Would This Work

j is assigned the address of **i**. The **&** operator can be used only with a variable, hence **j = &23** is invalid. It cannot be used with any expression or constant as in **j = &(i + 34)**.

(a) ***j** prints the value at address stored in **j**

(b) ***4568** prints value at address **4568**.

(c) ***(4568 + 1)** prints value at address **4568 + 1**.

These expressions are valid but needs some changes to work properly.

The Next Step

i contain a value **10,** its address (i.e. 4080) is stored in **j** and **j**'s address (i.e. 6010) is stored in **k.** The memory map of these three variables is shown in the figure.

i would print its value **10**. **&i** would print its address **4080**. ***(&i)** would print **10**, as seen in the earlier slide.

Now **j** is assigned address of **i**. **&j** would print **j**'s address **6010**. **j** would print its value **4080**. ***&j** would print value at address of **j** (i.e. ***(6010)**), which is **4080**.

Now **k** is assigned the address of **j**. Hence **k**, **&k**, ***k** and ***&k** would print **6010**, **5112**, **4080** and **6010** respectively.

The output for last **printf( )** would print 10 for all expressions. ***&i** means the value at address of **i**, which is **10**. ***j** means value at address stored in **j**, i.e. value at address **4080** which is again **10**. **&j** would give **6010**. ***&j** means value at **6010** which is **4080**. ****&j** means value at **4080** which is nothing but **10**.

Thus ****&j** would become ****(6010) which** in turn would become ***(4080)** which would give 10.

In Essence

This slide summarizes what we have learnt so far.

The statement **int *j** indicates that **j** is a variable that stores the address of another integer variable. In other words **j** is a pointer to an **int** or **j** is an integer pointer.

int **k indicates that **k** is a pointer to an integer pointer. **k** is storing the address of variable **j** which is holding address of another integer variable **i**.

l stores the address of **k**, which in turn holds the address of **j**, which holds address of an integer. Hence **l** is a pointer to a pointer to an integer pointer. On similar lines **m** is a pointer to a pointer to a pointer to an integer pointer.

& should not be used in the declaration. Using **&&** would make it a logical and operator.

Accessing Screen

Let's access the screen using pointers. The program shown in the slide intends to convert characters on the screen from uppercase alphabets to lowercase or vice versa.

A **char** pointer **s** is initialized with the memory address of 0^{th} row, 0^{th} column of screen, i.e. 100. The **for** loop executes the loop 2000 times (25 rows x 80 cols). The expression ***(s + i)** returns value at **(100 + i)**. In each iteration it checks for the character whether it is falling in the range of lowercase or uppercase alphabets. If it happens to be **'A'** then we are placing **'a'** at that address by adding the value 32 ('a' - 'A' = 32). If the character is **'a'** then to place **'A'**, we are subtracting 32 from the character. The whole code is repeated indefinitely through an indefinite **while** loop.

But on compilation the program won't work as excepted since the screen's base address is not 100 but **0xB8000000**. Also **s** should be declared as **far *.** We would discuss about **far** and **near** in next lecture.

Even after changing the address, the program shown in the slide won't execute properly. This is because each character on the screen is at an offset (distance) of 2 bytes from the previous character. So, the last character i.e. the character at 25^{th} row and 80th column is at an offset of 3998. Hence the loop should be repeated for 3998 times at a step of 2 beginning from the character at the base address.

More Pointers - I

In this lecture you will understand:

* How to create a program that directly accesses VDU memory and make characters fall to give a raindrop effect.
* How to play sound

To Make Characters Fall

Above 640 KB RAM, there are 2 blocks (block A and block B) of 64 KB each. While working in text mode all text displayed on the screen is written to the B block starting at address 0xB8000000. Each character present on the screen uses 2 bytes in B block. The first byte contains the ASCII value of the character, whereas the next byte contains the color of the character. Using this knowledge let us see a program that makes characters on screen fall.

In the program given in slide, in **main( )** two **far** pointers **s** and **v** have been declared. The pointer **s** stores the base address of the screen (**0xB8000000**) and **v** is used for making the characters fall.

As we know, for each character present on the screen there are two bytes in reserved in VDU (Visual Display Unit) memory. Thus, for **80** characters on each row there are **160** bytes in VDU memory.

The character present in 0^{th} row, 0^{th} column is first collected in **ch**. Next, through a **for** loop, the address of the next row, 0^{th} column is calculated and stored in **v**. At this address the value in **ch** is written.

Make All Characters Fall

If we want all characters present in 0^{th} row to fall one after another, then a slight modification is required as shown in the program given in the slide. Here, a loop for columns is added. Each time through this loop a character at 0xB8000000 + c * 2 location is collected in **ch**. Then through another **for** loop, which runs for row, the new position of the character is calculated in **v** and value in **ch** is placed in **v**. This moves the character from a column **c** of 0^{th} row to next row **r**, but same column **c**.

Any Screen Address

The code given in the slide shows how a character can be placed at a specified row and column. For example, to place a character at 10^{th} row and 20^{th} column, we must cross 10 rows (0 to 9). Every time we go to the next row we need to add 160 bytes (80 x 2). Similarly, to reach 20^{th} column we need to cross 20 columns (0 to 19), and to go from one column to another we need to add 2 bytes. Hence, while calculating address of position at which 'A' has to be displayed, we have multiplied 10 by 160 and 20 is multiplied by 2 and then added to the base address 0xB8000000. Once the address is generated the character 'A' is placed at that address.

In general, to display the character at the specified row column position, row number is multiplied by 160, column number by 2 and the result is added to the base address.

Bells And Whistles

If you execute the program shown in slide number 2 you would observe two problems. Firstly we do not get the effect of the characters falling down. Instead the characters seem to be just getting written in subsequent rows. Secondly, the activity happens very, very fast. A solution to this is given in the program given in slide.

To remedy these problems, after displaying the character the character above it is overwritten with space after a delay of 60 milliseconds by calling the **delay( )** function. As the characters fall down sound is produced through the **sound( )** function. The sound stops when **nosound( )** function is called. A delay is introduced so that the sound is heard for a little while.

As an exercise you can try for making the characters fall down randomly from the screen. For this use **random( )** function to generate a random row and column number. Initialize the random number generator through **randomize( )**. To be able to use this function **#include** 'stdlib.h' file.

For example, **random (30)** generates a random number between 0 to 29. If we want to generate a random number in the range 300 and 350 use **300 + random (50)**.

More Pointers - II

In this lecture you will understand:

* What are toggle keys
* How to switch On & Off Caps lock programmatically
* How to restart computer without physically depressing Ctrl + Alt keys

Are They Same

While declaring more than one **far** pointer **far** keyword should precede each pointer. Thus the statement,

```
char far *s, *v ;
```

declares **s** as **far** pointer whereas **v** simply a **char** pointer. The correct statement is given in the slide.

Also the declaration and initialization can be merged into one statement as shown in the slide. Note that here, **s** is a pointer and its type is **char far ***.

Why Two Bytes Apart

The character displayed on the screen, requires two bytes in VDU memory. In the first byte the ASCII value is stored and in the next byte (also called color byte) value representing the color of character is stored. One byte means 8 bits, thus the maximum value obtained by setting 1 in all bits (of color byte) would be 255.

The program given in the slide uses this knowledge to change the color of characters on screen. Here, through a **for** loop we have changed the color of character on screen. Note that the loop counter **i** in **for** varies from 1 to 3999, 1 to access the color byte of character at 0xB8000000 and 3999 because one screenful of characters need 4000 bytes (i.e. 80 x 25 x 2= 4000, i.e. 0 to 3999). The value of **color** is incremented by 1 and if it exceeds 255 then **color** is reset to 0. The internal **for** loop is enclosed within an outer **while** loop, which runs indefinitely, thus keeps on changing the color of characters on screen.

Caps Lock

Keys such, as Caps lock, Num lock, etc. are known as toggle keys. The status of the toggle keys is stored at address **0x417**. The status of Caps lock is stored in the 6^{th} bit. If this bit is **1** then Caps lock is on, whereas, if it is **0** then the caps lock is **off**.

If we are to set the caps lock bit **on** we must write a value **01000000** (i.e. 64) at **0x417**. The program given in the slide does this. To put off the caps lock store a value 00000000 at **0x417**, or simply hit the caps lock.

Don't Do Delete

The bit number 2 and bit number 3 of the 0x417 byte store the status of Ctrl and Alt keys. If we store 00001100 at 0x417 these keys would be considered to be depressed.

In the program given in the slide, we have stored 12 i.e. 00001100 in 0x417. On executing the program if Del key is pressed, the machine restarts because even though we have not physically depressed the Ctrl and Alt keys, their bit values at 0x417 are on.

Near Far

In this lecture you will understand:

* What is *far* and *near* pointer
* What is **segement:offset** addressing scheme

Why *far* and *near*

In the program given in the slide a pointer variable **kb** is declared as **far** whereas rest of the pointer variables are declared as **near** (the default). The reason for this is explained in the next few slides.

Internal Details

The microprocessor is the heart of all the PCs. The microprocessor is also called Central Processing Unit, or simply CPU.

A CPU consists of an Arithmetic Logic Unit to perform arithmetic and comparisons, Control Unit to control other units of computer and Memory chips to store information. The microprocessor usually consist of Arithmetic and Logical Unit and Control Unit. So microprocessor may not be complete CPU in itself. But with wide spread use of microprocessors, they have come to be called CPUs.

The CPU controls the computer's basic operation by sending and receiving control signals, memory addresses, and data from one part of the computer to another using Data lines, Address lines and Control lines respectively. The control signals, address and data are carried from one part to another through a '**bus**'. A bus refers to a group of interconnecting electronic pathways (or set of wires) that connect one part of the computer to another.

Each wire in this bus carries a bit of data at a time (either '1' or '0' in the form of an electronic pulse). If the data bus has 8 wires then 8 bits data bus, similarly we have 16-bit and 32-bit data buses that can carry 16 and 32 bits at a time respectively. A MP with a provision for 16-bit data bus is called a 16-but microprocessor. Microprocessor consists of registers in it, its size depends on the data bus size (in our case 16 bit wide).

The way the data bus width tells how many bits the bus can move at a time, there is another bus called address bus whose width tells how many addresses the microprocessor can access. For example if the address bus width of MP is 20 bits then it can access 2^{20} locations (1 MB) in memory.

To access any of the 2^{20} memory locations the MP uses 16-bit registers. However in 16 bit registers the maximum value that can be stored is 65536. In example shown in the slide variable **i** is assigned with value **3**, this gets stored in the memory at the location 0AB00. When we access this variable through the data lines, address lines and control lines 3, 0AB00 and 1 are passed to CPU respectively. Internally it is done using Segment and offset addressing scheme. This is discussed in detail in the next slide

Segment:Offset Scheme

To access any of the 2^{20} locations the MP use 16-bit CPU registers. However in 16 bit registers the maximum value that can be stored is 65536.We can access memory location beyond 65535^{th} byte by using two registers (segment and offset) in conjunction. For this total memory (1MB) is divided into a number of units each comprising 65536(64kb) locations. Each such units is called segment.

2^{20} = 1MB,

2^{16}= 64 KB,

Therefore, number of units = 1 MB /45 KB = 16 units (segments).

Each segment always begins at a location number, which is exactly divisible by 16(16 = 10h, 32 = 20h, 48 = 30h etc.). The segment register contains the address where a segment begins, where the offset register contains the offset of the data/code from where the segment begins. The segment address is 00040h (20-bitaddress)

Here 8000h (16 bit address) can be easily placed in offset register (Reg1), but how do we store the 20-bit 00040h address in 16-bit segment register? What is done is out of 00040h only the first four

hex digits 0004h (16 bits) are stored in segment register (Reg2). DOS can afford to do this because a segment address is always a multiple of 16 and hence always contains a 0 as the last digit. Therefore, the first byte in memory is referred using **segment : offset** format as 0040h: 8000h. Thus, the offset register works relative to segment register. Using both these, we can point to a specific location anywhere in the 1MB address space. If we want to store any variable in this address then internally the segment register gets shifted by 4 bit left and the offset address is added to it forming 08040h.This number is then passed on to address bus, to locket the variable in the memory.

Near and Far

A near pointer is 16 bits long. It uses current contents of the CS (Code Segment) register (if the pointer is pointing to code) or current contents of DS (Data Segment) register (if the pointer is pointing to data) for the segment part where as the offset part is stored in the 16 bit near pointer. Using near pointer limits your data/code to current 64KB segment.

The program shown in the slide has **i** as global variable and gets stored in DS. All the local variable **j, p, q** and **kb** are stored in SS. Since **kb** points to a location outside the data segment we need to declare it as a **far** pointer.

Far Huge

In this lecture you will understand:

* What are huge pointers
* The size of *far* and *near* pointer
* How to find the size of base memory

Far Pointers

Far pointer (32 bit) contains the segment as well as the offset. By using far pointer we can have multiple code segments, which in turn allow you to have programs longer than 64 KB. Like wise, with far data pointers we can address more than 64 KB worth of data. However while using far pointers some problems may crop up as shown in the slide. Note that both 32 bit addresses stored in variables **s1** and **s2** refers to the same memory location, we expect the **if** to be satisfied. However this doesn't happen. This is because while comparing the **far** pointers using == the full 32 bit value is used and since the 32 bit values are different the **if** fails and the **else** part gets executed as a result, prints **Bye.**

This limitation can be overcome if we use **huge** pointer instead of **far** pointers. Huge pointer is explained in the next slide.

Huge Pointers

Unlike far pointers huge pointer are 'normalized' to avoid these problems. Normalized is a 32 bit pointer which has as much of its value in the segment address as possible. Since a segment can start every 16 bytes, this means that the offset will only have a value from 0 to F (0 to 15). Normalizing is done by converting the pointer to its 20-bit address then uses the left 16 bits for the segment address and the rightmost 4 bits for the offset address. For example, pointer 500D:9407, we convert it to the absolute address 594D7, which we normalize to 594D:0007. Huge pointers are always kept normalized. As a result, for any given memory address there is only one possible huge address segment: offset pair.

Pointer Sizes

The program given in the slide illustrates that the size of a **near** pointer is always **2** bytes, whereas, size of a **far** pointer and huge pointer is always 4 bytes.

How Much Memory

The size of base memory is stored in the BIOS Data Area at locations 0x413 and 0x414. The program given in the slide reads this location and displays the amount i.e. size of base memory. Since, BIOS Data Area lies outside data segment, pointer **m** in the program is declared as a **far** pointer and is assigned the address 0x413. Then ***m** prints the actual value of base memory stored at given location. Generally it is 640. If it is less than 640, then it is a sure shot sign of a virus present in the memory.

Call By Value & Reference

In this lecture you will understand:

* What is the difference between call by reference and call by value
* How to return more than 1 value from a function

What would be the output

In the program given in the slide, **j** is an **int** pointer, **b** is **float** pointer and **dh** is **char** pointer. Also, **k** is a pointer to an integer pointer. Similarly **c** is a pointer to a **float** pointer and **eh** is a pointer to a **char** pointer. Observe carefully the figure given in the slide.

The pointer **j** stores address of **i**, **b** stores address of **a** and **dh** stores address of **ch**. Similarly, **k** a pointer to an **int** pointer stores address of **j**, **c** stores address of **b** and **eh** stores address of **dh**.

In the first **printf( )** we have displayed the values in pointers **j**, **b** and **dh** ,which happens to be the address of corresponding variables. In the second **printf( )** we have displayed values in pointers **k**,**c** and **eh** which happens to be the addresses of pointer **j**,**b** and **dh** respectively.

In the next **printf( )** we have displayed the size of variables **i**, a and **ch** which happens to be **2**, **4** and **1**,the size of an **int**, **flaot** and **char** respectively.

Slide Number 2

The size of any pointer is always 2 bytes. Hence on printing the size of pointers **j**, **b** and **dh**, **k**, **c** and **eh** the output for each gets displayed as 2.

**k becomes *(*(4000)) becomes *(1000) becomes 10

Hence ****k** prints 10 and similarly ****c** and ****eh** prints 3.14 and **z** respectively.

Slide Number 3

The program given in the slide tries to exchange the values of two variables through a function. In the program, **a** and **b** are two integer variables with values 10 and 20 respectively. Their values are printed and then a function **swapv( )** is called. To this function we have passed the values of **a** and **b**. This way of calling a function is known as call by value.

The function **swapv( )** receives the values of actual parameters in formal arguments **x** and **y** respectively. Then **t** is used as a temporary variable and being not initialized holds a garbage value. The initial values are shown in the memory map.

Initially the value of **x** is stored in **t** i.e. **t** contains 10. Next the value 20 of **y** is stored in **x**. Now **y** is assigned the value of **t**, which is originally the value 10 of **x**. On printing the values of **x** and **y** 20 and 10 gets printed, which are the interchanged values.

But as soon as the control returns to **main( )**, **x** and **y** die and are not available in **main( )**. Hence, printing the values of **a** and **b** again results in printing 10 and 20 respectively.

Slide Number 4

The program given in the slide is similar to the one discussed in previous slide but with a slight difference. Here, instead of passing value to the function we have passed addresses of variables. This way of calling function is known as call by reference.

The addresses of **a** and **b** are collected in **swapr( )** in **int** pointers **x** and **y** respectively. In **t** we are storing the value at address stored in **x** i.e. **10** as **x** holds the address 200 of **a**. Next the value at address stored in **y** (i.e. 20) is stuffed at address stored in **x**. Then the value of **t** is assigned to ***y**. Back in **main( )** the **printf( )** statement prints the interchanged values of **a** and **b**, i.e. as 20, 10.

Slide Number 5

We know that a function cannot return more than one value. The program given in the slide shows how to return more than one value through a function.

Here we have called **sumprod( )** with 5 parameters, values of variables **a**, **b** and **c** and addresses of variables **s** and **p** respectively. In the function we have calculated the sum of **x**, **y** and **z** and is placed at an address stored in **ss**, i.e. in **s**. Similarly, we have calculated the product of **x**, **y** and **z** and is placed at an address stored in **pp**, i.e. in **p**. Now, on returning back to **main( )**, **printf( )** would print the sum and product of variables **a**, **b** and **c** . In case of function called by reference, we can return more than one value.

Recursion

In this lecture you will understand:

* What is meant by recursion
* How to find factorial value of a number using recursive calls

Advanced Features of Functions

Now let us move on to the next feature of functions, the Recursion.

Simple Form

Recursion means a function that calls itself. The corresponding slide shows a simple example of recursion. From within the **main( )** function there exists a call to the same function. On execution, the program falls in an indefinite loop thereby printing 'Hi' infinite number of times.

In the second example **f( )** is being called recursively. The moral is that any function can call itself thereby causing a recursion.

More General

The program given in the slide calculates the sum of individual digits of any given number (i.e. a valid integer containing any number of digits).

The program accepts a number entered through keyboard, and calls the **sumdig( )** function and passes the number to it. The **sumdig( )** function runs a **while** loop until **n** becomes zero. In every iteration the last digit of the number is extracted using **n % 10** and stored in **d**. Then **d** is added to **s**, which is initially set to zero. The digits thus extracted is removed from the number itself using the statement **n = n / 10**. The process continues until **n** becomes **0**. Finally the value of **s** is returned to **main( )**, where it is stored in **sum** and printed out.

Slide Number 4

Let us now find out the sum of individual digits of a number using recursion.

In the program given in slide, the **sumdig( )** function is replaced with the **rsum( )** function. In the **rsum( )** function instead of a **while( )** loop a recursive call is made until **n** becomes zero. Before each recursive call the value of **n** is checked using **if-else** statement. The answer is then returned to **main()**.

Now the complexity lies in the following statement:

```
s = d + rsum ( n ) ;
```

The detailed explanation of this program is given in the next slide.

Slide Number 5

Let us understand the working of **rsum( )** function by considering **n** as **327**. Now when **rsum( )** is called for the first time with value **327** the control enters the **if** block since **n** is not zero. The last digit **7** is extracted and stored in **d**. Then the value of **n** is reduced to **32** by dividing it by 10. The very next statement,

```
s = d + rsum ( n ) ;
```

which tries to add **d** to the number returned by **rsum( )**. But before adding **d** a call to **rsum( )** is again made but this time with value **32**. Since we have removed the last digit form the number 327,**n** becomes 32.

In the second call to **rsum( )** function, **2** is extracted and stored in **d**. (Since **n, d** and **s** are local to the **rsum( )** function, each call to **rsum( )** creates three new variables.) Then the value of **n** is reduced to **3**. The digit **d** i.e. **2** should then get added, but before that, a call to **rsum( )** is again made value **3**.

Now **3** is extracted and the value of **n** is reduced to **0**. The digit **3** should get added, but again first a call is made to **rsum( )** with value 0. This time the **if** condition fails and **0** is returned back from where the call was made. The previous call was made from the statement:

```
s = 3 + rsum ( 0 ) ;
```

Hence this statement becomes:

```
s = 3 + 0 ;
```

s becomes **3**, which is returned back to the previous function from where this function call was made. The same procedure is repeated in every previous call where a new value of **s** is calculated. When all calls are resolved **s** ultimately contains the sum of individual digits.

Factorial Value

The program given in the slide uses a function to find out factorial value of a number passed to it.

In the **factorial( )** function the number is collected in **n** and a **while** loop is run until **n** becomes zero. The factorial value is calculated by multiplying **n** with **p**. In each iteration **n** is multiplied with **p** and then it is decremented by **1**. Hence in each iteration a decremented value of **n** is multiplied with the previous value of **p**. If **n** is **3** the resultant factorial value in **p** would be **6**.

Recursive Factorial

In the program given in slide, the factorial value is calculated by calling the function recursively instead of using a **while** loop. The function name is changed to **refact( )**.

The **refact( )** function is called recursively until **n** becomes **0**. In each call before multiplying the current value of **n**, a call is made to **rfact( )** function with a reduced value of **n** (reduced by 1). This continues till **n** is not zero. The moment **n** becomes 0, 1 is returned from where the function was called. The returned value is multiplied with the previous value of **n.** The resultant answer is stored in **p**, which is returned to the previous call. Ultimately **p** holds the factorial value that is returned to **main( )**.

Note the tips, which you should keep in mind while writing code for a recursive function.

From the developer's point of view, writing a recursive function is not at all easier. From the efficiency point of view, recursive functions occupy more space as well as take more time to execute. More space and time, because in each call all local variables are recreated.

Slide Number 8

The slide shows the game 'Tower of Hanoi'. If a program for this game has to be written then there is no other way than recursion. The rings are to be moved from peg A to peg C and they should be placed in the original order as it was in A. The movements of these pegs are shown in the slide. The problem is left to you as an exercise.

Data Types

In this lecture you will understand:

* Data types, their size and maximum and minimum range
* The type of integers, **float** and **char** data types

Data Types In C

The data types we have worked with so far are **int**, **char**, and **float**.

Type of Integers

The data type **int** is classified as a **short int** and a **long int**. The statement **int i ;** is equivalent to **short int i ;** Thus, the keyword **short** is optional. An **int** reserves **2** bytes (16-bits) of memory, of which most significant bit is reserved for sign and remaining 15-bits store the value. If sign bit is **0** then value stored in remaining bits is positive and if it is **1** then negative. When we write **int i ;** it is treated as a **signed** value. Thus the maximum positive value of a **signed short int** with all **1**'s in value bits is **32767** and minimum value is **-32768**.

The data type **short int** can also be **unsigned**. In this case most significant bit is not treated as sign-bit. All bits are used to store a value. Thus the range of **unsigned short int** is from **0** to **65535**.

Type of Integers

The data type **long int** is classified as **long signed int** and **long unsigned int**. The **long int**, reserves **4** bytes (32-bits) in memory of which the most significant bit is reserved for sign. Hence the maximum value it can accommodate is **2147483647**. For **long unsigned int** the range is **0** to **4294967295**.

Types of Chars

The **char** data type is classified as **signed char** and **unsigned char**. A **char** reserves **1** byte (8 bits) in memory. Again the most significant bit is reserved for sign hence remaining 7 bits are used to store value. Thus, the maximum value it can store is **127**. The valid ranges for **signed** and **unsigned char** are shown in the slide.

Types of Real

The real value in C is classified as—**float**, **double** and **long double**. The valid ranges allowed, the size in terms of bytes and format specifiers that are used for these data types are shown in the slide. Real types are always signed.

Three Questions

When we declare a variable as **char**, then by default it is treated as **signed char**. In the first statement, 'A' gets stored in **ch**, but in memory its ASCII equivalent (an integer 65) gets stored. This notifies that even an integer can be stored in a **char** variable, whether positive or negative. Hence the second statement in the slide is perfectly valid.

The **printf( )** statement given in a code snippet in the slide display **-128**. This is because the maximum value that a **signed char** can store is **127**. The number **128** exceeds this range. On exceeding the range, it moves to reverse side of the range, that is from right to left (positive to negative) thereby generating number **-128** which is the lowest number a **signed char** can store. Now a question arises as why we have a bias of these ranges from minimum to maximum. This is explained in next slide.

Slide Number 7

When **–128** is stored in **ch**, first the binary equivalent of **128** is determined. Then it's 1's complement is extracted, which is 01111111. The 2's complement is then obtained from this value. The value thus formed would be **-128**.

When 128 is stored in an **unsigned char**, then it is only converted to its binary equivalent and is then printed.

Various Forms

The examples given in the slide shows the various forms of declaring variables.

What Is 365? & What Is 3.14?

In this slide, the default type of constants is explained. For 365, the type would be **int**. To make it **long** add suffix **L** or **l**. For **unsigned** add **u** and add suffix **lu** or **ul** for **unsigned long** .

On the other hand by default real constant say 3.14, is treated as **double**. To make it **float** add suffix **f** and to make **long double** add suffix **L** as shown in the slide.

Storage Classes

In this lecture you will understand:

* What are different storage classes
* Features of variable defined to have automatic storage class
* Features of variable defined to have register storage class
* Features of variable defined to have static storage class
* Features of variable defined to have external storage class

Storage Classes In C

Let us see another feature of C, the Storage Classes. The definition of a variable consists of its type and storage class.

A storage class signifies four things:

(a) Storage tells where the variable would be stored.

(b) The default initial value that a variable would hold if initial value is not specifically assigned.

(c) The scope of a variable i.e. in which functions the value of variable would be available.

(d) Life of the variable i.e how long would the variable exist.

There are 4 types of storage classes as shown in the slide. When a variable is declared without any storage class, an automatic storage class is assumed by default.

Automatic Storage Class

The variable defined to have an automatic storage class, gets stored in memory, and holds garbage value as the default initial value. The value of variable is accessible in the block in which it is defined, i.e. the variable is local to block. The life of the variable is till the control is within the block where it is defined.

Consider the program given in the slide. Here, the variable **a** is declared as an **int** without any storage class, **b** is declared as an **int** with storage class as **static** and **c** is declared as an **int** with storage class as **automatic**. The contents are printed through the **printf( )** statement. But on execution an error 'Undefined Symbol automatic' would get flashed, as there is no such keyword. To declare an automatic variable use **auto** keyword, and then check out the output?

Now the program outputs garbage values for **a** and **c** and 0 for **b**. This clarifies that the initial value for a variable with no storage class as well for **auto** variable is always a garbage value. The **static** variable holds **0** as its initial value by default (if not initialized).

Scope & Life

The slide explains that no two variables can have same name within the same block even with different data types. Hence both the program segments shown in the slide would flash an error message as 'Redefinition not allowed'.

Look at the program shown in the slide. The variable **a** is declared within three different blocks with values 10, 20 and 30. On compilation, the program gets compiled successfully and on execution prints 30, 20 and 10. All the three variables used in the program hold same name, still the program works how? The error would be flashed when the variables with same name are declared within the same block.

However, if you try to execute the program given in a box in the slide, then would flash errors. This is because the variable **a** is defined in function **main( )** (i.e. within the **main( )**'s block). So, it would be available to **main( )** only and not to the function **f( )**'s block.

Death, But When?

In the program given in the slide, the variables **i**, **j**, **k** and **l**, declared in **main( )** won't die when the control goes to **f( )** as the block is not completed, whereas, variables **m**, **n** declared in **f( )** dies as soon as the control is returned or the block terminates.

Register Storage Class

A variable defined with register storage class, gets stored in CPU registers. All other attributes except storage, remains same as that of automatic storage class. Central Processing Unit (CPU) consists of microprocessor (µp). Intel's 8086 family of microprocessors has 14 CPU registers, each of 2 bytes in size.

Be Judicious

Both the programs given in the slide are same except the declaration of **i**. In the first one, the variable **i** is declared with storage class as **auto** and in the next one **i** is declared with storage class as **register**. On execution, the first program would take longer time than the second one, as second one uses CPU registers to access data. A value stored in a CPU register can always be accessed faster that the one that is stored in memory.

The other attributes of register storage class are listed in the slide.

It must be carefully decided which variables should be declared with register storage class. In the program given in slide, **i** and **j** are declared with register storage class. **i** is used as a counter in the **for** loop, which needs frequent access for incrementing, hence needs faster execution. Hence the judgment to declare it as a **register** is correct. But the variable **j** is used only once in the program. Such variables should not be declared as **register**, since we have only 14 registers.

Static Storage Class

The variables declared with a **static** storage class get stored in memory, and holds **0** as default initial value. The **static** variables are also local to the block in which they are defined. But, the **static** variable persists (retain its value) between different function calls.

When to Use Static

Consider the program given in the slide. In this program **p** is an **int** pointer and **f( )** is a function that returns pointer to an **int**. The function **f( )** returns the address of variable **a**, which is local to the function block of **f( )** and **a** would die the moment the function is over. In such a situation **p** would be collecting address of variable that does not exist. To solve this problem we can define the storage class of **a** as **static**. This is because, value in static variable is kept in memory when it is not active, i.e. the variable takes up space in memory. Moral is, when we have to return a pointer from a function then use **static** storage class.

External Storage Class

The program given in the slide demonstrates use of an external storage class. Variables declared with an external storage class are the global variables, which are declared outside all the functions.

Here, in the program, **a** is an external (global) variable. The **printf( )** statement in **main( )** prints the value of **a** as **10**. Then function **increment( )** is called in which **a** is incremented by **1** and its value is printed as **11**. Note that **a** is not declared in **increment( )**, the global variable **a** gets used. Again by calling **increment( )**, the value of **a** becomes 12, since **a** is global variable. And on calling **decrement( )**, its value is decremented which becomes 11.

Declaration V/s Definition

All the statements and functions in this slide are similar to the ones given in the program discussed in the earlier slide, except the declaration for **a**, which is done at the end below all the functions.

The statement **extern int a ;** given in functions like **main( )**, **increment( )** and **decrement( )** is called as a declaration, as no memory is reserved. It simply tells that variable **a** is an external (global)

variable which is defined somewhere else. Here it is defined at the end of the program by the statement **int a = 10 ;**.

If we execute the program without inserting the statement **extern int a ;**, then it would flash errors. This is because **a** is getting used in functions **main( )**, **increment( )** and **decrement( )**. Even though **a** is defined in the program, since it is defined below the program it would not be available to **main( )** and other functions. To make it available we must add a statement as **extern int a ;** in all these functions. Saying **extern int a**, is simply declaration of variable. The variable is said to be defined when memory is reserved for it. The variable **a** is defined through the statement

```
int a = 10 ;
```

we say that the variable is defined. This kind of declaration and definition is similar to the function declaration and the definition of the function.

Declaration V/s Definition

The program given in this slide is similar to the one given in previous slide, but with slight modifications. The statement that defines variable **a** is placed just before **decrement( )** function. Then the statement that declares **a** as an **extern** can be dropped from **decrement( )** function.

Declaration V/s Definition

This slide too contains the same program again with slight modifications. This time the statement that define global variable **a** is placed above definition of function **increment( )**. Then the statements that declares **a** as an **extern** can be dropped from **increment( )** and **decrement( )** function.

Thus concludes that that use of **extern** inside a function is optional as long as we declare it outside and above that function in the same source file.

Two Types Of Conflicts

Consider program given in the slide. Here, **a** is defined as a global variable with a value **10**. Then in **main( )**, **a** is again defined as a local variable with value **20**. Furthermore, in the inner block, **a** is defined as a local variable with value **30**.

Now the conflict is which value is printed through the **printf( )** statement of the innermost block? It prints value of **a** as **30**, since **a** with value as **30** is most local and hence gets the priority. In absence of the local variable, the value of variable defined in the outer block would have been printed.

The next **printf( )** statement as explained in the previous case would print the value of **a** as **20**.

Note that it is not allowed to write any executable statement after the closing braces of a function or **main( )**. Only declaration or initialization statements can be written.

Which is The Most Powerful

Dennis Ritchie has made available to the C programmer a number of storage classes with varying features, believing that the programmer is in a best position to decide which one of these storage class is to be used when. This slide briefly explains when and which storage class is the best suited.

Preprocessor - I

In this lecture you will understand:

* What is meant by compiling a program
* What is meant by linking
* What is a preprocessor
* Various types of preprocessor directives
* Various preprocessor directives and their use
* What are macros
* What is the difference between macro templates and macro expansion

Compilation Options

The shortcut keys used for compilation, execution and debugging of a program are given in this slide along with the function they perform.

Slide Number 2

You can trace the control flow, means the order in which the statements get executed, by using function key F7. Initially on pressing F7, the control would get placed over **main( )**. Pressing F7 again, would transfer control to the next statement. When the control comes on statement that calls function **display( )**, the control would get shifted to the body of function **display( )**. (If you do not want the control to trace the statements inside the function body you can press F8 when the control is on the function call). Further pressing of F7 key would execute the function line by line. On execution of the last statement of the function the control would return back to the **printf( )** statement following the call to **display( )**. The execution would get terminated the moment control reaches the closing braces of the function **main( )**.

This sort of tracing is also called debugging.

Slide Number 3

If you want to watch the value of a variable while debugging, press Ctrl + F7 by placing the cursor below that variable. The variable would get added in the watch window. Start debugging and the watch window will show the value of the variable.

A Closer Look

How a C program is compiled? A C program written using any text editor is called a C source code. The text editor helps to type and edit text of the program. The preprocessor directives (**#include**, **#define** etc.) in the source code get replaced with the corresponding code thereby expanding the source code by the preprocessor. The compiler then converts the expanded source code into machine code. The object code thus formed is then linked with the object code of the standard library through the linker, which results in to an executable code. The object code (.OBJ) and executable code (.EXE) files are placed in the output directory. Generally named as the **WORKS** directory.

What is meant by IDE and why it is used? IDE means Integrated Development Environment. IDE like Turbo C/C++ includes an editor that helps to create text files, a preprocessor to expand source code, compiler to convert source code object code, linker to link object code of source with that of the standard library file. In Turbo C/C++ IDE all the functions like preprocessing, compiling, linking and creating executable are integrated and done in one shot by pressing Ctrl + F9.

Types of Preprocessor Directives

There are four types of preprocessor directives as listed in the slide.

Preprocessor In Action

The program in the slide explains how a preprocessor works. In the program **#include** preprocessor directive is added to include a file called 'goto.c'. The file 'goto.c' contains the definition of function **gotorc( ).** On pressing Ctrl + F9, the preprocessor replaces the code of the file 'goto.c' in place of the **#include** directive in the source code, which is thus the expanded source code. The source code of file is say 'pr1.c', then the expanded source code becomes 'pr1.i'.

The expanded source code on compilation result the object code in machine language, which consists of the machine language form of **gotorc( )** and **main( )** function. If you execute this object code (.OBJ) you will get errors as unresolved externals. Why these errors? This is explained in next slide.

Linking

The errors are due to non-availability of the code for functions like **clrscr( )**, **printf( )**, etc. The machine code of **gotorc( )** and **main( )** is then linked with machine codes of **clrscr( )** and **printf( )** through the linker. After linking an executable file say 'pr1.exe' gets created, which would get executed successfully.

Your Wish

There are two ways of including a file in a program. When the name of the file is enclosed in double quotes, then it is searched in both the current directory as well as the specified list of directories as mentioned in the include search path. The include path in Turbo C/C++ IDE is available in the 'Options' menu's 'Directories' item as shown in the slide ("Alt o" shortcut key).

When the filename is enclosed within a pair of angle brackets (<>), then the file is searched only in the specified list of directories (i.e. include path).

Macro Expansion

Let's now understand preprocessor directives in detail. The macros are defined using **#define** preprocessor directive.

In the program given in the slide, LOWER and UPPER are called 'macro templates', whereas, 1 and 10 are called their corresponding 'macro expansions'.

These macros are used as lower and upper bounds in the **for** loop. On compilation, the **for** loop would get expanded as shown in the slide after preprocessing. The macro name has been replaced with their expansions 1 and 10.

Preprocessor -II

In this lecture you will understand:

* How to create macros with arguments
* When do we need condition compilation
* How to make conditional compilation possible
* Various miscellaneous preprocessor directives

Macro Expansion

In this slide a macro **PI** for the value of pie is defined and used while calculating the area **a** of the circle. On preprocessing the statement for calculating the area would be expanded as shown in the slide.

Instead of this we could have declared a variable for VALUE OF pie. Is there anything wrong with it? Even though 3.14 is such a common constant that it is easily recognizable, there are many instances where a constant doesn't reveal its purpose so readily. Secondly, if a constant like 3.14 appears many times in the program. This value may have to be changed some day to 3.141592. Ordinarily you would need to go through the program and manually change each occurrence of the constant. However, defining PI in a **#define** directive, only needs to make one change, i.e. in the **#define** directive itself as shown below:

#define PI 3.141528

Beyond this the change will be made automatically to all occurrences of PI before the beginning of compilation.

Don't Remember Constants

Remembering long constants is difficult. However, the names used for such constant are easy to remember. Hence use **#define** for such constants. In the slide the sketch of the program is shown that use the constant **PLANK**. In the last statement instead of **PLANK**, simply **PLAN** is written which would be detected while compiling the code. On the other hand if the constant contains any mistake, that cannot be identified by the preprocessor or compiler. Hence using Symbolic constants is safer.

Macros are of two types:

- Simple Macros
- Macros with Arguments

We have already discussed simple macros.

Macros With Arguments

The program given in the slide makes use of both, simple and macro with arguments. The macro **PI** is a simple macro and **AREA(x)** is a macro with arguments. While defining a macro with argument space should not be present between the macro name and the opening parentheses.

The statement,

```
a = AREA( r ) ;
```

would be expanded as,

```
a = 3.14 * r * r ;
```

The statement,

```
a = area ( r ) ;
```

would call the function **area( )**.

Macros are faster than function. This is because, in a macro call the preprocessor replaces the macro template with its macro expansion. As against this, in a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function. This process takes time and would therefore slow down the program. Any number of variables can be used in the argument as shown in the slide with **A (x, y, z)**.

Macros With Arguments

In the slide a macro **S** with arguments to determine the square of a number is defined. In the program several calls are made to it. The statement,

```
i = S ( 4 ) ;
```

would be expanded as

```
i = 4 * 4 ;
```

It's all right but if expressions such as 2 + 2, 3 + 1, 1 + 3, or ++n are used, then they would be expanded as shown in the slide and would give wrong results.

We can see the expanded source code using 'CPP.exe' command (present in TC\BIN directory). To do so, save your program and quit Turbo C. Change directory to C:\TC\BIN and run the command as shown in the slide. A file with extension '.I' would be created which contains the expanded source code. Using DOS command 'Type' you can view the contents of '.I' file.

Solution

Now the solutions to the problem that discussed earlier is given in this slide. The solution is either the parameter should be placed within the parentheses while calling the macro or the parameters in macro expansion should be placed within parentheses as shown in the slide.

Conditional Compilation

Sometimes we need to compile some part of the code and discard some other code. What if the **scanf( )** statement shown in the right side of the slide should be used based on a condition? This is shown in the left side of the slide.

To compile the code conditionally, the preprocessor directives **#ifdef** and **#endif** are used as shown in the slide. The preprocessor directive

```
#ifdef YES
```

read as 'if YES is defined through **#define** then compile the code till **#endif** is encountered'. Hence as a solution define YES above the **main( )** so that the code would be compiled. Otherwise to compile the code simply delete **#ifdef** and **#endif** preprocessor directives.

Miscellaneous Directives

In the program given in slide, YES is defined as a macro before **main( )** with value 10 and accessed in the first statement of **main( )**. Next, it is redefined as 20. Now on the usage it will be replaced with 20. This tells that redefinition of a macro is allowed.

Macros cannot be used within the double quotes, if used it won't get replaced with macro expansion. The macro can be undefined through the preprocessor directive **#undef**. In the slide YES is undefined, which undefined macro YES. The macro name cannot be used to declare variables, as their scope is global.

#pragma

#pragma is another miscellaneous directive. **#pragma** directives are special compiler directives. **inline** allows writing assembly language instructions within the C source code as shown in the slide. In this case the program cannot be compiled or traced through the keys shown in the slide.

Arrays

In this lecture you will understand:

* What are arrays and how to define them
* How to declare and initialize array
* How elements in an array are stored

How Much C

The keywords related to data types, control instructions, and storage classes that we have covered so far are listed in the slide.

Arrays

Let us now move towards the advanced features of C— Arrays.

To calculate percentage marks of 10 students for three subjects, three variables **m1**, **m2**, **m3** are declared. And through the **for** loop marks of 10 students are accepted and their percentage is calculated.

But if the percentage of say 3rd student is required, it cannot be made available once the **for** loop is over. The last **printf( )** statement prints the percentage of last student. So how can we retain the earlier calculated values?

Choices Available

The choices available for handling case discussed in earlier slide are:

- Use the number (10) of variables to hold each value
- Use one variable, capable to hold the specified number of values

Obviously the second alternative is better. A simple reason for this is, it would be much easier to handle one variable than handling number of different variables. Such a single variable is called an array. An array is a collective name given to a group of 'similar quantities'. These similar quantities could be percentage marks of 100 students, or salaries of 300 employees, or ages of 50 employees. What is important is that the quantities must be 'similar'.

Nothing Different

Each member in the group i.e. array is refereed to by its position in the group. For example, consider the group of percentage marks obtained by 10 students given in the slide.

If we want to refer to the 1st, 3rd, 6th and 10th number of this group, then the usual notation used (as a subscript as shown in the slide) is per_1, per_3, per_6 and per_{10} respectively. In general the i^{th} member can be referred as per_i. However, using this form the value cannot be printed on the text screen. Hence the subscript should be written in the parentheses as **per (i)** or in square brackets as **per[i]**. In C square bracket notation is used.

Slide Number 5

The program given in the slide the same program that calculates percentage marks for 10 students, but is written with the use of an array **per**. While declaring array variable the number of elements it can hold is mentioned within a pair of square brackets following **per** as **per[10]**. In the **for** loop the percentage of each student is calculated and stored in the i^{th} location of **per** through **per[i]**. Lastly all the percentages of the students are printed through another **for** loop.

As the row and column position of screen begins with 0, in C, the counting of elements in an array also begins with 0 instead of 1. Hence, lower and upper bounds of array used in **for** loop should be subtracted by 1 (i.e. they should be 0 and 9 respectively).

Initializing Arrays

Observe carefully the program given in the slide. Note the way arrays **a[ ]**, **b[ ]** and **c[ ]** have been declared. If the array is initialized where it is declared, mentioning the dimension of the array is optional as done in case of array **a[ ]** (in the program given in the slide). If the array is not initialized

where it is declared then mentioning its dimension is compulsory. While initializing, it is allowed if the number of elements used for initialization is less than the dimension given for the array.

In the program, after declaration, the sizes of arrays **a[ ]** and **b[ ]** are printed. As **a[ ]** is initialized with only 5 integer elements the size is 5 x 2 bytes. As **b[ ]** is declared to hold 10 **int** elements, the size is 10 x 2. After this the first element of **a[ ]** and **b[ ]** are printed which prints 7 and a garbage value as **b[ ]** is not initialized. Next, values the higher elements for **c[ ]** are accepted using **scanf( )**.

An element of an array can be initialized or assigned to with an expression as evident from the last two statements of the program.

Moral

This slide lists a few points about an array, which have been confirmed by the programs of array discussed so far. Let us now see how arrays are different from normal variable.

Storage

Arrays are different with respect to the normal variables in respect of storage in memory. Normal variables on declaring are stored at discrete locations as shown in the slide.

On the other hand, arrays in memory are stored in adjacent or contiguous locations. The program given in the slide confirms this, where the addresses of array elements are printed using the **printf( )** statement within the **for** loop. Thus, an array is a collection of similar elements and these elements are always stored in adjacent memory locations.

Bounds Checking

In the program given in the slide, an array **a [ ]** contain 5 elements. But in the **for** loops, one used for some calculations on the elements of array, and the other to print the values, the upper bound used is 40. The upper bound used should be 4, but still the program works.

The program works because in C there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array, probably on top of other data, or on the program itself. This will lead to unpredictable results, to say the least, and there will be no error message to warn you that you are going beyond the array size. In some cases the computer may just hang. Thus, to see to that we do not reach beyond the array size is entirely the programmer's botheration and not the compiler's.

So...

Finally the points regarding arrays that we have seen so far are listed in the slide.

Sorting

In this lecture you will understand:

* Various sorting techniques

Selection Sort

To arrange the elements of a list in a particular order (ascending or descending) is called as Sorting. There are various sorting methods. Of these, a method called Selection Sort is used in the program given the slide to arrange elements of array.

In this method, to sort the data in ascending order, the 0^{th} element is compared with all other elements. If the 0^{th} element is found to be greater than the compared element then they are interchanged. So after the first iteration the smallest element is placed at the 0^{th} position. The same procedure is repeated for the 1^{st} element and so on. This can be understood from the figure given in the slide.

Thus, if the number of elements to be arranged is 5 then there would be 4 (5 - 1) passes. Hence in the program **i** (loop counter **for** outer for loop) is varied from 0 to 3 and **j** (loop counter for inner **for** loop) is initialized with a number one next to **i** (**i + 1**) and comparison is made using **if** between **a[i]** and **a[j]** and values are interchanged, if the condition is satisfied.

Bubble Sort

The program given in the slide uses Bubble sort method to arrange elements in ascending order. In this method, to begin with the 0^{th} element is compared with the 1^{st} element. If it is found to be greater than the 1^{st} element then they are interchanged. Then the first element is compared with the 2^{nd} element, if it is found to be greater, then they are changed interchanged. In the same way all the elements (excluding last) are compared with their next element and are interchanged if required. This is the first iteration and on completing this iteration the largest element gets placed at the last position. Similarly, in the second iteration the comparisons are made till the last but one element and this time the second largest element gets placed at the second last position in the list. As a result after all the iterations the list becomes a sorted list. This can be understood from the figure given in the slide.

Sorting Procedures

There are various sorting techniques as listed in the slide. Among these the order of Selection sort and Bubble sort is n^2 i.e. for 10 of elements, 100 comparisons are made. The Quick sort **qsort( )** is the fastest sorting procedure since it is implemented through recursion and its order is $\log_2 n$ i.e. for 10 number of elements only 3 comparisons are made.

Quick Sort At Work

The program given in the slide uses standard library function **qsort( )** to arrange elements of array. The call made to **qsort( )** is,

```
qsort ( a, 5, sizeof ( int ), fun ) ;
```

where,

a is the base address of the array

5 is the number of elements in the array

sizeof (int) is the size of each element in the array

fun is a user-defined function that compares two elements and returns a value based on the comparison

Through the call made to **qsort( )**, it is clear that only writing a comparison function is the responsibility of the programmer.

TwoD Arrays - I

In this lecture you will understand:

* How to access array using pointers
* How and which arithmetic operations be performed on pointers
* How to pass elements of array to a function
* How to pass array entire array to a function

Another Form…

The slide shows one more way of accessing array elements.

In the program given in the slide, an array **a[ ]** is initialized and it is assumed that its base address is 102. Address of 0^{th}, 1^{st} and 2^{nd} element of the array is printed using the 'address of' operator (**&**).

On printing **a**, **a + 1** and **a + 2** same output is displayed. Next, the value present at the address locations is printed using 'value at address' operator (*). This method of referring array elements is called as Pointer Notation and the earlier one is called as Subscript Notation.

More Forms...

This slide shows all the possible ways in which an array element can be accessed.

When we say **a[i]**, the C compiler internally converts it to ***(a + i)**. Thus the notations, like ***(i + a)**, **i [a]** are same. The program shown in the slide proves this.

Flexible Arrays

Saying **5[a]** is equivalent to **a[5]**, but only while accessing the elements. While declaring this array saying **int 5[a]** is not allowed.

While declaring an array, in place of dimension, a variable cannot be used. It must always be a positive, non-zero, an integer constant. A macro as shown in the slide is allowed.

Pointer Arithmetic

The program given in the slide shows how arithmetic operations are carried out on pointers.

Here, three variables and their respective pointer variables of the type **float**, **char** and **int** are declared. Addresses of the variables are stored in their respective pointer variables. The figure in slide shows the variables their values and addresses and also the contents of pointer variables after their initialization. Next, the addresses hold by pointer variables are printed through the first **printf( )**.

Now the pointer variables **b**, **dh** and **j** are incremented. On printing the contents of these pointers we get the output as 1012, 2010, and 6004 (for **b**, **dh** and **j** respectively). It is clear from this output that, the **float** pointer **b**, gets incremented by 4 bytes, **char** pointer **dh**, gets incremented by 1 byte and the **int** pointer **j**, gets incremented by 2 bytes.

Next on adding 3 to **b** it gets incremented by 12 i.e.(4 * 3 bytes), on adding 8 to **dh** it gets incremented by 8 i.e. (8 * 1 bytes) and on subtracting 3 from **j** it gets decremented by 6 i.e.(3 * 2 bytes).

The following are some of the legal pointer arithmetic allowed in C language:

- a number added to a pointer results into a pointer
- a number subtracted from a pointer results into a pointer
- a pointer subtracted from a pointer results into a number

Access Using Pointers

The program given in the slide shows how to access an element of an array using pointers.

Here, an array **a** is initialized and it is assumed that its base address is 102. With the statement **p = a**, we are storing base address (102) of the array **a** in **p** which is same as the address of the 0^{th} element. Saying ***p** gives value present at the address 102 which is 7. On incrementing **p** by one, it points to the next element of the array (incremented by size of an **int**, i.e. 2 bytes) and hence **p** gives the value at 104.

Next, the base address is restored in **p** by assigning **a** and a **for** loop is used to print all the elements of the array using pointer **p**.

The two statements, ***p** and **p**++ can be clubbed together.

If we write ***p**++, first operation will be ***p** and then **p**++. If we write as ***++p**, firstly **p** will be incremented (++**p**) and then value at address in **p** is will be displayed.

If we write ++***p**, the value present at **p** gets incremented and not the address.

[ ] For Notation

The brackets **[ ]** used while accessing elements of array, is just a notation to show that this variable is an array. Otherwise, the array elements can be accessed in any of the four ways as shown in the program given in the slide.

The point to be noted here is that when we use expression like p**[i]** or **i[p]** or ***(p + i)** or ***(i + p)**, then one of them (either **i** or **p**) can be a pointer or an array, the other should be an **int**, but both of them can never be arrays. In other words if **p** is an array then **i** should be an **int**, or if **p** is a pointer then **i** should be an **int**.

Changing Array Address

In the program given in the slide, **a** is an integer array containing 5 elements and **p** is pointer to this array. Let us assume the base address of array **a** as 102. Then through first **for** loop we have printed the values of array **a** using pointer **p**. In every iteration of this loop, we have incremented **p** to make it point to the next element of array. In the second **for** loop also we have tried to print the values of array, not using pointer **p**, but using the array variable **a** itself. However, on executing the program the second **for** loop flashes an error. The error is in the statement **a++**. **a** being an array, just mentioning **a** gives it's base address. Now, saying **a++** (i.e. **a = a + 1**) means we are trying to change the base address and that's not allowed.

Always remember that the base address of an array can never be altered. Therefore, any arithmetic involving name of an array is wrong. Hence, the statements given in the slide, which perform arithmetic operations using array name, are all invalid.

Passing Array Elements

In the program given in the slide, elements of an array are passed to function in two different ways. In the first way, all elements are passed to the function **display( )**. In the second way, through a **for** loop only one element i.e. $\mathbf{i}^{th}$ element is passed to function **display1( )**.

We cannot prefer the first way, because if an array contains more elements say 100 elements, then it would become tedious. The second way is good, but it affects the speed of execution due to number of function call overheads.

Passing Entire Array

The program given in the slide, gives a better way of passing elements of an array to a function. Instead of passing the elements, the entire array can be passed to a function by passing the base address of the array. The **display2( )** function implements this and array elements are accessed using pointer notation. But this is not the generalized form, because in the **for** loop used in **display( )** function, the upper bound is given as 4. If we add or remove some elements from the array, then we need to change the upper bound in **for** loop of **display2( )** function also. This is needed because **display2( )** doesn't know the size of array. Hence the better way is to pass the size of the array also the function, when entire array is being passed. This is what is done in **display3( )** function. It receives base address of array **a** and the size (i.e. upper bound) of the array. The condition is for loop of **display3( )** should be **i <= n** and **i <= 4**.

So always remember, if an array is to be passed to a function both it's base address and size should be passed.

TwoD Arrays - II

In this lecture you will understand:

* How array elements are arranged in memory
* How to create 2-D arrays
* How to represent elements of 1-D and multidimensional array using pointer

Two Dimensional Arrays

So far we have explored arrays with only one dimension. It is also possible for arrays to have two or more dimensions. Let us now see a program that uses a two-dimensional array. A two-dimensional array is nothing but an array of one-dimensional arrays.

In the program, a 2-D array **a** is initialized. The first dimension i.e. number of rows is optional. But second dimension i.e. column dimension is necessary. A comma should separate each element. The braces to separate the rows are optional. Saying **a[2][4]** specifies 21 present at 2nd row and 4th column. Then we have found out the size of array **a** using **sizeof( )** operator and displayed the base address by providing the name of the array. To print all the elements of a two-dimensional array we need two **for** loops, one for row and the other for columns.

This is an interesting fact that for any n-dimensional array as shown in the slide, first dimension is optional but rest is compulsory.

Find Biggest

Here is the program to find the biggest number present in the 2-D array.

It is assumed that the element **a[0][0]** is the biggest element and is stored in **big** variable. Then individual element is compared with **big**. If the element is greater than **big**, then **big** is replaced by this element. Its row-column position is also stored in **r** and **c** respectively. Finally, the biggest element **big** thus, found out is printed.

On your part, try to find out the second biggest number and its position.

Slide Number 3

When a matrix, i.e. 2-D array gets stored in memory all elements of it are stored linearly. The figure given in the slide shows how elements of a 2-D array get stored in memory. This arrangement is called Row Major arrangement, where, elements are stored row-wise. Thus, if a 2-D array contain 3 rows 4 columns, then the elements of 0th row get stored, then gets stored the elements of 1st row, and lastly, the elements of 2nd row gets stored.

On executing the program given in the slide, to your surprise, you would find, most of the answers are wrong except the last **printf( )**. Let's see where we went wrong.

Slide Number 4

Consider the code snippet given in the slide. Let us assume the base address of one-dimensional array **b** is 402. Saying **b** in **printf( )**, prints its base address 402 and ***b** prints the value as 7. If an integer variable **i** is initialized with value 5, then on printing value of **i**, prints 5. The point is any variable prints what it is holding. However, this is not the case with a two-dimensional array.

If **a** is a 2-D array, simply mentioning **a** in **printf( )** prints the base address of the 2-D array and even, ***a** prints the base address, but of the first one-dimensional array (i.e. the base address of 2-D array itself) because that's what it is holding.

A 2-D array **a[3][4]** of integers can be thought of as setting up an array of 3 elements, each of which is a one-dimensional array containing 4 integers. **a[0]** gives the address of 0th one dimensional array, **a[1]** gives the address of the first-one dimensional array and so on.

Slide Number 5

Now you can appreciate the results. **a** prints 502, ***a** prints 502. **a + 0** , **a + 1**, **a + 2**, prints addresses of 0th , 1st and 2nd row i.e. 502, 510 and 510, so **a[0]**, **a[1]**, **a[2]** too prints the same.

a[0] + 1, **a[1] + 2**, **a[2] + 3** prints the addresses of 1st , 2nd and 3rd element of 0th , 1st and 2nd rows. By placing * before them gives the corresponding values of the elements. Lastly, the same elements are printed using bracket notation in which row and column indexes are specified.

Moral

This slide shows the subscript and pointer notation for referring an element of array.

3 Ways

The program given in the slide uses three ways to access element of a 2-D array. It also shows the ways in which **i**th, **j**th, **k**th element of a 3-D array can be accessed. Thus, for a multidimensional or more specifically **n**-dimensional array, there exists **n+1** ways of referring its elements.

Different

Consider the program given in the slide. Note the declaration of pointer **p** and **q**.

***p[4]** is an array of 4 pointers. Let us assume that the base address of **p** is 130. Then **q** is a pointer to a one-dimensional array of 4 elements. The **p[ ]** array is then initialized with the addresses of some elements of array **a**. Then the base address of the array **a** is stored in an integer pointer **r** and in **q**. Incrementing both the pointers by 1, **r** prints 504 but **q** prints 510, that is amusing. The result is so because **q** is a pointer to an array of 4 integers. Therefore on incrementing it points to next array starting at 510. On the other hand **r** is a pointer to an **int** holding address of element of 0th row, 0th column of array **a**. Hence, on incrementation, it points to the next 2 bytes which happens to be the address of element of 0th row, 1st column.

Application of Arrays - I

In this lecture you will understand:

* How to create 2-D arrays
* How to represent elements of 1-D and multidimensional array using pointer

Applications of 2-D Arrays

This slide lists a few applications where a 2-D array can be used.

A 2-D array, which is also called a matrix, can be used to perform matrix operations like, addition, multiplication, determinant, transpose, inverse, etc.

It can also be used to write program for the famous 'Eight Queens' problem that one can play on a chessboard, as shown in figure given in the slide.

In this game 8 queens (8 pawns can be used as 8 queens) have to be placed on an 8 by 8 chessboard so that none can take the other. Knowing that the queens can move in horizontal, vertical and diagonal directions, it is a challenge of sorts to come up with such a combination. The program for this is given in next slide that provides more than 90 different ways to manage this, only one of which is shown in figure given in the slide.

The Program

The solution to the program is given in this slide.

Knight's Tour

In a game called Knight's tour as shown in the figure given in the slide, we can make use of 2-D array.

Earn A TV…

Often in newspaper we find advertisements that claims you can "Earn a TV or an Audio System", etc. if you can solve the puzzle like the one given in the slide. To write a program to solve such puzzles, we need to use a 2-D array.

Tic-Tac-Toe

The concepts, working and features of 2-D array discussed so far are enough to develop a game Tic-Tac-Toe (shown in the slide).

Applications of Arrays - II

In this lecture you will understand:

* How to create a 3-D array
* Applications of 2-D array

Puzzle

Let us now write a program for a puzzle game shown in the slide.

To begin with we have initialized a 2-D array **a** as shown in the slide. This array has been defined globally so that it can be accessed in all the user-defined functions called in **main( )**. The **boxes( )**, a user-defined function has been called to display the grid as shown in the slide. The **display( )**, again a user-defined function has been called to display the numbers in that grid. The graphical characters required to draw box are shown in the slide.

Slide Number 2

The **display( )** function is discussed in the slide. The row **r** and column **c** are initialized with the 11 and 21 (because, the location 10^{th} row, 20^{th} column is used to display graphical character for left-most top corner).

Now two **for** loops (one for row and other for column) run through which the cursor is placed at specific position using **gotorc( )** (whose function definition is present in 'goto.c' file) and the elements of the array **a** are printed if the value is not equal to zero. If it is then two blank spaces are printed. For the next element to print at the appropriate row and column position, **r** and **c** are incremented by 2 and 3. In the outer loop again to restart printing from the same column, **c** is reset to 21.

Slide Number 3

Now after displaying the boxes and numbers, to move the numbers in the puzzle the arrow keys should be tackled. The scan codes of the arrow keys are given in the slide. Accepting a character through **scanf( )** or **getch( )** simply receives the ASCII equivalent and not the scan code. The ASCII values of arrow keys are zero.

The scan code of the keys of the keyboard can be retrieved using **getkey( )** function defined in 'goto.c' file. This is used in the program to receive the scan codes of the keyboard.

After collecting the scan code of the key, it is passed to the **switch** statement for the cases to be handled. Here code to handle down arrow key (80) is given. On pressing down arrow key, 15 should come down (where 2 blank spaces are placed) and place where 15 was should be replaced with blanks. Hence swapping between these corresponding elements is performed and again **display( )** is called to reflect the changes on the screen.

Slide Number 4

The case shown in the earlier slide is specific. To make it general initially **r** and **c** should be initialized with 3 (since, the row and column dimension of last element of the array i.e. 0 is 3, 3 respectively). After swapping, the value of **r** is decremented by 1 because 0 is now at position 2, 3. After this **display( )** is called to reflect the changes.

Since we are decrementing **r**, when it becomes zero it should not further get decremented. Hence, before interchanging we have checked whether **r** is non-zero. If **r** becomes 0 then we are simply playing a beep sound. This we have done in the **else** part.

Slide Number 5

On similar lines, when up arrow key (72) is pressed the element at the row below element 0 (or below row of blank space) should come up and the blank should move down. The code given in the slide handles case for up arrow key. Here, as the blank goes down the row **r** is incremented. Hence a condition is checked that whether **r** is equal to 3. If it is then again a beep sound is played. Rest of the

code is straightforward. Try the remaining two cases yourself, where column **c** is to be decremented or incremented depending on whether left or right arrow key is pressed.

Slide Number 7

After the numbers are shifted according to the arrow keys pressed, a check should be made to verify whether the elements of the array **a** are linearly arranged or not. This we have done by calling again a user-defined function **check( )**. Also to play the game the i.e. to shift the numbers till they don't get arranged in linear order, the whole logic is placed inside an infinite **while** loop.

Slide Number 8

Let us now discuss 3-D arrays. Note the way a 3-D array **a** is initialized in the slide. As discussed earlier, the first dimension of any dimensional array is optional, hence the first dimension is omitted.

A 3-D array shown in the slide, consists of three sets each of which is again a two dimensional array. The first **printf( )** of the program given in the slide, prints the size of **a** as 48 (24 * 2) and **a[2][1][3]** prints the value present in the second set, first row and second column i.e. 100. In the next **printf( )**, the same value is printed using the pointer notation.

In the last **printf( )**, **a** prints the base address of the set (i.e. 3-D array), ***a** prints base address of row, ****a** prints the base address of the column and finally *****a** prints the value present at the 0^{th} set, 0^{th} row, 0^{th} column which is 3.

Strings - I

In this lecture you will understand:

* What are strings
* How to create strings
* How to input a multiword string
* Difference between **printf( )** and **puts( )**

Strings

So far we have seen numeric arrays. Let's switch over to the character arrays. In C character arrays are treated as strings. A string is a character array that is terminated by a null (**'\0'**) character.

The program shown in the slide contains a string **name** initialized with character constants, as type is **char**. The way we printed numeric arrays, character arrays can also be printed in the same way. In the slide it has been done using a **for** loop which needs the number of elements to know when to stop the **for** loop.

Moreover, a string is a one-dimensional array of characters terminated by a null ('\0'). So, instead of relying on the length of the string to print its contents, a better way is to traverse a string till end of the string i.e. '\0' has not met. This is what we have done in the **while** loop. Note that printing character and incrementing **i** can be done using a single statement as **name[i++]**.

Two More Ways

Two more ways are there to print the contents of string. The ASCII equivalent of null '\0' is zero. Hence instead of checking for the null ('\0') character, we can check for 0 as well. This is what we have done in the first **while** loop in the program given in the slide.

Also we know that false value (zero) in the **while** statement terminates the loop, hence, only **name[i]** is mentioned in the condition of second **while** loop. **name[i]** used in **while** gives the ASCII value of $\mathbf{i}^{th}$ character. When **name[i]** reaches '\0', the condition in the **while** gets replaced by 0 thereby terminating the loop.

Which Is Best?

It is better to use **for** loop when number of iterations is known, i.e. when a loop has to run for finite number of times then **for** loop should be used. In case of handling strings, it is not known how many iterations are required to reach the end of the string, hence the better way is to use **while** loop. The slide shows three ways in which **while** loop can be used to access characters of a string. The best way to print the contents of string is to use **printf( )** with **%s** as format specifier. Lastly, when we pass an entire array of integers to a function, we need to pass the size of the array in addition to the base address of the array. However, in case of a string we need to pass only the base address to the function, because the end of the string can be determined by null terminating character '\0'.

Multiword Strings

The character array **str1** is initialized with character constants and is terminated by '\0' and **str2** is initialized with the string of characters enclosed within double quotes, where '\0' character is not explicitly specified, assumed by default.

On printing the sizes of the strings, both prints 7, the total number of characters present in the string.

str3 is accepted through **scanf( )**. Supplying a multiword string say "Rahul Sood" and on printing it prints only "Rahul". This is due to **scanf( )**, which treats space as a separator for input. Now again same string is accepted through **gets( )** and printing it through **printf( )**, it prints "Rahul Sood". **str3** can be printed through the **puts( )**, counterpart of **gets( )** to print strings.

Note the constants 3, 3.0, '3' and "3" are different, where 3 is an integer, 3.0 is a float value, '3' is a character constant and "3" is a string with '\0'.

Which Is Better?

A comparison between **printf( )/scanf( )** vs **puts( )/gets( )**. Through **printf( )** or **scanf( )** a number of strings can be printed or accepted. On the other hand, **puts( )** or **gets( )** prints or accepts only one string at a time.

When number of strings are to be printed or accepted use **printf( )/scanf( )**. Moreover **printf( )** is capable of printing multiword strings as well as other types using format specifiers.

When single multiword strings are to be printed or accepted use **puts( )/gets( )**.

Think Differently

A string **str** is initialized with "Sanjay". In pointer **p**, base address of **str**, let's say 401, is stored. On incrementing **p**, it will point to the next character in the string. Using this concept the string is printed character by character through a **while** loop using the pointer **p**.

Output?

Now when a string is passed to **printf( )**, it is accepted in a **char** pointer and printed using pointer notation as described. It is clear that in **p** only the base address of "Hello" is stored.

Now suppose an expression as in the second **printf( )** is used then only "hanical" would be printed. Suppose the base address of "Mechanical" is 100. Now the expression 2 + 3 % 2 implies 3, hence

3 + "Mechanical" => 3 + 100 => 103

Starting from 103, the string "hanical" is printed. On similar lines, **printf( )** prints the strings passed to it along with list of variables.

Strings - II

In this lecture you will understand:

* How to find length of a string
* How to copy one string to another
* How to concatenate two strings
* How to convert characters in a string from lower to uppercase
* How to use standard library functions for working on strings

Slide Number 1

Length of a string can be found by traversing through the string character by character. To find the length of a string a function **xstrlen( )** is written. The length of strings **str1** and **str2** are received in **l1** and **l2** as 6 and 9 respectively, as the count of visible characters in the strings.

The **xstrlen( )** function also returns the length when a constant string is passed to it.

In **xstrlen( )**, a **while** loop is run till end of the string is not met. Through this loop using character pointer **p**, the string is traversed character by character and **count** is incremented. Lastly **count** is returned as a length of the given string.

Copying Strings

The contents of one string can be copied into another string. To copy string pointed to by **str1** to **str2** saying **str2** = **str1** ; would lead to an error. This is because **str1** represents the base address, which is a constant and cannot be changed.

To carry out copy operation **xstrcpy( )** function is used which accepts base address of the target and source string respectively.

The source string is traversed till '\0' is not met. In every iteration of **while** loop a character of source string (pointed to by **s**) is inserted into the target string (pointed to by **t**). After copying the entire source string into the target string, it is necessary to place a '\0' into the target string, to mark its end.

Note that it is our responsibility to see to it that the target string's dimension is big enough to hold the string being copied into it.

More Ways…

More ways of copying source string into target string are shown in the slide. The first way is simple one, which we have already discussed, in the earlier slide. In the second way, in **while** loop it is checked whether ***s** is '\0'. If it is not, then copying of the character to the target string (pointed to by **t**) and shifting **t** and **s** to the next character of the corresponding strings is done through the statement,

*t++ = *s++ ;

Here, first ***s** is stored in ***t** and then **t** and **s** are incremented.

In the third way, note the condition given in the **while** loop. While executing the expression ***t**++ = ***s**++, first ***s** is copied into ***t**, the value ***t** is then checked for '\0'. The order of evaluation of the statement is shown in the slide. The moment ***s** becomes '\0', the loop terminates as *t also becomes 0 ('\0').

Concatenation

The process of adding one string at the end of another string is called concatenation. The target string, which is going to hold another string, should be made big enough. As **str1** is going to hold other string **str2**, its size should be more say 20 for example.

We have called **xstrcat( )** function for concatenation. This function accepts base address of target and source string. Initially the target string pointed to by **t** is traversed till '\0'. Then as done in case of copy operation, the source string pointed to by **s** is traversed and copied into target string (pointed to by **t**) piece-meal, character by character. Lastly '\0' is added to **t** as a null character.

Shorter Version

A shorter version of the same **xstrcat( )** can be written using the standard library function **strlen( )** and **strcpy( )** which can be used by **#includ**ing 'string.h'.

The pointer to target string **t** is made to point to the end of the string by adding length of the string (i.e. target string) to it. The length is extracted by calling **strlen( )** function. Then the source string (pointed to by **s**) is copied to the target string pointed to by **t** using **strcpy( )** function.

Concatenation

Some more ways of concatenation operation have been shown in the slide. In first way **str1** is copied into **str3** using **strcpy( )**. Then, **str2** is concatenated at the end of **str3** using **strcat( )**.

Concatenation can be merely performed using **strcpy( )** or **strcat( )** only. When **strcpy( )** is used, first **str1** is copied to **str3**. Then again in **strcpy( )**, first, **str3** is made to point to the end of the string by adding length (of string pointed to by **str3**) to it. Thus, **str3** points to '\0'. The string pointed to by **str2** is then copied to **str3**.

When **strcat( )** is used for concatenation, first **str3** is initialized with '\0'. Then a string pointed to by **str1** is appended to **str3**. Again using **strcat( )** second string pointed to by **str2** is appended to **str3**.

Convert To Upper Case

The lowercase characters present in the string can be converted to uppercase. For this **xstrupr( )** is used. In this function the given string pointed to by **p** is traversed till '\0' is not met. While traversing, it is checked whether the character is a lowercase character. If it is, then it is converted to uppercase by subtracting 32 from the ASCII equivalent of character being considered.

What's The Difference?

The standard library provides **strupr( )** for **xstrupr( )**. The counterpart for this is **strlwr( )**, which converts uppercase characters of the string to lowercase. The library functions like **toupper( )** and **tolower( )** are also used for case conversion, however they work on a character and not on a string. We can use **tolower( )** or **toupper( )** in place of **strupr( )** or **strlwr( )** to convert characters.

Comparing Strings

Often while working on strings it is required to arrange them in alphabetical order. This can be done by comparing the strings. In our program, to compare strings we are using **xstrcmp( )** function, yet not defined, but the purpose is clarified through the output shown in the slide. If the strings are unequal then a nonzero value is returned and when the strings are equal zero is returned.

Comparing Strings

The **xstrcmp( )** function is defined in this slide. Both the strings are traversed till the characters (pointed to by **t** and **s**) are identical. The loop terminates either if the characters are not matching or if the string pointed to by **t** has reached to null character '\0'(i.e. end of the string). Once, the loop is over, the difference between the ASCII values of the characters at which **char** pointer **t** and **s** are pointing to is returned.

Outputting Strings

To output a string a user-defined version **xputs( )** has been defined for standard library function **puts( )**. The **xputs( )** function, receives the base address of the string which is to be displayed on screen. The base address is collected in char pointer **p**.

In the function, the string is traversed till end of the string has not met. While traversing the string, the character pointed to by **p** is printed using **printf( )**. The character can also be printed using **putch( )**. However, **putch( )** merely prints one character at a time.

Standard Library Functions

Lastly, some of the standard library functions used for working on strings are listed in the slide.

Strings - III

In this lecture you will understand:

* What is a **const** pointer
* The need of **const** pointer
* How to handle two-dimensional array of characters
* How to create array of pointers to strings

What Is p

Let us understand following code, which is also shown in the slide:

```
char  *p = "Hello" ;
p = p + 1 ;
p = "Bye" ;
*p = 'M' ;
```

Here "Hello" is a constant string for which memory is allocated at runtime. Only 6 bytes would be allocated. The base address of this string would then be stored in the pointer **p**. Since **p** is pointer variable pointer arithmetic can be applied on it. Moreover, pointer **p** can be assigned new string like "Bye". We can change any character in the string pointed to by **p** using the indirection operator *.

Let us now try to understand the following code:

```
char p[ ] = "Hello" ;
p = p + 1 ;
p = "Bye" ;
*p = 'M' ;
```

Here **p** is a character array, which is assigned a string "Hello". 6 bytes would be allocated for the array and "Hello" would be stored in the array. **p** being the base address of the array, it cannot be changed. The pointer **p** can be changed neither by assigning some value like **p = p + 1** nor by assigning base address of some other string, for example **p = "Bye"**. To change a character in the array we can either use the subscript like **p[0]** or use the indirection operator * with **p**.

Go through the code snippet shown below:

```
const  char  *p = "Hello" ;
p = p + 1 ;
p = "Bye" ;
*p = 'M' ;
```

Here the string to which **p** is pointing is fixed not the pointer **p**. This means that using **p** we cannot change any character in the string to which it points. But the value of **p** can be changed. We can make **p** point to another string by storing the base address of other string in it.

Let us now try to understand the following code:

```
char  * const  p = "Hello" ;
p = p + 1 ;
p = "Bye" ;
*p = 'M' ;
```

Here **p** is declared as a **const** pointer. This means that **p** cannot be changed; hence we cannot make **p** to point to another string. But the string to which **p** points is not fixed. We can change the characters in the string. Note that it is mandatory to initialize **p** at the same place where it is declared.

Look at the following code snippet:

```
char  const  *p = "Hello" ;
p = p + 1 ;
```

```
p = "Bye" ;
*p = 'M' ;
```

This is same as **const char *p**.

Let us now try to understand the following code:

```
const  char * const  p = "Hello" ;
p = p + 1 ;
p = "Bye" ;
*p = 'M' ;
```

The declaration **const char * const p ;** is a combination of **const char *p** or **char const *p** and **char * const p**. Here neither **p** can be made to point to another string (once initialized), nor we can change characters in the string using **p**.

Utility of *const*

The program given in the slide copies source string **str1** to target string **str2**. We have already discussed the working of **xstrcpy( )** function. Note the function definition of **xtrcpy( )**. Here the first character of source string is changed from 'H' to 'A'. Can we not ensure that the source string doesn't change even accidentally in **strcpy( )**? We can by declaring **char *s** (of **xstrcpy( )**) as **const**. By declaring **s** as a **const** pointer we are declaring that the source string should remain constant (should not change).

Thus the **const** qualifier ensures that your program does not inadvertently alter a variable that you intended to be constant. It also reminds anybody reading the program listing that the variable is not intended to change.

Handling Several Strings

The code snippet in the slide shows how multiple strings can be handled using one-dimensional character arrays. Better way to do this is using 2D array.

Array of Strings / 2-D Array

The program given in the slide shows how a 2-D array of characters can be used to store multiple strings.

Notice how the two-dimensional character array has been initialized. The order of the subscripts in the array declaration is important. The first subscript (which is optional) gives the number of names in the array (which in our case would be 5). The second subscript gives the length of each item in the array.

The size of array **n** is then displayed which would be 100, as there are 5 strings and for each string 20 bytes are reserved. The next statement displays character of specified string where,

n[0][1] means 0^{th} string's first element, i.e. 'a' of "Sanjay"

n[1][0] means first string's 0^{th} element, i.e. 'A' of "Amol".

Slide Number 5

The slide shows how the strings of array **n** are stored in memory. The strings are stored in contiguous or adjacent memory locations. Note that each string ends with a '\0' and each string is 20 bytes apart from the other. The arrangement as you can appreciate is similar to that of a two-dimensional numeric array. Hence the strings in the array are displayed in the same manner as we do for two-dimensional numeric arrays, i.e. using two **for** loops. However, with string array the printing of characters of string stops when a '\0' is encountered.

Slide Number 6

The program given in the slide interchanges the first string with the second one. A **for** loop is run, through which strings are interchanged character by character. The strings are then printed. In the **printf( )** statement, format specifier **%s** is used and the base address of the string to be displayed is supplied. Even using **n + i** in place of **&n[i][0]** would work.

Disadvantages

Note the in-memory representation of two-dimensional array of characters shown in the slide. Here, 401, 421, 441, etc. are the base addresses of successive names. Some of the names do not occupy all the bytes reserved for them. For example, even though 20 bytes are reserved for string the name 'Sanjay', it occupies only 7 bytes. Thus 13 bytes go waste. Similarly for each name there is some amount of wastage. In fact, more the number of names, more would be the wastage. Thus, the processing of strings becomes inefficient. This can be avoided by using what is called an 'array of pointers'.

Slide Number 8

In the program shown in the slide **n[ ]** is an array of character pointers. It contains base addresses of respective names. That is, base address of "Sanjay" is stored in **n[0]**, base address of "Amol" in **n[1]** as shown in the slide.

In the two-dimensional array of characters, the strings occupied 100 bytes, as against this, in array of pointers, the strings occupy only 42 bytes. A substantial saving, you would agree. Thus, one reason to store strings in an array of pointers is to make a more efficient use of available memory.

Another reason to use an array of pointers to strings is to obtain greater ease in manipulation of the strings. Note here, to exchange the position of first string with the second string, we are required to do is exchange the addresses (of strings) stored in the array of pointers, rather than names themselves. Thus, by effecting just one exchange we are able to interchange strings. This makes handling strings very convenient.

Slide Number 9

The slide shows the selection sort method applied on strings, in which $\mathbf{i}^{th}$ element is compared with the rest.

For comparing strings **strcmp( )** is used. If it returns non-zero positive value then the $\mathbf{i}^{th}$ string is alphabetically greater than the $\mathbf{j}^{th}$ string. In this case only the base addresses of strings are interchanged.

Calendar

In this lecture you will understand:

* How to display a monthly calendar

Calendar

Let us now write a program to display calendar for the specified month and year. To display calendar for given month and year, first we need to find the first day for the given month of given year. To get this we also need to know the total days from 1/1/1 to first day of the given month of given year (i.e. upto 1/<given month>/<given year>). We need to find total days from 1/1/1 to 31/12/<given year-1> plus the total days from 1st January of given year to first day of the given month of given year. We have already discussed a program that finds the first day of the specified year. The similar logic is used here.

In the program shown in the slide, month **m** and year **y** have been accepted through the keyboard. Suppose the values entered for **m** and **y** are 8 and 2002. First the normal days, i.e. from 1/1/1 to 31/12/2001 are calculated. We have also calculated the leap days for the same period. The normal days and leap days thus found are then added to get total days.

...Calendar

The logic discussed is implemented in this slide. An integer array **days[ ]** has been used that stores days of every month. First the total days i.e. from 1/1/1 to 31/12/2004 are calculated and stored in **totaldays**. Then the total days from 1/1/2005to 31/7/2005 are calculated and stored in **s**. The days thus found are then added to **totaldays**. The **firstday** i.e. day of 1/8/2005 is then determined.

However, while calculating total days for the year 2005 we have not checked for leap year. The code snippet given in the slide now checks whether given year is leap or not. If it is then 29 days are added for February month.

Slide Number 3

The slide shows the row and column position for the day names for printing the calendar.

...Calendar

The column position for printing first day of the specified month (i.e. August) is calculated. Then at appropriate positions the month name, year and names of days are printed using **gotorc( )** and **printf( )**.

...Calendar

The additional code shown in the slide, prints the days of the given month (i.e. August). To print the days **for** loop is run. Each time through loop, **col** is incremented by 6 and when it becomes greater than 56, **row** is incremented and **col** is reset to 20.

...Calendar

Now to make the program interactive, we shall use arrow keys. On hitting right arrow key calendar for same year next month should get displayed, whereas on hitting left arrow key calendar for same year but previous month should get displayed. Hitting up arrow key should display calendar for next year for same month and hitting down arrow key should display calendar for previous year for the same month.

The **getkey( )** function used in the program shown in the slide returns the scan code of the arrow key being hit. When Right Arrow key (77) is hit the month **m** is incremented and if **m** becomes greater than 12, then **y** is incremented and **m** is reset to 1. On similar lines write logic for handling rest of the arrow keys. Note, the whole logic is put in a **while** loop that runs indefinitely.

...Calendar

The program being at some places gives incorrect number of total days. We have checked for possibility of leap year, if the condition satisfies we have straight away stored 29 in the array **days**. Next time even though when the condition fails the days are taken as 29 only. This is the reason why we get incorrect number of days. The change required to be done is shown in the slide. Note the **else** part.

The September month for year 1752 contained 16 days only, (After 3rd directly 14th). Using **if-else** statements we can display calendar for September'1752. This is done using the **if** statement as shown in the slide in the block.

Structures - I

In this lecture you will understand:

* What are structures
* How to access elements of a structure through a variable and pointer
* How to create an array of structures

Terminology

The figure in slide illustrates the basic elements of a database. The columns shown in this figure that are labeled as Name, Age, and Salary are called Fields. Each row holds a specific value for every field, and is called a record. A database is a collection of such records.

Handling Data

Suppose, the data of an employee database, containing name, age and salary has to be stored. One way is to create three different arrays for three fields as shown in the program given in the slide.

Though this approach allows us to store data, it is an unwieldy approach that obscures the fact that you are dealing with a group of characteristics related to a single entity-an employee.

The program becomes more difficult to handle as the number of items relating to an employee go on increasing. For example, we would be required to use a number of arrays, if we also decide to store the name of department, address, etc. This will even destroy the natural relationship of the fields. To solve this problem, C provides a special data type—the structure.

Structures

A structure contains a number of data types grouped together. These data types may or may not be of the same type.

In the program given in slide, we have declared a structure named **employee**. It contain elements **n**, **a**, **s** as **char**, **int** and **float** representing name, age and salary respectively. The **struct** is a keyword used to declare a structure. The structure declared here, holds information for an employee, whereas we can create a structure to hold information about a book or a student also. Hence, a structure is called a user-defined data type.

Next, to be able to use the structure **employee**, we have defined three variables **e1**, **e2** and **e3** and then printed the data hold by these variables. Note how elements of structure are accessed. For accessing the structure elements through a variable of structure a **.** operator is used, whereas an -> operator is used when elements of a structure are to be accessed through a pointer to a structure. Here we can see that for each structure variable we need **printf ()** statement to print the structure elements, this can be avoided by using structure array, explained in the next slide.

Array of Structures

In the program given in earlier slide, we had created three structure variables to hold information. Suppose information about 100 employees has to be stored now. Declaring 100 structure variables is definitely not very convenient. A better approach would be to use an array of structures. The program given in the slide demonstrates how an array of structure can be created and used.

Terminology

Let us now look at the fundamental aspects of a structure. A structure is declared using **struct** keyword. Following this keyword is the name of the structure, which is also called structure tag. Within a pair of braces structure members/elements are declared. Note that the closing brace in the structure type declaration must be followed by a semicolon.

Once the new structure data type has been defined one or more variables can be declared to be of that type. For example the variables **e1**, **e2**, **e[10]**, etc. where **e1**, **e2** are structure variables and **e[10]** is an array of structures.

Conclusion

Unlike arrays, a structure is a collection of dissimilar elements. Whatever be the elements of a structure, they are always stored in contiguous memory locations.

Array of Structures

Consider code snippet given in the slide. Here **e[ ]** is an array of structures of structure **employee**. The slide shows in-memory representation of e[ ]. Then a few pointers are declared, where p is a **char** pointer, **q** is a pointer to **struct employee**, and **r** is a pointer to an array of structures of dimension 3.

All the pointers **p**, **q**, and **r** are initialized with the base address **e**. Each of the pointers, **p, q** and **r** are then incremented by 1. **p** being a **char** pointer, it will get incremented by 1 byte. The pointer **q** being a pointer to structure employee (of size 7 bytes) will get incremented by 7 bytes. The.pointer **r** being a pointer to an array of **struct employee** of dimension 3, it will get incremented by 7 * 3 locations.

If **z** is declared as,

struct employee *z[3] ;

then **z** would be an array of pointers to structures.

Declaration & Definitions

While declaring a structure the tag name is compulsory only if we wish to create structure variables later. But while declaring the structure if the required variables are also declared or defined, then the tag name is optional.

Copying

The program given in the slide shows two ways of copying a structure. In the first way, piece-meal copying of structure elements is done from structure variable **e1** to structure variable **e2**. However, it is not necessary to copy the structure elements piece-meal. The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator. In the program this is done by the statement, **e3 = e1 ;**

This copying of all structure elements at one go has been possible only because the structure elements are stored in contiguous memory locations.

Copying Arrays

Suppose there are two integer arrays and the values from one array are to be copied to other array. The code snippet given in the slide shows two ways to achieve this. In the first way through a loop, elements of **a[ ]** are copied to **b[ ]**. A better solution is given in the second code snippet. Here, a structure is created containing an array of integer as its element. A structure variable **a** is initialized with the required values. Then using assignment operator elements of **a** are copied in one shot to structure variable **b**.

Structures - II

In this lecture you will understand:

* How to create nested structures
* How to pass elements of a structure to a function
* How to pass structure to a function
* How to return structure from a function

Nested Structures

One structure can be nested within another structure. Program given in the slide, shows nested structures at work.

Here, there are two structures—**address** and **emp**. The **address** structure has city and pin code as its members. The **emp** structure contains name, age, salary and a variable of structure **address**. A variable **e** of struct **emp** is then created and initialized. The elements of **e** are then printed.

Notice the method used to access the element of a structure that is part of another structure. For this the dot operator is used twice, as in the expression,

e.a.(member name)

To refer **city**, use **e.a.city**, and to refer **pin**, use **e.a.pin**.

The format specifier to be used should be decided according to the final member being accessed. For example, in **a.b.c.d.e.f**, the type of **f** would determine the format specifier.

Passing Structure Elements

Structure elements can be passed to a function by passing value of individual elements or address of individual elements as shown in the program given in the slide.

Passing Structures

To pass individual elements would become more tedious as the number of structure elements go on increasing. A better way would be to pass the entire structure variable at a time. This is shown in the program given in the slide.

Here, a structure is passed to two functions. To the **display1( )** function the value of the structure variable **b** is passed to the variable **z** of type structure book as shown in the slide. This is similar to passing by values, copy of the variable **b** gets created,(i.e., memory space gets allocated for the variable **z).** This value is then printed using **printf( )**.

In the **show1( )** function address of **b** is passed, which is collected in a pointer to structure **pb**. Here we need to use the -> (Arrow) operator or the * operator as shown in the slide to access the members. While using * operator, the parentheses are necessary as **.** operator has higher precedence over *.

Now the question comes, how do we define the formal arguments in the function. We cannot say **struct book z** and **pb** because the data type **struct book** is not known to the function **display1( )** and **show1( )** respectively. Therefore, it becomes necessary to define the structure type struct book outside **main( )**, so that it becomes known to all functions in the program.

Complex Nos.

The program given in the slide demonstrates how a structure is passed and returned from a function. Here, the program contains a structure called **com** to hold two complex numbers i.e. **float** variables **r** and **i**. Then **a**, **b** and **c** have been declared as three structured variables, where **a** and **b** are initialized at the same place. Then we have called **add( )** function. To this function both **a** and **b** have been passed and the result is collected in **c**.

In the function **add( )**, we have created a temporary structure variable **z**. We have added the values of **r** of **x** and **y** and stored in **r** of **z**. Similarly we have added values of **i** of **x** and **y** and stored it to **i** of **z**. Finally we have returned the structure from the function. Note that to be able to user structure in a function, declare structure **com** as global (i.e. declare it outside **main( )**).

Complex Nos.

The program given in this slide is similar to the one discussed in previous slide, but with slight modifications. Here, in place of structure, we have used a **float** array. To the **add( )** function we have passed the base addresses of the arrays **a** and **b** respectively. In the function we have created a temporary array of **float**s of size 2, added the corresponding members and stored the result in array **z**. Lastly, we have returned the array which is collected in **float** pointer **c** in **main( )**. However, **c** is now pointing to an array, which does not exist. This is because, **z** being a local member of function **add( )** dies after the function is over. To avoid this we will have to declare array **z** as a global array, or declare it as member of **main( )** and pass its base address to the function. However, this increases the complexity of the program.

Passing Structures

If we execute the program shown in the slide using TC/TC++ compiler we get the address as 518, 502,521. As expected, in memory the char begins immediately after the **int** and **float** begins immediately after the **char**. However, if we run the same program using VC++ compiler then the o/p turns to be: 444, 448, 456. It can be observed from this o/p that the **float** doesn't get stored immediately after the **char**. In fact there is a hole of 3 bytes after the **char** as shown in the slide. This is because VC++ is a 32-bit compiler targeted to generate code for a 32-bit microprocessor.

Some programs need to exercise control over the memory areas where data is placed. For example, reading the contents of the boot sector (first sector on the floppy / hard disk) into a structure. For this the byte arrangement of the structure elements must match the arrangement of various fields in the boot sector of the disk. The **#pragma pack( )** directive specifies packing alignment for structure members. The **pragma** takes effect at the first structure declaration after the **pragma** is seen. Turbo C/C++ compiler doesn't support this feature, VC++ compiler does.

Here, **#pragma pack(1)** lets each structure elements to begin on a 1-byte boundary as justified by the output of the program 444, 448 , 449.

Applications of Structures

Some applications of structures are listed in the slide.

Data Structures - I

In this lecture you will understand:

* The disadvantages of arrays
* The disadvantages of static allocation
* How to allocate memory dynamically
* How to implement a linked list

Arrays

Arrays are simple to understand and elements of an array are easily accessible. However, arrays suffer from certain limitations. Consider the program given in the slide. In this program we are calculating the percentage marks of 10 students. For this an array **per[10]** is used. At a time, the marks of three subjects of a student are accepted and his percentage marks are calculated and printed. What if we wish to calculate percentage marks for more than 10 students? We can, but we need to change the program, i.e. change the dimension of the array **per[]**. This is what is the limitation of an array. Arrays have a fixed dimension. Once the size of an array is declared it cannot be increased or decreased during execution. When we declare an array, we mention the size of the array. The size tells what amount of memory has to be allocated for the array. This is what is known as static allocation.

Dynamic Allocation

Suppose, the percentage marks are to be calculated for students and the number of students would be entered through keyboard, i.e. at runtime. In such a situation we cannot use an array, since while declaring array the size is required and we don't know the size. In other words the memory for an array gets allocated statically. Since the number of students is decided at runtime, the required memory should also get allocated at runtime. This is what is known as dynamic allocation of memory. The program given in the slide shows how to achieve this.

In the program, first the number of students, **n** is accepted using **scanf( )**. Then the memory is allocated dynamically using a function called **malloc( )**. Note the way this function has been called. We want to allocate memory for **n** integers. An **int** takes 2 bytes, hence we have given **malloc (n * 2)**. To collect the base address of the memory thus allocated, the statement should be,

```
p = malloc ( n * 2 ) ;
```

But, **malloc( )** returns a **void** pointer. Hence we need to typecast it to the required data type. Thus the correct statement to allocate memory would be,

```
p = ( int * ) malloc ( n * 2 ) ;
```

Now through a **for** (which runs **n** times) marks are accepted, percentage is calculated and stored in the allocated memory through

```
*( p + i ) = per ;
```

Lastly, we have displayed the percentages of the students using pointer notation.

Memory Allocation

Using arrays allocates memory statically, which is decided at compile time. On the other hand, when memory is allocated dynamically using **malloc( )**, the memory is reserved at runtime i.e. at the time of execution.

Better Still…

In the previous program first we accepted the total number of students and then allocated required memory dynamically. The program given in the slide shows one more way of dynamic memory allocation, where as long as the user wishes to enter data for students, the memory gets allocated for that student.

In the program, a **while** loop runs as long as user wishes to enter data for the student. In every iteration, the marks of a student are accepted, the percentage is calculated, and then the memory is allocated to store the percentage. Note that the base address of the allocated memory is collected in **p**, and the percentage **per** is stored in it. The loop gets terminated if user enters 'n' (or 'N' or value other

than 'Y'). Suppose the percentages for the students thus calculated are to be printed now. We have stored them in **p**. However, if you print the value in **p** you would get the percentage of the last student. This is because in the next iteration again memory is allocated and collected in **p**. While doing so, the previous memory location pointed to by pointer **p** is lost which causes memory leak as shown in the figure given in the slide. To avoid memory leak a better solution is to take an array of pointers as shown in the slide. However, again the size of the array would be required to specify which would be static memory allocation.

Memory Leak?

Observe the program given in the slide.

Here, in **main( )**, an **int** pointer **j** and a function **f( )**, which returns an int pointer, is declared. Then we have called **f( )** and collected the returned value in **j**. In function **f( )**, we have declared an **int a** and returned the address of **a** (which is collected in **j**). However, **a** being a local variable objects, dies as soon as the function **f( )** is over. Thus **j** is pointing to a variable, which does not exist. Is this a memory leak? No, it's a dangling pointer pointing to dead or non-existing objects.

Best...

A better way is to store value and pointer to next value together. In the slide, 55 is the value, which is stored at 200 and 400 is the address of the next value. Same is the case with other values like 63, 28, etc. But the question is where to end this list. This is shown in the slide, where, in the last element along with the value 60, NULL is stored indicating the end of the list.

This is nothing but a linked list in which each element points to the next element. Each element of the linked list is called a node and each node consists of two parts viz., the data and the link to the next node. The last node points to the NULL.

Linked List

Let us see a program that builds a linked list.

Here, we have created a structure called **node**. The members of this structure are **data** and **link**, where **data** is declared as an **int** and **link** is declared as pointer to **struct node ***. The pointers **p**, **q**, **r** and **s** are declared of type **struct node *** and represents the nodes of a linked list. For each of these nodes first memory is allocated dynamically using **malloc( )**. Then a value is stored in **data** of each of the nodes. Then links are established between the nodes. To establish a link between nodes pointed to by **p** and **q** the statement,

```
p -> link = q ;
```

is given. Similar statements are given to establish a link between nodes **q** and **r** and **r** and **s**. In the last node **s**, NULL is stored. Thus node **p** points to node **q**, **q** points to **r** and **r** points to **s.** Finally **s** points to NULL. Note the statement that allocates memory for a node.

Slide Number 8

The first **printf( )** prints the values stored in nodes. The second **printf( )** shows one more way of printing values. It prints the values stored in **p**, **q**, and **s** respectively. It shows that elements can be accessed through the links when the link starts at **p**.

A better way is to use a temporary pointer to visit each node and print the data. This is what we have done in **while** loop. **t,** a temporary pointer to node, is made to point to the first node (i.e. **p**). In every iteration of this loop, first the value is printed and then **t** is made to point to next node by the statement

```
t = t -> link ;
```

In **while** loop we have checked for condition **t != NULL**. Thus after printing the value of last node when **t** becomes NULL and the loop gets terminated.

Most General

The generalized form of a program, which we discussed in slide 4, is given here. The program makes use of linked list.

Here, the structure **node**, which is made global, contains **per** an **int** and **link** a pointer to the structure itself, as data members. A pointer **p** of **struct node** is also declared as global so that it can be used in another function.

In **main( )**, **p** is initialized with NULL to indicate that the list is empty. Through a **while** loop marks are accepted and percentage **pp**. The percentage **pp** is then passed to function **add( )**, which creates a node and adds it to the list. The definition of this function is given in the next slide. This process continues till user wishes to enter marks for students.

Slide Number 10

The **add( )** function deal with two situations—adding a node to an empty list and adding node at the end of an existing list.

First memory is allocated for a node (which is pointed to by **r**), percentage **pp** is stored in **per** and NULL is stored in **link** part of the node.

Now if the linked list is empty i.e. **p == NULL**, then **p** is made to point to the first node by the statement,

```
p = r ;
```

However, if the linked list is non-empty, then a temporary pointer **t** is initialized with **p** and through the **while** loop the list is traversed till the node having its link as NULL is received. In the **while** loop each time **t** is made to point to the next node through the statement,

```
t = t -> link ;
```

When t reaches the last node the condition **t -> link != NULL** fails and the loop terminates. Once outside the loop, the previous last node is connected with the new node by the statement,

```
t -> link = r ;
```

and the link gets established. Note that **p** will always point to the first node in the linked list.

Data Structures - II

In this lecture you will understand:

* What is stack
* How to implement stack as linked list

Stack As A Linked List

A stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is often known as **top** of stack. When an item is added to a stack, the operation is called **push**, and when an item is removed from a stack the operation is called **pop**. If we implement stack as an array, it suffers from the limitation of an array that we have discussed already. This limitation can be overcome if we implement stack as a linked list.

The stack as a linked list as shown in the slide, contains the data and a pointer that gives location of the next node in the list. Initially a node with value 45 is created having a link as NULL and **top** is pointing to this current node. The elements are pushed to the stack, thereby updating the **top** and establishing the link between current node and the previous node.

On the other hand, while popping elements from the stack, **top** has to made point to the previous node.

Stack As A Linked List

The program given in the slide demonstrates how stack can be implemented as a linked list.

Initially a structure **node** is created as global. In **main( )**, we have declared **top** as a pointer to **struct node**. It is set to NULL. The first node with value 45 is created using function **push( )** and to make **top** to point this current node, its address is passed to **push( )**. On similar lines few elements are pushed on to the stack.

The elements of the stack are displayed using **displaystack( )** function. The number of elements in a stack is determined using **count( )**.

Elements are retrieved one at a time using **pop( )** and collected in **item**. Lastly the final count is again determined using **count( )** as an evidence to know the size of stack.

Push Element On Stack

Lets understand the push operation in detail. **push( )** accepts pointer to pointer to **struct node** in **s** and **item**, an **int**.

Suppose the address of **top** is 90, which is collected in **s**. A pointer to **struct node**, **q** is declared and memory is allocated using **malloc( )** for the new node whose address is 10 (pointed to by **q**). Now the element 45, is inserted into this node by the statement

```
q -> data = item ;
```

and the link field of the node has been updated to point to **top** through the statement

```
q -> link = *s ;
```

Lastly **top** is made to point to **q** (as it points the new node) through the statement

```
*s = q ;
```

At the time of adding second element 28 to the stack, again memory for a new node is allocated and its address 20 is stored in **q**. 28 is inserted into new node and link field is made to point to the **top** i.e. previous element. Now **top** is updated to point the **top** node pointed to by **q**. On similar lines rest of the two elements gets added to the linked list.

Pop Element From Stack

Let us now see the working of pop operation in detail. The **pop( )** function accepts address of **top**, which gets collected in **s,** a pointer to pointer to **struct node**. The function returns an **int**.

The figure given in the slide shows an initial status of the stack, where **top** is pointing to node with address 60. In the **pop( )** function, first it is checked whether stack is empty. If **top**, i.e. value in **s** is NULL then it indicates that the stack is empty and if so then we have displayed a proper message. But if stack is not empty, then **q** is made to point to the **top** and data from the node at which **q** is pointing to is collected in **item** and **top** is updated to point to node previous to current node. Next node pointed to by **q** is freed using **free( )** and the element removed i.e. **item** is returned.

Display Stack Elements

Now the **displaystack( )** function is straightforward, it simply traverses the list through a **while** loop and displays value of node visited through loop. The loop terminates when **q** reaches end of the list i.e. it becomes NULL.

No. Of Elements In Stack

The working of **count( )** function which returns number of elements (or nodes) is similar to **displaystack( )** function with a slight difference. Here, a counter **c** is incremented by one while traversing through the list and the count **c** is then returned.

Disk - I

In this lecture you will understand:

* The logical organization of the disk
* How to read the contents of boot sector
* What are Boot Parameters

Disk

The diagram given in the slide shows the various parts of a floppy disk.

The slide also explores the hard disk. The advantages of the hard disk are its higher capacity to store data, reliability and faster access to the data in the slide.

Parts of Platter

The platter of disk consists of circular tracks and each track is again divided into sectors. All the sectors are of same capacity.

The slide also displays the specifications of the various types of floppy disks. Every sector consists of 512 bytes.

The size of a disk is calculated by:

Sides * Tracks/side * Sectors/Track * Bytes/Sector / Bytes/Kb

The size of 360 Kb floppy is calculated as shown in the slide.

Logical Organization

A disk is logically organized into sides, tracks and sectors where information is stored. Side 0, Track 0, Sector 1 is the Boot Sector. From second sector onwards, FAT chain entries F1 (first copy of FAT) and F2 (second copy of FAT) begins. Each FAT requires 9 sectors, hence the second FAT's last entry is made in Side 1, Track 0, Sector 1. From Sector 2 onwards there are 14 Directory entries. In all other sectors data can be stored.

Reading Boot Sector

Let us now understand what is a boot sector and how to read it. Boot Sector contains information about how the disk is organized. That is, how many sides does it contain, how many tracks are there on each side, how many sectors are there per track, how many bytes are there per sector, etc. On a 1.44 MB floppy, the Boot Sector is located on side 0, track 0, and sector 1. Using **absread( )** function we have read the Boot Sector of 1.44 MB floppy disk. As a boot sector is of 512 bytes, **arr** is declared as **char** array of that size. The information thus read is then printed through the **printf( )** statement.

Contents of Boot Sector

The Boot Sector consists of two parts: 'Boot Parameters' and 'Disk Bootstrap Program'. The Boot Parameters are useful while performing read/write operations on the disk. The Boot Parameters basically contain information indicating how the disk has been organized.

Boot Parameters

The table given in the slide lists the Boot Parameters along with their byte configuration for a 1.44 MB floppy.

Reading Boot Sector

Let us now see a program that reads the Boot Sector of a 1.44 MB floppy disk. In the program we have declared a structure called **boot**. First few members of the structure are given here with data type that is suitable to store the corresponding information. Add members to this structure for remaining parameters. Thus to store sectors in reserved area a data member could be **sra** with data type as **int** (since, it takes 2 bytes).

Then, we have declared a variable **b** of **boot** structure and called **absread( )** to read the specified sector (i.e. Boot Sector).

Slide Number 8

In the program given in this slide, after collecting the values in elements of **b** they are displayed. Note, how the structured information is displayed using proper format specifiers within **printf( )**.

The jump instruction consists of 3 bytes, hence it is printed using **%x**, format specifier. The System ID is 8 bytes long and is printed using **%c** through the **for** loop and the Bytes/Sector and Sectors/Cluster are printed using **%d**. On similar lines write code for printing rest of the parameters.

Disk - II

In this lecture you will understand:

* How to read a disk
* How to know a disk is virus infected
* What is an Anti-Viral software

In General

The general form or syntax of **absread( )** function is shown in the slide, where the drive no. can be an integer. The integer 0 stands for drive A, 1 = B, 2 = C, etc. The no. of sectors specifies the number of sectors to read. Next parameter specifies the starting logical sector from where reading should begin and the last parameter is the buffer into which the information read is to be stored.

Hence in the statement,

```
absread ( 0, 1, 0, arr ) ;
```

0 is the A drive, 1 sector to be read and 0 is the logical sector number (here boot sector) and **arr** is the buffer.

To read sectors, ROM-BIOS refers them through side, track, sector whereas DOS reads them using logical sector numbers (LSN).

ROM-BIOS refer the drives using the physical configuration whereas DOS uses logical drives starting from 0 onwards as shown in the slide.

What Is It?

The slide shows the typical values and the values obtained when a boot sector is infected. The virus infects the Disk Bootstrap Program (DBS).

Anti-Viral

To disinfect the disk most antiviral softwares read the Boot Parameters and Disk Bootstrap Program (DBS) from the logical sector number (LSN) 50 and writes the sector information in the LSN 1 through **abswrite( )**. The function **abswrite( )** is the counterpart of **absread( )** having same parameters except the function that it performs, it writes the sectors.

Better...

Unnecessarily remembering the logical sector number where the copy of boot sector is stored, we can read the parameters from an uninfected floppy disk and write it to infected floppy disks. The program shown in the slide achieves this. The array of pointer **names** contains some of the names of the viruses.

Directory

In this lecture you will understand:

* How to read directory sectors
* How directory entries are stored on the disk
* How date and time of creation/modification of a file is stored
* What is meant by an attribute of file and how it is stored
* How a file is loaded or saved on a disk

Directory Sector

The root directory of a 1.44 MB disk containing a 12-bit FAT system occupies 14 sectors. The directory sectors contain 32-byte entries for various files/sub-directories present in the root directory. Since each entry is of 32 bytes, one directory sector can accommodate 16 such entries. As there are 14 directory sectors on a 1.44 MB disk, there can be a maximum of 224 entries (16 * 14) in the root directory.

Each 32-byte entry contains either a filename or a sub-directory name. If it is a file entry then it contains information about file's name, its size, attributes, starting location on the disk, etc. The order of and size (in terms of bytes) for this information is shown in the slide.

Printing Directory

The program given in the slide reads the directory sector and displays the information that is read. The directory sector consists of 16 directory entries. The logical sector number for directory entry is 19. We have used **absread( )** to read this sector and collected information in **e**, an array of **entry** structure. We have declared **entry** structure with the elements required to store file information.

Why 2 Bytes?

The table given in the left side of the slide lists the parameters of directory entry. Note that for the entries like date and time only 2 bytes are reserved, whereas, to store a date as 25/01/04, 9 bytes are required. How do these 9-byte strings get converted to 2-byte entries?

The calculation needed to store the date in two bytes is shown in the slide. First the difference between 2004 and 1980 (the year since DOS exists) is multiplied by 512. To this is added the value obtained by multiplying the month by 32. Then the date is added to this sum. The binary equivalent of the resultant sum is placed in the date field in the directory entry of the file.

The first 5 bits represent the days, next 4 bits represent month and the remaining 7 bits represent difference of 1980 and the year (of the date being considered). The bit representation for the date 25/01/2004 is shown in the slide. The bit representation of year, month and days is also verified, for example, the bit representation for the year is 00110000 i.e. $2^4 + 2^3 \Rightarrow 16 + 8 \Rightarrow 24$. 24 is the difference between 1980 and 2004. Hence, 1980 is added to 24, which gives the year 2004.

Similar to date, time is also stored within 2 bytes. The distribution of bits for time is displayed in the slide.

Attribute Byte

The attribute of a file tells whether a file is hidden, read-only, sub-directory, etc. In the attribute byte each bit represents either the type of the file or whether the entry is a sub-directory entry. The 0-bit is used for read only attribute and 1-bit is used for hidden attribute. If bit 0 is set to 1 then the file can only be read, it cannot be modified or deleted. Similarly if bit 1 is set to 1 then the file is made hidden.

The program given in the slide reads the directory sector and writes 2 in the attribute byte to make the file(s) hidden. After setting the attribute, the directory sector is written back on to the disk using **abswrite( )**.

Attribute Byte

The meaning of each bit of attribute byte is shown in the slide. The volume label gets displayed when we execute the command DIR on the command prompt. The volume label is set when we format a disk using DOS command FORMAT. This command creates tracks and sectors, writes Boot

Parameters and Disk Bootstrap Program and lastly asks for volume label to enter. Using DOS command **Label** we can set a new volume label for the disk.

Loading/Saving a File

Suppose a file PR1.C is of 1500 bytes in size. Suppose the starting cluster number for this file is 3. The starting cluster number indicates the place where the file begins in the data space of the disk. The first 512 bytes are stored in sector 3. Then suppose sector 5 is allotted to store next 512 bytes. If the next sector allotted is 8, then the remaining 476 bytes are stored in this sector.

The 32-byte directory entry of this file is shown in the slide. T & D stands for time and date of file and 3 is the starting cluster number and 1500 is the size of the file.

The FAT entry of this file is also shown in the slide. The entry 5 indicates that the remaining bytes of this file begin from sector 5 and the entry 8 signifies that next remaining bytes are stored in sector 8 and FFF signifies the end of the file.

Console I/O

In this lecture you will understand:

* The functions available for Console I/O
* Difference between formatted and unformatted Console I/O
* How to R/W From / To String

Input/Output In C

Though C has no provision for I/O, it of course has to be dealt with at some point or the other. Each operating system has it's own facility for inputting and outputting data from and to the files and devices. It's a simple matter for a system programmer to write a few small programs that would link the C compiler for particular Operating system's I/O facilities. The developers of C Compilers do just that. They write several standard I/O functions and put them in libraries.

The input/output operations in C are done through functions. The slide lists some of the I/O functions that we have used so far.

Slide Number 2

There are numerous library functions available for I/O. As shown in the slide, these are classified into two broad categories—Console I/O and Disk I/O, and Port I/O functions. The Console I/O functions are those, which receive input from keyboard and write output to VDU. The Disk I/O functions perform I/O operations on a floppy disk or hard disk. The Port I/O functions perform I/O operations on various ports.

The Console I/O functions are further classified into two categories—formatted and unformatted console I/O functions. The basic difference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. The functions available under each of these two categories are shown in the slide. Note that as far as unformatted I/O functions are concerned, no standard library functions are available for **int** and **float**.

printf( )

The general syntax for **printf( )** is shown in the slide.

The format string can contain format specifiers, escape sequences, and any other characters. The format specifiers for **char**, **int**s, and **float**s and strings are given in the slide along with various escape sequences that begins with '\'.

Now a question arises as how **printf( )** function interpret the contents of the format string. For this it examines the format string from left to right. So long as it doesn't come across either a **%** or a \ it continues to dump the characters that it encounters, on to the screen. The moment it comes across a conversion specification in the format string it picks up the first variable in the list of variables and prints its value in the specified format.

Formatting Output

Let's understand the format specifications. In the program **a** is initialized with 35. Then using **printf( )** the value in **a** is printed. Note the format specification given in each of the **printf( )** statement. The field-width specifier tells **printf( )** how many columns on screen should be used while printing a value. Thus, **%2d** says, "print the variable as a decimal integer in a field of 2 columns". If the value to be printed happens not to fill up the entire field, the value is right justified and is padded with blanks on the left. If we include the minus sign in format specifier, this means left justification is desired and the value will be padded with blanks on the right.

Formatting Output

Let us now see how to format a **float** value. In the program given in slide, field-width is also used with format specifier **%f**. The format specifier **%7.4f** reserves space for 7 characters on the screen of which 4 are used for digits after decimal point, 1 for the decimal point and remaining for digits before decimal point. Thus, entire space is used for printing the number 35.6927. The format specifier

%8.2f reserves space for 8 characters of which 2 are used for digits after decimal point. The format specifier **%-7.1f** reserves space for 7 characters on the screen and due to '-' sign the value is left justified with blanks padded on the right. Due to precision 1, the fractional value is rounded off to nearest value as the next value is greater than 5. Using **%1.1f**, space is reserved for only 1 character on the screen, but since the value in **a** cannot be accommodated, it uses the space as required displaying only 1 digit after decimal point.

On similar lines, the format specifications are used for **double** and **long double**.

Escape Sequences

Let us now explore the escape sequences.

We have already used **\n** a newline character which takes cursor to the beginning of the next line. The newline character is an 'escape sequence', so called because the backslash symbol (\) is considered as an 'escape' character—it causes an escape from the normal interpretation of string, so that the next character is recognized as one having a special meaning. A program is given in demonstrates the use of escape sequences.

The string "Hello\b\bHi" in **printf( )** prints 'HelHi'. This is because **\b** is a backspace character, which moves the cursor one position to the left of its current position. The string "\nHello\t\tHi" prints the output on new line as **'\n'** (New line) is used and 'Hi' is printed in the third tab zone. A 80-column screen usually has 10 tab stops, which divides screen into 10 zones of 8 columns each. Similarly, "\nHello\nHi" prints 'Hello' and 'Hi' each on a new line. Then "\nHello\rHi" prints 'Hello'. After printing 'Hello', **'\r'** takes the cursor to the beginning of the screen from where onwards the string "Hi" is printed, hence only 'Hello' is displayed on the screen.

"\nHe said \"Let us go\"." prints 'He said "Let us go".'. The **'\"'** character prints double quotes. "\n// \\\\" prints '// \\'. **'\\'** prints a backslash character. "\nHello\t\tHi" prints the output as shown in the slide.

Conclusion

Once again let us have a look at the syntax of **printf( )** and **scanf( )** statements. The format string of **printf( )** statement can have format specifiers, escape sequences or any other characters. The format string is followed by a list of variables, constants or expressions, but is optional.

The **scanf( )** statement also consists of format string and list of variables. The format string must contain format specifiers only and the list of variables is also compulsory. Though it is rarely used, we can use field-width in the format specifier.

Unformatted Console I/O Functions

Let us now switch over to the unformatted Console I/O Functions. There are several standard library functions available under this category—those that can deal with a single character and those that can deal with a string of characters. We often want a function that will read a single character the instant it is typed without waiting for the Enter key to be hit. **getch( )** and **getche( )** are the two functions that serve this purpose. The program given in the slide demonstrates the use of some such functions.

The functions **getch( )** and **getche( )** return the character that has been most recently typed. The 'e' in **getche( )** function means it echoes the character that you typed to the screen. As against this **getch( )** just returns the character that you typed without echoing it on the screen. **getchar( )** works similarly and echo's the character that you typed on the screen, but it requires Enter key to be typed following the character that you typed. We can use **fgetchar( )** as well. The difference between **getchar( )** and **fgetchar( )** is that the former is a macro whereas the latter is a function.

The functions **putch( )**, **putchar( )** and **fputchar( )**, prints a character on the screen. As far as working of these three functions is concerned it's exactly same, however, they can output only one character at a time.

The slide has given the peculiarities of functions that read a character. **getch( )** doesn't wait for the enter key to be pressed while supplying the input character. **getche( )** is same as that of **getch( )** except that it echoes (display) it on the screen.

getchar( ) waits for the enter key to be pressed while supplying the input character. Moreover it's a macro not a function, whereas its counterpart is **fgetchar( )**, which is same as **getchar( )** except that it is a function.

File I/O - I

In this lecture you will understand:

* How Disk I/O operations are performed
* What is buffered I/O
* How to read a file and display its contents
* Why and when to use **typedef**

Disk I/O

Having dealt with the Console I/O functions, let us now turn our attention to Disk I/O functions. As shown in the slide Disk I/O functions can be broadly divided into two categories—High Level/Standard Disk I/O and Low Level Disk I/O. The High Level I/O is categorized into Text and Binary mode I/O.

Again the Text mode is classified into Formatted and Unformatted I/O. In Formatted Text I/O the data of all types (i.e. **char**, **float**, etc.) can be formatted using **fprintf( )** and **fscanf( )** functions. In case of Unformatted Text I/O, the data of **char** type is handled using functions **getc( )**, **fgetc( )**, **putc( )**, **fputc( )**. For **int** and **float** type no standard library functions are available. For strings **fgets( )**, **fputs( )** functions are available which works on line by line basis.

Buffered I/O

When we try to read file contents, the contents are brought into memory and are then read through a program. To write the contents in the file, the contents are first written to the buffer allocated in memory and are then moved to the disk when we close the file. This mechanism is termed as Buffered I/O.

Displaying File Contents

The program given in the slide reads the contents of file and displays them. In this program we asked for the file to be read, opened it, read it character by character till end of file is not encountered and displayed the characters on screen.

We have declared a pointer **fp** of type **FILE**. Each file that we open will have its own **FILE** structure. The **FILE** structure contains information about the file being used, such as its current size, its location in memory, etc. More importantly it contains a character pointer that points to the character that is about to get read. Then we have opened the given file using **fopen( )** function, which accepts two parameters—filename and the mode in which the file has to be opened. Since we are reading the file we have specified "r", which represent read mode. **fopen( )** returns a pointer to the file which is stored in **fp**.

Once, the file is opened, it is read on character by character basis using function **getc( )** function. The characters obtained are then printed using **printf( )**. When an end of the file is reached, the end of the file character is returned. This character i.e. EOF is a macro defined in the 'stdio.h' file. The **FILE** structure is also defined in 'stdio.h', hence we have **#include**d this file in our program. Lastly the file is closed using **fclose( )**. This function deactivates the file.

typedef

New types can be defined through the **typedef** statement. Instead of writing lengthy type names in the declaration of variable, a type can be **typedef**ed with a short name and can be used in the rest of the program.

Once the type **unsigned long int** is **typedef**ed using

typedef unsigned long int uli ;

the variables can be created using

uli i, j ;

Similarly structures can also be **typedef**ed as shown in the slide. The **file** structure is **typedef**ed as **FILE** in 'stdio.h' and later it is used to declare file pointers as shown in the slide.

A Fresh Look

Now again take a look at the program to read the file contents. The **FILE** structure contains a **char** pointer member say **fp**, which points to the first character of the chunk of memory where the file has been loaded.

When the contents are read using the **getc( )** function , the function returns the character and increments the pointer **fp** to the next location. Again when **getc( )** is called the new character is returned and pointer is incremented. This continues until the end of the character is reached.

Tips

The slide summarizes some points about the file operations.

File I/O - II

In this lecture you will understand:

* How to write data to a file
* How to copy contents of one file to another
* How to encode and decode file contents

Copying Files

We can copy the contents of one file into another, as demonstrated in the program given in the slide. This program takes the contents of a text file and copies them into another text file, character by character.

Here, we have opened source and target file, which are pointed to by **fs** and **ft** respectively. The source file is opened in read mode whereas target file is opened in write mode.

The file pointed to by **fs** is read through **getc( )** and the character is written into the file pointed to by **ft** using **putc( )**. The process is repeated until the character read is **EOF**. Lastly, the opened files are closed through the **fclose( )**.

Filecopy

We should always check whether the function **fopen( )** is successful to open specified file. This can be done by checking the value returned by **fopen( )** If **fopen( )** fails to open a file it returns **NULL**. In the program shown in the slide, if **fopen( )** returns NULL while opening the source file we have terminated the program by calling the **exit( )** function. If the function opens the source file but fails to open target file then also we have terminated the program, but before exiting the program we have closed the source file.

We have read the source file character by character and using **getc( )** function and the characters thus read are written to target file using **putc( )** function.

The entire logic of reading / writing given in the program can be replace the code (also given in the slide) given below:

```
while ( ( ch = getc ( fs ) ) != EOF )
    putc ( ch, ft ) ;
```

Here, the character read is checked. If it is not a **EOF** character, then the character is written into the file, otherwise the loop is terminated.

Tips

The following are some important points to remember:

(a) While supplying the filename, the path can be included but the length of the path should not exceed 66 characters.

(b) **fopen( )** returns NULL if file is absent when opened in read mode, otherwise returns the address of the structure.

(c) When **fopen( )** is used to open a file in write mode, then it creates a file if it is absent otherwise if present it overwrites the same.

(d) **fclose( )** not even closes the file but also it adds 26 at the end of the file, if opened for writing.

Coding/Decoding

To code/encrypt a file means to rewrite the contents of file in such a manner so that the contents of the file would not be in readable format or would not contain any meaningful information. On the other hand decoding/decrypting refers to regenerate the file in its original form. Some logic like adding some number to ASCII value of each character of the file, etc. is used while coding the file. While decoding the file the same number is subtracted from the ASCII value of the character read from (coded file) to get original character(s).

The program given in the slide demonstrates how coding/decoding of a file can be done. To code/decode a file, source and the target files are opened in read and write mode respectively. Next a choice has been accepted to know whether to encrypt or decrypt a file. For encrypt or decrypt a file, **encrypt( )** and **decrypt( )** functions are used. After encrypting or decrypting a file, the source file is removed or deleted using **remove( )**.

Offset Cipher

The function definition of **encrypt( )** and **decrypt( )** functions have been given in this slide. In **encrypt( )** function the source file pointed to by FILE pointer **fs** is read character by character. This character is then written in target file after adding 128 to the ASCII value of the character.

In **decrypt( )** function we have read the file character by character. Before writing the character to the target file we have subtracted the same number i.e. 128 from the ASCII value of the character.

Note that the file pointers **fs** and **ft** used in these functions are declared globally so that they can be used within **encrypt( )** and **decrypt( )** functions.

We have chosen an unusual number 128, instead of using simple numbers like 1, 2, 3 etc. This is because if 1 is added to **ch** then the contents can be easily decoded, but on adding 128 results graphical characters which are difficult to decode. This way of encrypting / decrypting a file through an offset is called offset ciphering.

Substitution Cipher

Instead of using offset cipher one can use substitution cipher i.e. by substituting a different character in place of original character. The function given in the slide uses substitution cipher to encrypt a file. Here, the corresponding characters of **str1** are substituted with those of **str2**. Each time a character **ch** read from the file is checked against the character of **str1** and on finding it the corresponding character of **str2** is written into the file.

More File I/O - I

In this lecture you will understand:

* How to remove blank lines from a file
* How to read and write records from/to the file
* Various file opening modes
* Difference between text mode and binary mode file I/O

Remove Blank Lines

The program given in the slide removes blank lines from a specified file. To remove blanks from a file we need two files, the source from which blank lines are to be removed and the target into which non-blank lines are to be written. Hence two files are opened pointed to by **fs** and **ft**.

Each time a string is read through

```
fgets ( str, 79, fs )
```

The line thus read in **str** is send to a user-defined function **isblank( )**. If the given string i.e. **str** contains a blank line then **isblank( )** returns BLANK (A macro defined with value 1) otherwise it returns NOTBLANK (again a macro defined with value 0). If the returned value is NOTBLANK then we are writing the line i.e. **str** into the target file using **fputs( )**. The **fgets( )** function is similar to **gets( )** except it works on files and requires the length and the file pointer. The **fputs( )** function need not require the length of the string.

Lastly, the while loop gets terminated, we have closed files. We have removed the source file s and renamed the target file with the same name as that of the source file.

Slide Number 2

The function definition of **isblank( )** is given in this slide. Here, through a **while** loop each character of the string pointed to by **p** is read. Each character of the file thus read is then checked for whether it is a space or tab or new line character. If any of these characters are found then the loop continues by incrementing the pointer otherwise **NOTBLANK** is returned. If the entire string gets scanned, and the control reaches outside the **while** loop, then **BLANK** is returned.

Handling Records

Let us now see how records can be stored to a file. The program given in the slide demonstrates how records containing information about employee can be written to or read from a file.

Here, we have declared a structure called **employee**. Then we have opened a data file called 'emp.dat' for writing. A **while** loop runs as long as **ch** contains 'y' (i.e. till we wish to add records). To input data for record we have used **scanf( )** and the record is written into the file pointed to by **fp** using **fprintf( )**, which writes into the file and have similar syntax as that of **printf( )** except the file pointer **fp**. After writing the record the user is prompted whether to enter a record or not through the 'Add another y/n' message and accepts a character **ch**, if it is not equal to 'y' then the loop is terminated and the file is closed.

Slide Number 4

This slide contains the code to read the records from a file. Again the file is opened in read mode. A **while** loop runs, whereby, in each iteration of this loop a record is read from the file using **fscanf( )**. The syntax of **fscanf( )** is similar to that of **scanf( )** except it needs a pointer to **FILE**. The record thus read is then displayed on the screen using **printf( )**. Lastly the file is closed using function **fclose( )**.

Why The Difference

The size of a record shown in the slide is 26 bytes. However, if we write this record on disk using **fprintf( )**, then its size turns out to be 29 bytes. Since the file is opened in text mode for reading and writing (i.e."wt"and "rt") the number 4500.50 requires 7 bytes, as there are 7 characters present in it.

Instead if we read / write the records using **fread( )** / **fwrite( )** in binary mode (**"rb"** / **"wb"**), they store numbers more efficiently.

File Opening Modes

Let us now have a look at the modes available for opening a file.

"w" is equivalent to **"wt"**, used for writing to a file in text mode. **"wb"** is used for writing to a file in binary mode. When opened the file with this mode a new file is created if the file is not present on the disk. If the file is present, the existing file is overwritten. The operation possible is writing to the file.

Similarly for reading a file we have modes like **"rb"** and **"rt"** used to read a file in binary and text mode respectively. When opened the file with this mode NULL is returned if the file is not present on the disk. If the file is present, the file is loaded into the memory and pointer is set to point to the first character in file. The operation possible is reading from the file.

To open a file in append mode we can use **"ab"** and **"at"**. When opened the file with this mode a new file is created if the file is not present on the disk. If the file is present, it is loaded into the memory and pointer is set to point beyond the last character in file. The operation possible is writing at the end of the file.

Slide Number 7

This slide summarizes the modes available for opening a file.

More File I/O - II

In this lecture you will understand:

* How to recover data from a virus infected file
* How to change an internal DOS command called **DIR**
* How to perform low-level Disk I/O

Delete

Our aim is to perform the steps listed in the slide to remove the Jerusalem virus from an infected file.

The Program

The steps listed in earlier slide are carried out in the program given in this slide. Here, a **char** array **s[ ]** is declared and initialized with the signature byte values of the Jerusalem virus. The file 'WS.EXE' is opened in binary mode for reading. Then using **fread( )** first 10 bytes of file are read and each byte is checked with the corresponding signature byte. If anyone of the byte is mismatched with the signature bytes, the file is closed and program terminates.

If all the bytes are found to be matching with the signature byte, then it indicates that the file is infected and hence the program proceeds towards removal of the first 1701 bytes of the file. The code to recover the file is given in next slide.

Slide Number 3

A temporary file called 'TEMP.EXE' is opened in binary mode for writing. Now the first 1701 bytes are bypassed by placing the file pointer past 1701 bytes using **fseek( )**. The macro **SEEK_SET** is used to move the file pointer to the specified position.

The remaining characters of the file are now written into 'TEMP.EXE'. Both the files are closed, 'WS.EXE' is removed and 'TEMP.EXE' is renamed as 'WS.EXE' using **rename( )**.

Changing Commands

We can change the name of an internal DOS command by a C program. The internal commands are stored in a file called 'Command.com'. The program shown in the slide attempts to change DIR command.

Here, 'command.com' is opened in binary mode for reading and writing. The file is read and checked for the three consecutive letters of command through consecutive **if**s. Once, the consecutive letters DIR are found, the file pointer is moved three characters back and made to point to D of DIR. Then using **putc( )** DIR is replace by three characters 'Y', 'P', 'K'. To proceed as usual, we have called **fseek( )**. This is necessary as the sequence is altered. After changing the command, file is closed.

Note: While writing the program provide the correct path of 'Command.com' file. After executing the program, run command.com file and then run YPK command.

Low level Disk I/O

Let us now see how to use low-level disk I/O functions.

We have already discussed a program that copies contents of one file to another. But, the program could copy only the text files and not the .EXE or .COM files. This was because .EXE and .COM are binary files. If such files are opened in text mode and read character by character using **fgetc( )** then if the file contains a character with ASCII value 26 then copying would stop abruptly as **fgetc( )** would return EOF. So we need to copy .EXE/.COM files differently.

Secondly, in the earlier program need to compile the program every time to copy files and the program used to prompt us to enter the source and target file names. We want a program that should copy .EXE/.COM files, should not be required to be compiled every time and we must be able to supply the filenames at the command prompt (as we do in DOS commands like COPY, MOVE, etc.). The program given in the slide shows how this can be done.

The arguments that we pass on to **main( )** at the command prompt are called command line arguments. Here, the function **main( )** has two arguments viz, **argc** in which count of command line

parameters is stored and an array of pointers to strings **argv** which holds the addresses of the command line parameters.

Then in **main( )**, **argc** is checked for the necessary arguments, if not, proper messages are displayed and program is terminated. Next the source file whose name is stored in **argv[1]** is opened using **open( )** function for reading in binary mode. The syntax of **open( )** is similar to **fopen( )**, except the 'O-flags' used to open file. If the **open( )** function is successful to open the file then it returns handle (non-zero value) to the file, otherwise, on failure it returns –1. Hence we have checked for the return value and if it is found to be -1, then we have terminated the program after displaying proper messages.

Next the target file whose name is stored in **argv[2]** is opened in write mode.

Slide Number 6

If the **open( )** function fails to open target file then we closed the source file, displayed proper message and terminated the program.

Next, if the function **open( )** is successful to open target file, then a **while** loop runs till end of file is not encountered. In every iteration of this loop certain number (i.e. 512) bytes are read and stored in buffer. This we have done using **read( )** function. This function needs the file handle from which the contents are to be read, the buffer and the size of the buffer and it returns the number of bytes read. The returned value **n** is used to write the contents read into the target file. The syntax of **write( )** is exactly similar to **read( )** except, the operation. After writing the whole contents both the files are closed using **close( )** function.

The variable **n** and **char** array **buffer[ ]** should be declared as shown in the slide. Also to be able to use low-level disk I/O functions, **#include** the files whose names are given in the slide.

Miscellany

In this lecture you will understand:

* Introduction to Enumerations, Structures and Unions
* What are bitwise operators
* The utility of << (Left-Shift) and >> (Right-Shift) operators

Enumerations

The enumerated data type gives you an opportunity to invent your own data type and define what values the variable of this data type can take .

As an example, one could invent a data type called **ms (**marital status**)** which can have three values—single, married or divorced .Here married has same relation to the variable **ms** as the number 15 has with an integer variable.

Another example is a data type called color which can have three values—red, green or blue.

The format of the **enum** definition is similar to that of structure.

(a) The first part declares the data type and specifies its possible values. These values are called 'enumerators'.

(b) The second part declares variables of this data type.

Internally compiler treats enumerators as integer .Each value on the list of permissible values corresponds to an integer starting with 0.Thus in our example single is stored as 0, married is stored as 1 and divorced is stored as 2 .

Structures

Observe carefully the code given in the slide. Here, a variable **z** of **struct a** is declared and the member **z.i** is assigned value 512. The memory organization of **z** shows that all the members are stored in contiguous memory locations. The size of **z** comes out to be 4, which is the sum of sizes of the individual numbers of the structure. Again on printing the individual members prints 512 for **z.i** and garbage value for **z.ch[0]** and **z.ch[1]**.

Unions

Let's see one more derived data type, the Unions. Unions are derived data types, the way structures are. The syntax of declaring, defining and accessing the members of union is similar to structures. For unions, **union** keyword is used.

In the program given in slide, a **union a** is declared with same members as declared in the structure in earlier slide. Then a variable **z** is declared. The member **z.i** is assigned value 512. The memory organization of **z** is shown in slide. Memory is reserved for **z.i** only and the same memory is shared for **z.ch[0]** and **z.ch[1]** as shown in the slide.

On printing the size of **z**, it prints 2. The value of **z.i** is printed as 512. From the binary equivalent of 512 (stored in memory as shown in the slide), for **z.ch[0]** the number equivalent to the binary representation stored in low-order bit of **z** is printed and for **z.ch[1]** the number equivalent to the binary representation stored in high-order bit of **z** is printed. This clarifies that union permits access to the same memory locations in more than one way.

How Many Bytes

Now, a question arises as memory of how many bytes is allocated for a variable of union. The program given in the slide clarifies this.

Here, a union **a** is declared which contains three members, a **double**, **float** and a **char** array of size 5. Then a variable z is declared of type union **a**. The size of this variable is then printed which turns out to be 8 bytes. All the 8 bytes are used for the **double** variable **d**. The same 8 bytes are shared by **z.f[0]** and **z.f[1]** (4 bytes each) and again the lower 5 bytes are used by **z.ch[]** array. Thus, it is clear from this arrangement that the size of the union is the size of the longest member of the union.

Utility

Utility of union and structure together is given in this slide .This slide shows that there can be union in a structure or a structure in a union .In this way we can also do better memory management

Bitwise Operators

One of C's powerful features is a set of bit manipulation operators. These permit the programmer to access and manipulate individual bits within a piece of data. Let us now find out the utility of bitwise operators.

On declaring a **char** variable **ch**, 1 byte or 8 bits are reserved in memory. We can check whether any bit is 1 or 0 or we can set any bits to 0 or 1.

~ (Tilde) is the one's complement operator. On taking one's complement of a number, all 1's present in the number are changed to 0's and all 0's are changed to 1's. Thus, as shown in the slide, the one's complement of 32 i.e. 00100000 is 11011110 which is a binary equivalent of a negative number. Hence 1 is subtracted from the resultant and again one's complement is taken which results -33.

As, one's complement operator changes the original number beyond recognition, one potential place where it can be effectively used is in development of a file encryption utility as shown in the slide.

Bitwise Operators

Let us now see the working of bitwise 'left shift' (<<) and 'right shift' (>>) operator. The right shift operator shifts each bit in the operand to the right. The number of places the bits are shifted depends on the number following the operand. Note that as the bits are shifted to the right, blanks are created on left. These blanks are always filled with zeros.

Thus, on right shifting 32 by 2 bits the right most 2 bits are dropped and from the left 2, 0-bits are appended hence the resulting value would be 8.

The working of 'left shift' operator is similar to 'right shift' operator, the only difference being that the bits are shifted to the left, and for each bit shifted, a 0 is added to the right of the number. Thus, on left shifting 32 by 1, the left most bit is dropped and one 0-bit is added at the beginning, hence prints 64.

Utility Of << & >>

Having acquainted ourselves with the left shift and right shift operators, let us now see the practical utility of these operators.

As discussed earlier, the date on which a file is created is stored as a 2-byte entry in 32-byte directory entry of that file. For example, for the date 06/01/99 the calculation is shown in the slide. The binary equivalent of the resulting value 9766 is stored in the two bytes. The bit configuration is shown in the slide.

d, m, y

The program given in the slide, extracts date, month and year from the value 9766 (which is a result of calculation done in previous slide).

Here, this value is stored in an **int** variable **dt**. To get year as a separate entity, we have right shifted the value in **dt** by 9. Similarly, left shifting **dt** by 7, followed by right shifting by 12 yields month. Lastly to extract the day, **dt** is left shifted by11 and then right shifted by 11. Finally the extracted values are printed for confirmation.

Bitwise Operators

In this lecture you will understand:

* The utility of & (Bitwise AND) operator
* How to change attributes of a file using bitwise operators
* How to create function pointers

Bitwise Operators

Some more bitwise operators are there like **&** (Bitwise AND), | (Bitwise OR), ^ (Bitwsie XOR). The truth tables of these operators are shown in the slide.

Utility of &

The bitwise AND operator **&** is used to check whether a particular bit is on / off. The program shown in the slide checks whether the bit number 3 of the given number **n** is on / off. Since we want to check the bit number 3, the second operand for the AND operator should be $1 * 2^3$ which is equal to 8. If on ANDing the result turned out to be 8, then the bit number 3 is on and if the value is 0 then the bit is off. If the bit is on, then **n** is ANDed with 0xF7, the number formed by setting bits to 1 except the third bit, that is set to 0.

Slide Number 3

In every 32-byte file entry present in the directory, there is an attribute byte. The status of a file is governed by the value of individual bits in this attribute byte. The AND operator can be used to check the status of the bits of this attribute byte. The program given in the slide demonstrates how this can be done.

Here, we have read the directory sector of floppy disk, and for all the 16 directory entries the read only (0^{th} bit) and hidden (i.e. 1^{st} bit) attribute is checked. If the read only bit is found to be on then the bit is put to off by ANDing with 0xFE. The 0^{th} bit is set to off.

Similarly for hidden attribute the second bit is checked and if the hidden bit is not set it is set by using OR operator. Lastly the sector is rewritten to the floppy disk.

Calling Functions

To be able to pass control to the TSRs, we must know a mechanism by way of which we would be able to transfer control to a routine by merely knowing the address of the routine. This mechanism is nothing but a pointer to a function. Let us see how to declare such pointer and use it to call a function.

In the program given in the slide, we have defined a function called **display( )**. We have also declared its prototype. Then we have declared a pointer **p**. Note the way p has been declared. **p** is a pointer to such a function which neither receives any parameter nor returns any value. In other words **p** is a pointer to the function **display( )**.

Then we have called function **display( )**. To call the same function, using pointer, we have stored the address of **display** in **p** through the statement,

```
p = display ;
```

and the function is called through the statement

```
( *p )( ) ;
```

which calls **display( )** as **p** holds its address.

Some Definitions

Let us try to determine the meaning of the declarations shown in the slide.

void (*p)( ) ; Here **p** is a pointer to a function that returns nothing.

void *p( ) ; Here **p** is a function that returns a **void** pointer.

void (*p)(int, float) ; Here **p** is a pointer to a function that receives an **int** and **float** and returns nothing.

void (*p)(int *, char **) ; Here **p** is a pointer to a function that receives an **int** pointer and pointer to **char** pointer and returns nothing.

int **(*p)(char **, float *) ; Here **p** is a pointer to a function that receives pointer to **char** pointer and a **float** pointer and returns pointer to **int** pointer.

Define

Few more examples of how a pointer to a function can be defined are given in the slide.

Having understood try to give the interpretation of the statement,

```
int * ( *p[3] )( int *, int ** )
```

The declaration means that **p** is an array of three pointers to functions that receive **int** pointer and pointer to an **int** pointer and returns an **int** pointer.

Hardware Interaction - I

In this lecture you will understand:

* Ways to interact with hardware
* Hardware interaction, DOS perspective
* Hardware interaction, Windows perspective
* What are Ports, CMOS
* How to access CMOS data

Hardware Interaction

Interaction with hardware suggests interaction with peripheral devices. However, interaction may also involve communicating with chips present on the motherboard. other than the microprocessor. During this interaction one or more of the following activities may be performed:

(a) Reacting to events that occur because of user's interaction with the hardware. For example, if the user presses a key or clicks the mouse button then our program may do something.

(b) Explicit communication from a program without the occurrence of an event. For example, a program may want to send a character to the printer, or a program may want to read/write the contents of a sector from the hard disk.

Let us now see how this interaction is done under different platforms.

User Initiated H/W I/A, DOS Vs Win

The slide distinguishes hardware interaction under DOS and Windows.

Under DOS whenever an external event (like pressing a key or ticking of timer) occurs a signal called hardware interrupt gets generated. For different events there are different interrupts. As a reaction to the occurrence of an interrupt a table called Interrupt Vector Table (IVT) is looked up. IVT is present in memory. It is populated with addresses of different BIOS routines during booting. Depending upon which interrupt has occurred the Microprocessor picks the address of the appropriate BIOS routine from IVT and transfers execution control to it. Once the control reaches the BIOS routine, the code in the BIOS routine interacts with the hardware. Naturally, for different interrupts different BIOS routines are called.

Under Windows too a hardware interrupt gets generated whenever an external event occurs. As a reaction to this signal a table called Interrupt Descriptor Table (IDT) is looked up and a corresponding routine for the interrupt gets called. Unlike DOS the IDT contains addresses of various Kernel routines (instead of BIOS routines). These routines are part of the Windows OS itself. When the kernel routine is called, it in turn calls the ISR present in the appropriate device driver. This ISR interacts with the hardware.

Prog. Initiated H/W I/A, DOS Vs Win

As shown in the slide, different methods are used to interact with the hardware under DOS and Windows environment.

(a) Directly interacting with the hardware

At times the programs are needed to directly interact with the hardware. For example, while writing good video games one is required to watch the status of multiple keys simultaneously. The library functions as well as the DOS/BIOS functions are unable to do this. At such times we have to interact with the keyboard controller chip directly.

For this one has to have good knowledge of technical details of the chip Moreover, not every technical detail about how the hardware from a particular manufacturer works is well documented.

(b) Calling DOS Functions

To interact with the hardware a program can call DOS functions. These functions can either directly interact with the hardware or they may call BIOS functions which in turn interact with the hardware. However, since DOS functions do not have names they have to be called through the mechanism of interrupts. This is difficult since the programmer has to remember interrupt service numbers for calling different DOS functions.

(c) Calling BIOS Functions

DOS functions can carry out jobs like console I/O, file I/O, printing, etc. For other operations like generating graphics, carrying out serial communication, etc. the program has to call another set of functions called ROM-BIOS functions which are to be called using interrupts and involve heavy usage of registers.

(d) Calling Library Functions

We can call library functions which in turn can call DOS/BIOS functions to carry out the interaction with hardware. Good examples of these functions are **printf( ) / scanf( ) / getch( )** for interaction with console, **absread( ) / abswrite( )** for interaction with disk, **bioscom( )** for interaction with serial port, etc.

Under Windows explicit communication with hardware is much different than the way it was done under DOS. This is primarily because under Windows every device is shared amongst multiple applications running in memory. To avoid conflict between different programs accessing the same device simultaneously Windows does not permit an application program to directly access any of the devices. Instead it provides several API functions to carry out the interaction.

When we call an API function to interact with a device, it in turn accesses the device driver program for the device. It is the device driver program that finally accesses the device.

Calling BIOS / DOS Routines

Steps required in calling BIOS / DOS routines are as follows -:

BIOS/DOS routines do not have names. They are invoked through 'interrupts'. Hence, first an interrupt is issued. An interrupt is a signal to the microprocessor that its immediate attention is needed. The addresses of BIOS/DOS routines are stored in the Interrupt Vector Table (IVT) in DOS's base memory. To reach the location where the address of the interrupt handler is stored, the interrupt number is multiplied by 4. Each address is of 4 bytes in size, hence the interrupt number is to be multiplied by 4.

After reaching the location, the address of the handler routine is picked and the current values in CPU registers are stored onto the stack so that they can be restored when the routine terminates. The CPU registers are then set with the values needed by the handler routine and the control is transferred to handler routine for execution. Since these routines serve the interrupts they are called **'Interrupt Service Routines'.**

After executing the routine, the values returned by the routine are collected into ordinary variables and the values in stack are restored to the CPU registers. The process of restoring values from the stack is called pop operation. After popping the values the original interrupted program resumes.

Calling Dev. Dri. Routine - Windows

The figure in the slide shows how interrupt mechanism is handled under Windows.

As explained earlier, under Windows a hardware interrupt gets generated whenever an external event occurs. As a reaction to this signal a table called Interrupt Descriptor Table (IDT) is looked up and a corresponding routine for the interrupt gets called. Unlike DOS the IDT contains addresses of various Kernel routines (instead of BIOS routines). These routines are part of the Windows OS itself. When the kernel routine is called, it in turn calls the ISR present in the appropriate device driver. This ISR interacts with the hardware.

I/A with Disk

To actually read the contents of boot sector of the floppy disk the program makes a call to a user-defined function called **ReadSector( )**.

The first parameter passed to **ReadSector( )** is a string that indicates the storage device from where the reading has to take place. The syntax for this string is **\\machine-name\storage-device name**. The second parameter is the logical sector number. We have specified this as 0 which means the boot sector in case of a floppy disk. The third parameter is the number of sectors that we wish to read. This parameter is specified as 1 since the boot sector occupies only a single sector. The last parameter is the address of a buffer/variable that would collect the data that is read from the floppy. Here we have passed the address of the boot structure variable **b**. As a result, the structure variable would be setup with the contents of the boot sector data at the end of the function call.

ReadSector( ) function begins by making a call to the **CreateFile( )** API. The **CreateFile( )** function opens the specified device as a file.

The first parameter of **CreateFile( )** function is the string specifying the device to be opened. The second parameter is a set of flags that are used to specify the desired access to the file (representing the device) about to be opened. By specifying the **GENERIC_READ** flag we have indicated that we just wish to read from the file (device). The third parameter specifies the sharing access for the file (device). Since floppy drive is a shared resource across all the running applications we have specified the **FILE_SHARE_READ** flag. The fourth parameter indicates security access for the file (device). Since we are not concerned with security here we have specified the value as **0**. The fifth parameter specifies what action to take if the file already exists. When using **CreateFile( )** for device access we must always specify this parameter as **OPEN_EXISTING**. Since the floppy disk file was already opened by the OS a long time back during the booting. The remaining two parameters are not used when using **CreateFile( )** API function for device access. Hence we have passed a **0** value for them. If the call to **CreateFile( )** succeeds then we obtain a handle to the file (device).

The device file mechanism allows us to read from the file (device) by setting the file pointer using the **SetFilePointer( )** API function and then reading the file using the **ReadFile( )** API function. The first parameter to **SetFilePointer( )** is the handle of the device file that we obtained by calling the **CreateFile( )** function. The second parameter is the byte offset from where the reading is to begin. We have specified the third parameter as **FILE_BEGIN** which means the byte offset is relative to the start of the file.

The first parameter to **ReadFile( )** function is the handle of the file (device), the second parameter is the address of a buffer where the read contents should be dumped. The third parameter is the count of bytes that have to be read. The fourth parameter to **ReadFile( )** is the address of an **unsigned int** variable which is set up with the count of bytes that the function was successfully able to read. Lastly, once our work with the device is over we should close the file (device) using the **CloseHandle( )** API function.

Once the contents of the boot sector have been read into the structure variable **b** we have displayed the first few of them on the screen using **printf( )**.

More Calls

Since capacity of hard disks is huge, logical partitions are created on it to accommodate different operating systems. The information about where each partition begins and ends, the size of each partition, etc. is stored in a partition table (PT) in side 0, track 0, sector 1. This sector also contains a Master Boot Program. The partition table indicates which is bootable partition. The partition may contain Windows whereas the other might contain Linux and so on. The partition table as shown in the slide stores Master Boot Program. Thus, a PT consists of data and code part. The data part begins at 447^{th} byte. The last two bytes in the PT are always 0x55, 0xAA. The data part is 64 bytes and is further divided into 4 parts of 16 bytes each. Each 16-bytes chunk consists of information about a partition on the hard disk.

Ports

Different devices are connected at different port addresses as shown in the slide. A device usually has many device register (locations within a physical device). Each device register is assigned a unique address within the I/O address space. Thus a device ends up working in a range of addresses.

A Port is a particular location in the I/O space. Slide shows how microprocessor interacts with memory and I/O space. Ports are used to connect external devices to the computer. There exist several types of ports like serial port, parallel port, USB port, AGP port. The 16-bit address bus (Even for windows the I/O address bus remains 6 bits) really speaking is a part of 20-bit address bus, each address is physically connected to some hardware. For example, the address 378h (LPT1) is mapped on to the 25 pin female connector. The dotted line in the slide indicates there is circuitry. For writing the data to the device connected to the parallel port the microprocessor places the address on the address bus followed by the data on the data bus.

Interaction With CMOS

The CMOS chip as shown in the slide, has a SRAM type of memory. This memory is special in that it can retain information for a long duration while consuming negligible amount of power. The CMOS chip's memory is used in a computer to retain critical hardware information like system time, system date, amount of base memory, hard disk type, etc. The chip retains this information even after the computer has been switched OFF. This is because it has a separate auxiliary battery support that powers it after shut down.

This chip provides two interfaces—control register and a data register. Both of these registers are I/O mapped. The control register is used to instruct the CMOS chip to return specific information on the data register. The port address of the CMOS control register is 70h, and that of the data register is 71h. The value that is written on the control register will decide what data is returned on the data register. For example, if we write 00h at 70h, the CMOS chip in return places the seconds part of the system time at port 71h. Similarly when 02h is placed at 70h, minutes are obtained at 71h. The table given in the slide shows a more such values.

Accessing CMOS Data

The program discussed in the slide accesses the data i.e. system date and system time from CMOS chip.

Here, we have called **outportb( )**, which takes two parameters, the address of the control port, and the second parameter is the value being send to the port, which here is 0x70 and 0x00 respectively. Next, we have called **inportb( )** to collect value from port 0x71. The seconds returned by this function have been collected in **sec**. On similar lines, we have collected data for minutes, hours and day of the month.

Slide Number 11

Here, we have collected data for month, year and day of the week, by calling **outportb( )** and **inportb( )** functions. Lastly, the values thus collected have been printed. To print the appropriate name of the day of week, we have used **days** array, which has been initialized with names of the days of a week. Note that day of week returned by **inport( )** function is an integer. The format specifier used in **printf( )** is **%x** (i.e. hex) because the values are available in BCD (Binary Coded Decimal form).

Hardware Interaction - II

In this lecture you will understand:

- Serial communication using null modem
- How parallel port works
- How to communicate with parallel port programmatically
- How speaker works
- How to play tunes using ports

Serial Communication

A common use of the "serial-port" device is to connect two computers together for transfer of data. In this case pins other than transmit or receive are looped back (connected) to the other pins. Looping back is something like fooling the computer to believe that it is actually connected to some serial device. The slide shows the pins of connector for a null modem cable, which uses loop back technique.

The "serial-port" device is commonly used by software developers to connect two computers for testing new software. One computer runs the software while the other records the information sent by the software. This method is often known as debugging, it helps the developers in finding errors in their program. The cable used for connecting the two computers is known as Null Modem cable, Debug-cable, COM port to COM port cable or simply Serial cable.

The communication is called serial communication because only 1 bit gets transmitted at a time. The looping back of pins makes the computer think that a modem (null-modem) is actually connected to it. All COM ports are serial - some are 9 pin some are 25 pin male connectors. All parallel ports are 25-pin female connectors.

Server

In the program given in the slide, first we have called **bioscom( )** function. The first parameter passed to this function specifies port number. The value 0 used here, specifies the communication port COM1. The meaning of the values used in place of second parameter is given in the slide. The second parameter specifies the port number where operation is to be performed. The last parameter is used to check whether data has been transmitted in an error free manner. We are not using parity hence we have specified a value of 0. In the first call to **bioscom( )**, we have use 0x80 i.e. 1200 baud.

Then a **while** loop runs, in which we have again called **bioscom( )** function to know the status. We have checked whether status retrieved indicates data is ready at communication port. If it is then we have called **bioscom( )** twice, one to receive data and the other to send character **ch** (received through **getche( )**). The loop continues till we have not pressed ESC key.

Client

In the program given in the slide, we have called **bioscom( )** with same values as discussed in the earlier slide. Then a **while** loop runs till a key is not hit. In this loop, we have called **bioscom( )** to send a data. The same function is called again to know the status of communication port. If the status is ready, then we have called **bioscom( )** again to receive data. Before collecting the data received in **ch** we have anded (using bitwise **&** operator) it with 0x007F. This is because, when we send a character to **bioscom( )** it splits it into bits, adds a stop bit and then sends it bit by bit. Data read is 16 bit. Of this, lower 7 bits contain the actual data. Rest of the bits contains command completion flags like time-out, bit lost etc.

Lastly, the data received is displayed. Client continues receive values till a key is not hit on the client. Once hit client terminates so server-sending values has no meaning. Hence server loop (i.e. in the server program discussed in earlier slide) is terminated by hitting Esc key.

Parallel Port Basics

Parallel ports can send or receive a byte (8-bit) at a time. Unlike the serial port, these 8-bits are transmitted parallel to each other. Parallel port comes in the form of a 25-pin female connector. Parallel ports are popularly used to connect printer, scanner, CD writer, zip drive, external hard disk drive, tape backup drive, etc.

Each parallel port device consists of three device registers—data, status and control. These registers are mapped in the I/O address space and hence have port addresses. The port addresses for the three registers are in sequential order. That is, if the data register is at address 0x408, the corresponding status register is at 0x408 + 1 and the control register at 0x408 + 2.

The parallel port device registers are mapped to a 25 pin female connector that is present on the backside of the CPU box. The 25 pins of the connector are mapped onto the three device registers. The pins are the device register's interface with the outside world.

The ground pins (18 to 25) remain at a constant voltage of 0 volts. Ground pins provide the zero reference voltage for the connected peripherals. That is, these pins are used by the computer and the peripheral device to agree on a common 0 volts signal. All communication between the computer and peripheral device will be with respect to this 0 volts signal.

Data pins (2 to 9) are true logic pins. The data port pins are the actual data carriers of the parallel port device. Whatever data we wish to send to the peripheral device is transmitted via these pins.

I/A With Parallel Port

To better understand the working of the parallel port device let us write a simple program. Let us connect a Multi-meter across a data pin as shown in the figure given in the slide.

As we can see, pin 2 (1st data pin) is connected to the positive end of Multi-meter. The other end of the Multi-meter is connected to the pin 25 (ground pin). Parallel ports work on TTL - Transistor to Transistor logic. It always has a voltage range from 0 to 5v. Hence parallel port cables cannot be longer because in transmission losses the voltage may drop. For serial communication the range is upto 50v. Hence the cables can be longer.

Now the only thing required would be to provide +5 volts. This means that we would have to turn the pin 2 (first bit of data register) on. The program given in the slide shows how to achieve this.

In this program out of 3 ports associated with parallel port device we are using only the data port. The value written onto port 0x378 would go to pins 2-9. The first call to **outportb( )** writes 0 to the data register. This sets up all the data pins to 0. Then the function **getch( )** has been called for the user to hit a key. The second call to **outportb( )** sends value 1 to the data register of the parallel port. Since the binary representation of 1 is 00000001 only the first pin of the data register receives +5V signal and rest of the data pins remain at 0V.

This example shows how simple it is to communicate with the parallel port through the 'C' programming language.

Program

Let us now put our knowledge of parallel port programming to work with a simple Seven Segment (7-segment) display. Here we would connect the 7-segment display to the PC parallel port and then display numbers from 0 – 9 on it using a loop.

The 7-Segment Display has totally 10 pins—5 at the top and 5 at the bottom. Of the 5 pins on either end the center pin is the common ground. The balance 8 pins can be connected to the 8 data lines from the parallel port connector through 8 resistances each of 560 Ohms as shown in figure given in the slide. The ground pins of the 7-segment display are connected to pin number 25.

Each bit of the data register is responsible for glowing one segment of the display. With this information available at our disposal we can easily manage to display different digits on a 7-segment display.

The program given in the slide shows how this can be achieved. In this program we have first constructed an array of characters. The array contains values for forming the digit by glowing suitable

segments of the 7-segment display. These values are obtained by performing bitwise OR operations on individual segment values.

Slide Number 7

The program then uses a **while** loop within which we have called the standard library function **outportb( )**. To **outportb( )** we have passed two parameters, first one is the port address of the data register and the second one is the actual value we want to write to the port. The condition **if (i == 10)** is used to reset the array index once it reaches 10.

On execution the program will display digits from 0 to 9 in a cyclic manner, with each digit being displayed for exactly 1 second (1000 milliseconds).

Working of Speaker

The speaker is made to vibrate by the electrical impulses sent to it by the PC. These vibrations set the air particles around the vibrating source in motion. As the particles bump into one another a sound is produced. The figure given in the slide shows the working of a speaker.

The cylindrical bar (surrounded by the white coil) is actually a soft iron core. When the coil around the core receives a pulse the soft iron core temporarily becomes a magnet. This temporary magnet interacts with the permanent magnet. This interaction causes the diaphragm to be pushed forward. When the pulse dies the magnetism is lost and the diaphragm returns back to its original position. When this happens frequently we hear a sound. There are two parameters that govern the sound that the speaker produces-frequency of the sound and its duration.

Speaker Operation

The sound frequency is controlled by the 8253 programmable Timer chip, whereas the duration has to be controlled programmatically.

The timer chip produces signals at a default frequency of 1.19318 MHz (119318 x 10^6 cycles per second) through a clock present inside the timer chip. This frequency can be changed programmatically.

The timer chip provides three independent channels. Through these channels signals can be sent to separate devices. The speaker uses the signal coming from channel number 2. The timer chip can operate in six different modes, each mode deciding the nature of the signal produced. For example, while working in mode number 3 it always generates a square wave output.

We can communicate with the timer chip through any of the four I/O ports with addresses 64 through 67, which we shall discuss in next slide.

Speaker Circuit

This slide explains the working model of the speaker circuit.

CS, OUT2, CLOCK shown in the yellow block are the pins present on the chip. CS stands for Chip Select. Its purpose is to enable/disable the chip.

There are 3 channels in 8253—OUT1, OUT2 and OUT3. The channel is a fancy term for number of inputs or outputs. The OUT2 channel indicates that we are using second channel. Each channel can independently drive a device.

The Pulse is sent through channel2 to OUT2 and is sent to the speaker. To manage time interval PIT needs a clock frequency. The quartz crystal generates a base frequency of 1.19318 Mhz. The 8253 chip itself cannot generate the frequency but depends upon an external crystal to generate the frequency. Generated frequency from the crystal is fed to the clock pin of the 8253 IC.

The 8253 chip features a port 66 (also called the count port) to supply a numeric value (count) via software. The 8253 will wait till the specified count before sending another pulse to the speaker. The count keeps on getting decremented. When it becomes 0 a pulse is sent to the speaker.

The 8253 chip also features a control port (address 67) via which the operational mode (one of the 6 modes) of the chip can be decided. The 8255 chip is connected to 8253 and to the speaker. The 8255 chip can be used to turn on/off the speaker. This can be achieved via port 97 of the 8255 chip.

8253 – Port 67

We can communicate with the timer chip through any of the four I/O ports with addresses 64 through 67. The timer can be configured to mode number 3, channel number 2. This can be done by writing a 8-bit data to port number 67. The value of this 8-bit data is decided as shown in the slide.

The Control byte is a special 8-bit value send to the control port of the 8253 chip. The value sent signifies the configuration as well as operational mode of the chip. The value we are interested in sending to the port turns out to be 182 (10110110). This is because to drive the speaker we have to select values as show below:

(1) Channel 2—speaker is physically connected to channel 2 of 8253.

(2) As the count specified is 16-bit & count port is only 8-bit we chose to write the value 8-bit at a time low byte followed by high byte.

(3) Operational mode 3 is used to generate square wave pulses.

(4) Finally the count to be specified is in binary rather than BCD binary coded decimal.

Playing Tunes

The program given in the slide plays tunes.

Here, the values in **float** array **n** specifies frequency of a note to play. We are using union **p** since we have to send an **int** to **outportb( )** and **outportb( )** can send only a byte at a time. Then we have sent the value 182 (we have already discussed the meaning of bits of this value in earlier slide) to the control port 67 using **outportb( )** function. Next, using the **inportb( )** function we have read the status of speaker from port 97.

To turn on the speaker we have called **outportb( )** function. Using this function we can turn on/off the speaker. Since, we want to turn on the speaker, we must set the lower 2 bits of the value present at port number 97 to 1, without disturbing the other bits. This we have achieved by using bitwise OR operator.

To control the pitch of the speaker we should provide a frequency number to the timer and then turn on the speaker for the duration of the beep. The frequency number is actually a counter value that tells the PC how many of cycles to wait before sending another pulse. A smaller count value will cause the pulses to be sent quicker, resulting in a higher pitch. The count value can be calculated by the following formula:

```
count = 1193280 / n[ i ] ;
```

where, **n[i]** is the frequency of the note that we wish to play.

We have played the notes by calculating the count value for the frequency and sending it to the port 66 through **outportb( )**. This is repeated for all the frequency values of **n[]** through a **for** loop.

Lastly we have turned off the speaker by calling **outportb( )** function where we have sent **status** to port 97.

Windows - I

In this lecture you will understand:

* How Windows is Different
* Advantage of Windows Programming model over DOS model

How Windows Is Different

In this lecture we would explore how C programming is done under Windows. Let us first see some of the changes that have happened under Windows environment.

Under 16-bit environment (DOS) the size of integer is of 2 bytes. As against this, under 32-bit environment an integer is of 4 bytes. Hence its range is -2147483648 to +2147483647. Thus there is no difference between an int and a long int. When we know that the number to be stored in it is hardly going to exceed hundred. In such a case it would be more sensible to use a short int since it is only 2 bytes long.

In data types COLORREF, HANDLE, etc. given in the slide are merely **typedef**'s of the normal integer data type.

A typical C under Windows program would contain several such **typedefs**. There are two reasons why Windows-based C programs heavily make use of **typedefs**. These are:

A typical Windows program is required to perform several complex tasks. For example a program may print documents, send mails, perform file I/O, manage multiple threads of execution, draw in a window, play sound files, perform operations over the network apart from normal data processing tasks. Naturally a program that carries out so many tasks would be very big in size. In such a program if we start using the normal integer data type to represent variables that hold different entities we would soon lose track of what that integer value actually represents. This can be overcome by suitably typedefining the integer as shown in the slide at the top right corner block.

At several places in Windows programming we are required to gather and work with dissimilar but inter-related data. This can be done using a structure. But when we define any structure variable we are required to precede it with the keyword **struct**. This can be avoided by using **typedef** as shown in the slide (at bottom right corner block), such as RECT **r** and PRECT **pr**.

Under 32-bit environment like Windows several programs reside and work in memory at the same time. Hence it is known as a multi-tasking environment. But the moment there are multiple programs running in memory there is a possibility of conflict if two programs simultaneously access the machine resources. This is done by the Microprocessor & OS in the memory management. This is explained in the next paragraph.

Under Windows several applications run in memory simultaneously. The maximum allowable memory—1 MB—that was used in 16-bit environment was just too small for this. Hence Windows had to evolve a new memory management model. This operation is often called page-out operation. Here page stands for a block of memory (usually of size 4096 bytes). When that part of the program that was paged out is needed it is brought back into memory (called page-in operation) and some other programs (or their parts) are paged out. This keeps on happening without a common user's knowledge all the time while working with Windows.

All devices under Windows are shared amongst all the running programs. Hence no program is permitted a direct access to any of the devices. The access to a device is routed through a device driver program, which finally accesses the device. There is a standard way in which an application can communicate with the device driver. It is device driver's responsibility to ensure that multiple requests coming from different applications are handled without causing any conflict.

DOS Programming Model

Typical 16-bit environments like DOS use a sequential programming model. In this model programs are executed from top to bottom in an orderly fashion. In this programming model it is the program and not the operating system that determines which function gets called and when. The operating system simply loads and executes the program and then waits for it to finish. If the program wishes it can take help of the OS to carry out jobs like console I/O, file I/O, printing, etc. For other operations

like generating graphics, carrying out serial communication, etc. the program has to call another set of functions called ROM-BIOS functions.

Unfortunately the DOS functions and the BIOS functions do not have any names. Hence to call them the program had to use a mechanism called interrupts. This is a messy affair since the programmer has to remember interrupt numbers for calling different functions. Moreover, communication with these functions has to be done using CPU registers. This lead to lot of difficulties since different functions use different registers for communication. To an extent these difficulties are reduced by providing library functions that in turn call the DOS/BIOS functions using interrupts. But the library doesn't have a parallel function for every DOS/BIOS function. DOS functions either call BIOS functions or directly access the hardware.

At times the programs are needed to directly interact with the hardware. This has to be done because either there are no DOS/BIOS functions to do this, or if they are there their reach is limited.

There are several limitations in the DOS programming model. These have been listed below:

No True Reuse: The library functions that are called from each program become part of the executable file (.EXE) for that program. Thus the same functions get replicated in several EXE files, thereby wasting precious disk space.

Messy calling mechanism: It is difficult to remember interrupt numbers and the registers that are to be used for communication with DOS/BIOS functions

Hardware dependency: DOS programs are always required to bother about the details of the hardware on which they are running. This is because for every new piece of hardware introduced there are new interrupt numbers and new register details

Inconsistent look & feel: Every DOS program has a different user interface that the user has to get used to before he can start getting work out of the program. For example, successful DOS-based software like Lotus 1-2-3, FoxPro, WordStar offered different types of menus. This happened because DOS/BIOS doesn't provide any functions for creating user interface elements like menus..

All these limitations mentioned above are eliminated under Windows (as shown in the next slide).

Windows Programming Model

Windows programming model is designed with a view to:

Permit true reuse of commonly used functions

A C under Windows program calls several API functions during course of its execution. The API functions are stored in special files that have an extension .DLL (Dynamic Link Libraries). A DLL is a binary file that provides a library of functions. The functions present in DLLs can be linked during execution. These functions can also be shared between several applications running in Windows. Since linking is done dynamically the functions do not become part of the executable file.

Eliminate the messy calling mechanism of DOS

Instead of calling functions using Interrupt numbers and registers Windows provides functions within itself, which can be called using names. These functions are called API (Application Programming Interface) functions.

Provide consistent look and feel for all applications

Each program offers a consistent and similar user interface. As a result, user doesn't have to spend long periods of time mastering a new program. Every program occupies a window—a rectangular area on the screen. A window is identified by a title bar.

Eliminate hardware dependency

A Windows program can always call Windows API functions. Thus an application can easily communicate with OS and OS can also communicate with application. At no time does the application carry out any direct communication with the devices. Any differences that may be there in the new set of mouse and keyboard would be handled by the device driver and not by the application program. So this eliminates H/W dependency.

First Windows Program

Let us now discuss a simple windows program.

As in DOS **main( )** is the entry point **WinMain( )** is the entry in the Windows. It receives four parameters. The first is an unsigned int and the second is a pointer to a char. These macros are defined in 'windows.h'. This header file must always be included while writing a C program under Windows. Here, HINSTANCE and LPSTR are nothing but **typedefs**.

__stdcall is a calling convention that pass arguments to functions from right to left. In case of **__stdcall** the stack is cleaned up by the called function. All API functions use **__stdcall** calling convention. If not mentioned, **__cdecl** calling convention is assumed by the compiler.

The variable h is the 'instance handle' for the running application. Windows creates this ID number when the application starts. A handle is simply a 32-bit number that refers to an entity. The entity could be an application, a window, an icon, a brush, a cursor, a bitmap, a file, a device or any such entity. Then **hp** is a remnant of earlier versions of Windows and is no longer significant. Now it always contains a value 0. It is being persisted with only to ensure backward compatibility. The variable s is a pointer to a character string containing the command line arguments passed to the program. This is similar to the **argv**, **argc** parameters passed to **main( )** in a DOS program. The variable n is an integer value that is passed to the function. This integer tells the program whether the window that it creates should appear minimized, as an icon, normal, or maximized when it is displayed for the first time.

The **MessageBox( )** function receives first parameter as handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window. Second parameter is a pointer to a null-terminated string that contains the message to be displayed. Third parameter is pointer to a null-terminated string that contains the dialog box title. If this parameter is NULL, the default title 'Error' is displayed. The last parameter specifies the contents and behavior of the dialog box.

Returning 0 from **WinMain( )** indicates success, whereas, returning a nonzero value indicates failure. When the program is executed the **MessageBox( )** function pops up a message box whose title is 'Title' and which contains a message 'Hello!', with an 'OK' button on it.

Command Line Arguments

The program given in the slide illustrates how to read command line arguments. The command line arguments can be supplied to the program by executing it from Start | Run as shown in the slide. 'myapp.exe' is the name of our application, whereas, 'abc ijk xyz' represents command line arguments. The parameter **s** points to the string 'abc ijk xyz'.

Windows - II

In this lecture you will understand:

- Creating a window

Window Elements

The slide shows various elements of a window.

Creating A Window

To actually create a window we need to call the API function **CreateWindow( )**. This function requires several parameters including the window class. Windows insists that a window class should be registered with it before we attempt to create windows of that type. Once a window class is registered we can create several windows of that type. Each of these windows would enjoy the same properties that have been registered through the window class. There are several predefined window classes. Some of these are BUTTON, EDIT LISTBOX, etc. Our program has created one such window using the predefined BUTTON class.

The second parameter passed to **CreateWindow( )** indicates the text that is going to appear on the button surface. The third parameter specifies the window style. WS_OVERLAPPEDWINDOW is a commonly used style. The next four parameters specify the window's initial position and size—the x and y screen coordinates of the window's top left corner and the window's width and height in pixels. The next three parameters specify the handles to the parent window, the menu and the application instance respectively. The last parameter is the pointer to the window-creation data.

Note that **CreateWindow( )** merely creates the window in memory. We still are to display it on the screen. This can be done using the **ShowWindow( )** API function. **CreateWindow( )** returns handle of the created window. Our program uses this handle to refer to the window while calling **ShowWindow( )**. The second parameter passed to **ShowWindow( )** signifies whether the window would appear minimized, maximized or normal. If the value of this parameter is **SW_SHOWNORMAL** we get a normal sized window, if it is **SW_SHOWMINIMIZED** we get a minimized window and if it is **SW_SHOWMINIMIZED** we get a maximized window. We have passed **nCmdShow** as the second parameter. This variable contains **SW_SHOWNORMAL** by default. Hence our program displays a normal sized window. The WS_OVERLAPPEDWINDOW style is a collection of the following styles:

```
WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX |
WS_MAXIMIZEBOX
```

On executing this program a window and a message box appears on the screen as shown in the slide. The window and the message box disappear as soon as we click on OK. This is because on doing so execution of **WinMain( )** comes to an end and moreover we have made no provision to interact with the window.

You can try to remove the call to **MessageBox( )** and see the result. You would observe that no sooner does the window appear it disappears. Thus a call to **MessageBox( )** serves the similar purpose as **getch( )** does in sequential programming.

More Windows

In the previous slide we learnt to create a window let us now try to create several windows on the screen. This can be done using **CreateWindow( )** and **ShowWindow( )** function with in the **for** loop. Note that each window created in this program is assigned a different handle. You may experiment a bit by changing the name of the window class to EDIT and see the result.

Windows - III

In this lecture you will understand:

* What is Event Driven Programming Model
* How to create and display a real world window

Interaction – Event Driven Model

When a user interacts with a Windows program a lot of events occur. For each event a message is sent to the program and the program reacts to it. Since the order in which the user would interact with the user-interface elements of the program cannot be predicted the order of occurrence of events, and hence the order of messages, also becomes unpredictable. As a result, the order of calling the functions in the program (that react to different messages) is dictated by the order of occurrence of events. Hence this programming model is called 'Event Driven Programming Model'.

There can be hundreds of ways in which the user may interact with an application. In addition to this some events may occur without any user interaction. For example, events occur when we create a window, when the window's contents are to be drawn, etc. Not only this, occurrence of one event may trigger a few more events. Thus literally hundreds of messages may be sent to an application thereby creating a chaos. Naturally, a question comes—in which order would these messages get processed by the application. Order is brought to this chaos by putting all the messages that reach the application into a 'Queue'. The messages in the queue are processed in First In First Out (FIFO) order. In fact the OS maintains several such queues. There is one queue, which is common for all applications. This queue is known as 'System Message Queue'(SMQ). In addition there is one queue per application. Such queues are called 'Application Message Queues'(AMQ). Let us understand the need for maintaining so many queues.

When we click a mouse and an event occurs the device driver posts a message into the SMQ. The OS retrieves this message finds out with regard to which application the message has been sent. Next it posts a message into the AMQ of the application in which the mouse was clicked.

Each Message has a unique ID (address) called window message (WM_). For example WM_LBUTTONDOWN message is passed when the left mouse button is depressed. Similarly for WM_MOUSEMOVE (when mouse is moving), WM_CREATE (when window is created using the **CreateWindow()** function). All these handlers are #defined in the windows.h file.

A Real World Window

Creating and displaying a window on the screen is a 4-step process. These steps are shown in the slide:

Creation of a window class involves setting up of elements of a structure called **WNDCLASSEX** This structure contains several elements. They govern the properties of the window. Registration of a window class, creation of a window and displaying of a window involves calling of API functions. **RegisterClassEx( )**, **CreateWindow( )** and **ShowWindow( )** respectively.

In **WinMain( )** message is retrieved from the message queue by calling the API function **GetMessage( )**. This would pick the message info from the message queue and place it in the structure variable passed to it. After picking up the message from the message queue we need to process it. This is done by calling the **DispatchMessage( )** API function.

The **DispatchMessage** function dispatches a message to a window procedure **WndProc( )**. It is typically used to dispatch a message retrieved by the **GetMessage( )** function. In **InitInstance( )** while filling the **WNDCLASSEX** structure one of the elements has been set up with the address of a user-defined function called **WndProc( )**.Using this address **DispatchMessage( )** calls the function **WndProc( )**.

Program

Suppose we wish to create a window and draw a few shapes in it. For creating such a window there is no standard window class available. Hence we would have to create our own window class, register it with Windows OS and then create a window on the basis of it.

As expected **WinMain( )** starts off by calling the function **InitInstance( )** present in 'helper.h' file. This file has been **#include**d at the beginning of the program. Remember to copy this file to your project directory—the directory in which you are going to create this program.

Once the window has been created and displayed let us see how we can interact with it. As and when the user interacts with the window—by stretching its boundaries or clicking the buttons in the title bar, etc. a suitable message is posted into the message queue of our application. Our application should now pick them up from the message queue and process them.

A message contains a message ID and some other additional information about the message. Since it is difficult to memorize the message IDs they have been suitably **#define**d in 'windows.h'. The message ID and the additional information are stored in a structure called MSG.

As explained earlier, in **WinMain( )** this MSG message structure is retrieved from the message queue by calling the API function **GetMessage( )**. The first parameter passed to this function is the address of the **MSG** structure variable. **GetMessage( )** would pick the message info. from the message queue and place it in the structure variable passed to it. Don't bother about the other parameters right now.

After picking up the message from the message queue we need to process it. This is done by calling the **DispatchMessage( )** API function.

Since several messages get posted into the message queue picking of the message and processing it should be done repeatedly. Hence calls to **GetMesage( )** and **DispatchMessage( )** have been made in a **while** loop in **WinMain( )**. When **GetMessage( )** encounters a message with ID **WM_QUIT** it returns a **0**. Now the control comes out of the loop and **WinMain( )** comes to an end.

Slide Number 4

As we saw in the previous section, for every message picked up from the message queue the control is transferred to the **WndProc( )** function. This function is shown below:

```
LRESULT CALLBACK WndProc ( HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam )
```

This function always receives four parameters. The first parameter is the handle to the window for which the message has been received. The second parameter is the message ID, whereas, the third and fourth parameters contain additional information about the message.

LRESULT is a **typedef** of a **long int** and represents the return value of this function. **CALLBACK** is a **typedef** of **__stdcall**. This **typedef** has been done in 'windows.h'. **CALLBACK** indicates that the **WndProc** function has been registered with Windows (through **WNDCLASSEX** structure in **InitInstance( )**) with an intention that Windows would call this back (through **DispatchMessage( )** function).

In the **WndProc( )** function we have checked the message ID using a **switch**. If the ID is **WM_DESTROY** then we have called the function **OnDestroy( )**. This message is posted to the message queue when the user clicks on the 'Close Window' button in the title bar. In **OnDestroy( )** function we have called the API function **PostQuitMessage( )**. This function posts a **WM_QUIT** message into the message queue. As we saw earlier, when this message is picked up the message loop and **WinMain( )** is terminated.

For all messages other than **WM_DESTROY** the control lands in the **default** clause of **switch**. Here we have simply made a call to **DefWindowProc( )** API function. This function does the default processing of the message that we have decided not to tackle. The default processing for different message would be different. For example on double clicking the title bar **DefWindowProc( )** maximizes the window.

Actually speaking when we close the window a **WM_CLOSE** message is posted into the message queue. Since we have not handled this message the **DefWindowProc( )** function gets called to tackle this message. The **DefWindowProc( )** function destroys the window and places a **WM_DESTROY** message in the message queue. As discussed earlier, in **WndProc( )** we have made the provision to terminate the application on encountering **WM_DESTROY**.

Working

As explained earlier, the whole process is represented in the form of flow chart. A very clear understanding of it would help you make a good Windows programmer.

Graphics

World has progressed much beyond 16 colors and 640 x 480 resolution graphics that Turbo C/C++ compilers offered under MS-DOS environment. Today we are living in a world of 1024 x 768 resolution offering 16.7 million colors. Graphical menus, icons, colored cursors, bitmaps, wave files and animations are the order of the day.

Drawing In The Window

Drawing in Windows is device independent. Device independence means that the same program should be able to work using different screens, keyboards and printers without modification to the program. Windows takes care of the hardware, allowing the programmer to concentrate on the program itself. The key to this 'device independence' is Windows' use of a 'device context.

Windows programs do not send data directly to the screen or printer. A Windows program knows where (screen/printer) its output is being sent. However, it does not know how it would be sent there, neither does it needs to bother to know this. This is because Windows uses a standard and consistent way to send the output to screen/printer. This standard way uses an entity called Device Context, or simply a DC. Different DC's are associated with different devices. For example, a screen DC is associated with a screen; a printer DC is associated with a printer, etc. Any drawing that we do using the screen DC is directed to the screen. Similarly, any drawing done using the printer DC is directed to the printer. Thus, the only thing that changes from drawing to screen and drawing to printer is the DC that is used.

A windows program obtains a handle (ID value) for the screen or printer's DC. The output data is sent to the screen/printer using its DC, and then Windows and the Device Driver for the device takes care of sending it to the real hardware. The advantage of using the DC is that the graphics and text commands that we send using the DC are always the same, regardless of where the physical output is showing up.

The part of Windows that converts the Windows graphics function calls to the actual commands sent to the hardware is the GDI, or Graphics Device Interface. The GDI is a program file called GDI32.DLL and is stored in the Windows System directory.

The steps involved in creating device context and drawing shapes are shown in the slide.

Hello Windows

We would begin our tryst with graphics programming under windows by displaying a message "Hello Windows" in different fonts. Note that though we are displaying text under Windows even text gets drawn graphically in the window.

The code given in this slide is similar to the earlier one.

Slide Number 9

Drawing to a window involves handling the **WM_PAINT** message. This message is generated whenever the client area of the window needs to be redrawn. This redrawing would be required in the following situations:

(a) When the Window is displayed for the first time.

(b) When the window is minimized and then maximized.

(c) When some portion of the window is overlapped by another window and the overlapped window is dismissed.

(d) When the size of the window changes on stretching its boundaries.

(e) When the window is dragged out of the screen and then brought back into the screen

When the **switch-case** structure inside **WndProc( )** finds that the message ID passed to **WndProc( )** is **WM_PAINT**, it calls the function **OnPaint( )**.

Slide Number 10

In **OnPaint( )** we have called the API function **BeginPaint( )**. This function obtains a handle to the device context. Additionally it also fills the PAINTSTRUCT structure with information about the area of the window, which needs to be repainted. The PAINTSTRUCT structure contains information that can be used to paint the client area of a window (invalid rectangle).

We have setup a **LOGFONT** structure **f**. This structure is used to indicate the font properties like font name, font height, italic or normal, etc. The properties that we have not setup in the loop are all initialized to **0**. Once the font properties have been setup we have called the **CreateFontIndirect( )** API function to create the font. This function loads the relevant font file.

Then using the information in the font file and the font properties setup in the **LOGFONT** structure it creates a font in memory. **CreateFontIndirect( )** returns the handle to the font created in memory. This handle is then passed to the **SelectObject( )** API function to get the font into the DC. This function returns the handle to the existing font in the DC, which is preserved in **holdfont** variable. Next we have used the **SetTextColor( )** API function to set the color of the text to be displayed through **TextOut( )**. The **RGB( )** macro uses the red, green and blue component values to generate a 32-bit color value. Note that each color component can take a value from **0** to **255**. To **TextOut( )** we have to pass the handle to the DC, position where the text is to be displayed, the text to be displayed and its length.

With **hfont** only one font can be associated at a time. Hence before associating another font with it we have deleted the existing font using the **DeleteObject( )** API function. Then we have called the **EndPaint( )** API function to release the DC handle. If not released we would be wasting precious memory, because the device context structure would remain in memory but we would not be able access it.

Windows - IV

In this lecture you will understand:

* What is Device Context
* How to perform Graphics under Window

Drawing Shapes

Let us now discuss a simple program that displays different shapes in a window, as shown in the slide.

The Program

For drawing any shape we need a pen to draw its boundary and a brush to paint the area enclosed by it. The DC contains a default pen and brush. The default pen is a solid pen of black color and the default brush is white in color. In this program we have used the default pen and a blue colored solid brush for drawing the shapes.

As before, we begin by obtaining a handle to the DC using **BeginPaint( )** function. For creating a solid colored brush we need to call the **CreateSolidBrush( )** API function. The second parameter of this function specifies the color of the brush. The function returns the handle of the brush, which we have preserved in the **hbr** variable. Next we have selected this brush in the DC. The handle of the default brush in DC is collected in the **holdbr** variable.

Once we have selected the brush into the DC we are ready to draw the shapes. For drawing the line we have used **MoveToEx( )** and **LineTo( )** API functions. Similarly for drawing a rectangle we have used the **Rectangle( )** function.

The **RoundRect( )** function draws a rectangle with rounded corners. In **RoundRect (x1, y1, x2, y2, x3, y3), x1, y1** represents the *x* and *y*-coordinates of the upper-left corner of the rectangle. Likewise, **x2**, **y2** represent coordinates of the bottom right corner of the rectangle. **x3**, **y3** specify the width and height of the ellipse used to draw the rounded corners(as shown in the slide).

Note that rectangle and the rounded rectangle are drawn from **x1, y1** up to **x2**-**1**, **y2**-**1**. Parameters of **Ellipse( )** specify coordinates of bounding rectangle of the ellipse.

Slide Number 3

The **Pie( )** function draws a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines. The center of the arc is the center of the bounding rectangle specified by **x1, y1** and **x2, y2**. In **Pie(x1, y1, x2, y2, x3, y3, x4, y4), x1, y1** and **x2, y2** specify the *x* and *y*-coordinates of the upper left corner and bottom right corner respectively, of the bounding rectangle. **x3, y3** and **x4, y4** specify the *x* and *y*-coordinates of the arc's starting point and ending point respectively.

In **Polygon (lpPoints, nCount)**, **lpPoints** points to an array of points that specifies the vertices of the polygon. Each point in the array is a **POINT** structure. **nCount** specifies the number of vertices stored in the array. The system closes the polygon automatically, if necessary, by drawing a line from the last vertex to the first.

Once we are through with drawing the shapes the old brush is selected back in the DC and then the brush created by us is deleted using **DeleteObject( )** function and **EndPaint( )** API function to release the DC handle.

Pen Types

In the previous program we have used the default solid black pen of thickness 1 pixel. We can create pens of different style, color and thickness to do our drawing. In the slide, the **OnPaint( )** handler shows how this can be achieved.

A new pen can be created using the **CreatePen( )** API function. This function needs three parameters—pen style, pen thickness and pen color. Different macros like PS_SOLID, PS_DOT, etc.

have been defined in 'windows.h' to represent different pen styles. Note that for pen styles other than PS_SOLID the pen thickness has to be 1 pixel.

Brush Types

The way we can create different types of pens, we can also create three different types of brushes. These are—solid brush, hatch brush and pattern brush. Consider program given in the slide that shows how to build these brushes and then use them to fill rectangles.

In the **OnPaint( )** handler we have drawn three rectangles—first using a solid brush, second using a hatched brush and third using a pattern brush. Creating and using a solid brush and hatched brush is simple. We simply have to make calls to **CreateSolidBrush( )** and **CreateHatchBrush( )** respectively. For the hatch brush we have used the style HS_CROSS There are several other styles defined in 'windows.h' that you can experiment with.

For creating a pattern brush we need to first create a bitmap (pattern). Instead of creating this pattern, we have used a readymade bitmap file present on your hard disk.

When we compile such a program we usually want these resources to become a part of our EXE file. If so done we do not have to ship these resources separately. To be able to use a resource (bitmap file in our case) it is not enough to just copy it in the project directory. Instead we need to carry out the steps

(a) From the 'Insert' menu option of VC++ 6.0 select the Resource' option.

(b) From the dialog that pops up select 'bitmap' followed by the import button.

(c) Select the suitable .bmp file.

(d) From the 'File' menu select the save option to save the generated resource script file (Script1.rc). When we select 'Save' one more file called 'resource.h' also gets created.

(e) Add the 'Script1.rc' file to the project using the Project | Add to Project | Files option.

While using the bitmap in the program it is always referred using an id. The id is #**define**d in the file 'resource.h'. Somewhere information has to be stored linking the id with the actual .bmp file on the disk. This is done in the 'Script1.rc' file. We need to include the 'resource.h' file in the program.

Slide Number 6

To create the pattern brush we first need to load the bitmap in memory. We have done this using the **LoadBitmap( )** API function. The first parameter passed to this function is the handle to the instance of the program. When **InitInstance( )** function is called from **WinMain( )** it stores the instance handle in a global variable **hInst**. We have passed this **hInst** to **LoadBitmap( )**. The second parameter passed to it is a string representing the bitmap. This string is created from the resource id using the **MAKEINTRESOURCE** macro. The **LoadBitmap( )** function returns the handle to the bitmap. This handle is then passed to the **CreatePatternBrush( )** function. This brush is then selected into the DC and then a rectangle is drawn using it.

Note that if the size of the bitmap is bigger than the rectangle being drawn then the bitmap is suitably clipped. On the other hand if the bitmap is smaller than the rectangle it is suitably replicated. While doing the clean up firstly the brush is deleted followed by the bitmap.

Windows - V

In this lecture you will understand:

* How to perform freehand drawing
* How to Capture Mouse
* How to displaying Bitmap
* How to write program that performs animation

Freehand Drawing

PaintBrush provides a facility to draw a freehand drawing-using mouse. We too can achieve this. We can indicate where the freehand drawing begins by clicking the left mouse button. Then as we move the mouse on the table with the left mouse button depressed the freehand drawing should get drawn in the window. This drawing should continue till we do not release the left mouse button.

The mouse input comes in the form of messages. For free hand drawing we need to tackle three mouse messages **WM_LBUTTONDOWN** for left button click, **WM_MOUSEMOVE** for mouse movement and **WM_LBUTTONUP** for releasing the left mouse button.

Slide Number 2

Let us now discuss each mouse handler. When the **WM_LBUTTONDOWN** message arrives the **WndProc( )** function calls the handler **OnLButtonDown( )**. While doing so, we have passed the mouse coordinates where the click occurred. These coordinates are obtained in **lParam** in **WndProc( )**. In **lParam** the low order 16 bits contain the current x - coordinate of the mouse whereas the high order 16 bits contain the y – coordinate as shown in the slide. The **LOWORD** and **HIWORD** macros have been used to separate out these x and y - coordinates from **lParam**.

Slide Number 3

In **OnLButtonDown( )** we have preserved the starting point of freehand in global variables **x1** and **y1**. If in the process of drawing the freehand the mouse cursor goes outside the client area then the window below our window would start getting mouse messages. So our window would not receive any messages. If this has to be avoided then we should ensure that our window continues to receive mouse messages even when the cursor goes out of the client area of our window. The process of doing this is known as mouse capturing.

We have captured the mouse in **OnLButtonDown( )** handler by calling the API function **SetCapture( )**. As a result, the program continues to respond to mouse events during freehand drawing even if the mouse is moved outside the client area.

In the **OnLButtonUp( )** handler we have released the captured mouse by calling the **ReleaseCapture( )** API function.

Slide Number 4

When **OnMouseMove( )** gets called it checks whether the left mouse button stands depressed. If it stands depressed then the **flags** variable contains **MK_LBUTTON**. If it does, then the current mouse coordinates are set up in the global variables **x2**, **y2**. A line is then drawn between **x1**, **y1** and **x2**, **y2** using the functions **MoveToEx( )** and **LineTo( )**. Next time around **x2**, **y2** should become the starting of the next line. Hence the current values of **x2**, **y2** are stored in **x1**, **y1**.

Note that here we have obtained the DC handle using the API function **GetDC( )**. This is because we are carrying out the drawing activity in reaction to a message other than **WM_PAINT**. Also, the handle obtained using **GetDC( )** should be released using a call to **ReleaseDC( )** function.

DC, A Color Look

Now that we have written a few programs and are comfortable with idea of selecting objects like font, pen and brush into the DC, it is time for us to understand how Windows achieves the device independent drawing using the concept of DC. In fact a DC is nothing but a structure that holds handles of various drawing objects like font, pen, brush, etc. A screen DC and its working is shown in the slide

(a) The DC doesn't hold the drawing objects like pen, brush, etc. It merely holds their handles.

(b) With each DC a default monochrome bitmap of size 1 pixel x 1 pixel is associated.

(c) Default objects like black pen, white brush, etc. are shared by different DCs in same or different applications.

(d) The drawing objects that an application explicitly creates can be shared within DCs of the same application, but is never shared between different applications.

(e) Two different applications (App1 and App2) would need two different DCs even though both would be used to draw to the same screen. In other words with one screen multiple DCs can exist.

(f) A common Device Driver would serve the drawing requests coming from different applications. (Truly speaking the request comes from GDI functions that our application calls).

Screen and printer DC is OK, but what purpose would a memory DC serve? Well, that is what the next slide would explain.

Display A Bitmap

We are familiar with drawing normal shapes on screen using a device context. How about drawing images on the screen? Windows does not permit displaying a bitmap image directly using a screen DC. This is because there might be color variations in the screen on which the bitmap was created and the screen on which it is being displayed. To account for such possibilities while displaying a bitmap Windows uses a different mechanism—a 'Memory DC'

The way anything drawn using a screen DC goes to screen, anything drawn using a printer DC goes to a printer, similarly anything drawn using a memory DC goes to memory (RAM). But where in RAM—in the 1 x 1 pixel bitmap whose handle is present in memory DC. (Note that this handle was of little use In case of screen/printer DC). Thus if we attempt to draw a line using a memory DC it would end up on the 1 x 1 pixel bitmap. You would agree 1 x 1 is too small a place to draw even a small line. Hence we need to expand the size and color capability of this bitmap. To do this we have to just replace the handle of the 1 x 1 bitmap with the handle of a bigger and colored bitmap object. This is shown in the slide.

Whatever we draw here would get drawn on the bitmap but would still not be visible. We can make it visible by simply copying the bitmap image (including what has been drawn on it) to the screen DC by using the API function **BitBlt( )**.

Before transferring the image to the screen DC we need to make the memory DC compatible with the screen DC. Here making compatible means making certain adjustments in the contents of the memory DC structure. Looking at these values the screen device driver would suitably adjust the colors when the pixels in the bitmap of memory DC is transferred to screen DC using **BitBlt( )** function.

Program

In **OnPaint( )** we have retrieved the screen DC using the **BeginPaint( )** function. Next we have loaded the vulture bitmap image in memory by calling the **LoadBitmap( )** function. Its usage is similar to what we saw while creating a pattern brush in an earlier program. Then we have created a memory device context and made its properties compatible with that of the screen DC. To do this we have called the API function **CreateCompatibleDC( )**. Note that we have passed the handle to the screen DC to this function. The function in turn returns the handle to the memory DC. After this we have selected the loaded bitmap into the memory DC. Lastly, we have performed a bit block transfer (a bit by bit copy) from memory DC to screen DC using the function **BitBlt( )**. As a result of this the vulture now appears in the window.

We have made the call to **BitBlt( )** as shown below:

```
BitBlt ( hdc, 10, 20, 190, 220, hmemdc, 0, 0, SRCCOPY ) ;
```

Let us now understand its parameters. These are as under:

hdc – Handle to target DC where the bitmap is to be blitted

10, 20 – Position where the bitmap is to be blitted

190, 220 – Width and height of bitmap being blitted

0, 0 – Top left corner of the source image. If we give 10, 20 then the image from 10, 20 to bottom right corner of the bitmap would get blitted.

SRCCOPY – Specifies one of the raster-operation codes. These codes define how the color data for the source rectangle is to be combined with the color data for the destination rectangle to achieve the final color. SRCCOPY means that the pixel color of source should be copied onto the destination pixel of the target.

Animation

Speed is the essence of life. So having the ability to display a bitmap in a window is fine, but if we can add movement and sound to it then nothing like it.

Program

From the **WndProc( )** function you can observe that we have handled two new messages here—**WM_CREATE** and **WM_TIMER**. For these messages we have called the handlers **OnCreate( )** and **OnTimer( )** respectively. This is explained in next slide.

OnCreate( )

The **WM_CREATE** message arrives whenever a new window is created. Since usually a window is created only once, the one-time activity that is to be carried out in a program is usually done in **OnCreate( )** handler. In our program to make the ball move we need to display it at different places at different times. To do this it would be necessary to blit the ball image several times. However, we need to load the image only once. As this is a one-time activity it has been done in the handler function **OnCreate( )**.

You are already familiar with the steps involved in preparing the image for blitting—loading the bitmap, creating a memory DC, making it compatible with screen DC and selecting the bitmap in the memory DC.

Apart from preparing the image for blitting we have also done some intialialisations like setting up values in some variables to indicate the initial position of the ball. We want that every time we run the application the initial position of the ball should be different. To ensure this we have generated its initial x, y coordinates using the standard library function **rand( )**. However, this function doesn't generate true random numbers. To ensure that we do get true random numbers, somehow we need to tie the random number generation with time, as time of each execution of our program would be different. This has been achieved by making the call

```
srand ( time ( NULL ) ) ;
```

Here **time( )** is function that returns the time. We have further passed this time to the **srand( )** function.

To be able to use **rand( )** and **srand( )** functions include the file 'stdlib.h'. Similarly for **time( )** function to work include the file 'time.h'.

We have also called the **SetTimer( )** function. This function tells Windows to post a message **WM_TIMER** into the message queue of our application every 50 milliseconds. This is explained in the next slide.

One application can set up multiple timers to do different jobs at different intervals. Hence we need to pass the id of the timer that we want to set up to the **SetTimer( )** function. In our case we have specified the ID as 1.

For multiple timers Windows would post multiple **WM_TIMER** messages. Each time it would pass the timer ID as additional information about the message.

OnDestroy( )

The **WM_DESTROY** message is sent when a window is being destroyed. It is sent to the window procedure of the window being destroyed after the window is removed from the screen.

A window receives this message through its **WndProc( )** function.

When the application terminates we have to instruct Windows not to send **WM_TIMER** messages to our application any more. For this we have called the **KillTimer( )** API function passing to it the ID of the timer. Deleting the GDI (Graphical Device Interface) object by freeing all system storage associated with it.

We have called the API function **PostQuitMessage( )**. This function posts a **WM_QUIT** message into the message queue. As we saw earlier, when this message is picked up the message loop and **WinMain( )** is terminated.

OnTimer( )

For drawing as well as erasing the ball we have used the same function—**BitBlt( )**. While erasing we have used the raster operation code **WHITENESS**. When we use this code the color values of the source pixels get ignored. Thus red colored pixels of ball would get ignored leading to erasure of the ball in the window.

The size of client area of the window can be obtained using the **GetClientRect( )** API function. For the smooth motion of the ball we have incremented x and y co-ordinates by 10. When the ball touches the edges of window we have called **PlaySound( )** function. This function plays a sound specified by a file name, and the sound is played asynchronously, and returns immediately after beginning the sound. To terminate an asynchronously played waveform sound, in the **PlaySound( )** function second parameter (*pszSound*) is set to NULL.

Slide Number 13

If the ball hits any side of the window, it should appear like, bouncing back. It is achieved by, decrementing the x and y co-ordinates by 10.

Slide Number 14

As a part of exercise develop a program for a game shown in the slide. In this game we have to hit the enemies that are dropped down by the helicopter which after every few seconds appears on the screen at the top-right corner and moves forward towards top-left corner and then disappears once it reaches left edge. While the enemies are coming down near the gun we have to hit them by rotating the gun using arrow keys. If two or more enemies reaches near the gun then the enemies destroy it.

Linux

In this lecture you will understand:

* What is Linux
* C programming under Linux
* What is Processes
* Parent and Child Processes
* Communication using Signals

What Is It

Linux is a clone of the Unix operating system. Its kernel was written from scratch by Linus Torvalds with assistance from a loosely-knit team of programmers across the world on Internet. It has all the features you would expect in a modern OS. Moreover, unlike Windows or Unix, Linux is available completely free of cost. The kernel of Linux is available in source code form. Anybody is free to change it to suit his requirement, with a precondition that the changed kernel can be distributed only in the source code form. Several programs, frameworks, utilities have been built around the Linux kernel. A common user may not want the headaches of downloading the kernel, going through the complicated compilation process, then downloading the frameworks, programs and utilities. Hence many organizations have come forward to make this job easy. They distribute the precompiled kernel, programs, utilities and frameworks on a common media. Moreover, they also provide installation scripts for easy installations of the Linux OS and applications. Some of the popular distributions are RedHat, SUSE, Caldera, Debian, Mandrake, Slackware, etc. Each of them contain the same kernel but may contain different application programs, libraries, frameworks, installation scripts, utilities, etc. Which one is better than the other is only a matter of taste. Linux works on literally every conceivable microprocessor architecture.

Under Linux one is faced with simply too many choices of Linux distributions, graphical shells and managers, editors, compilers, linkers, debuggers, etc. For simplicity we have chosen the combination as shown in the slide.

The First Program

The program is exactly same as compared to a console program under DOS/Windows. It begins with **main( )** and uses **printf( )** standard library function to produce its output. So what is the difference? The difference is in the way programs are typed, compiled and executed. The steps for typing, compiling and executing the program are discussed below.

The first hurdle to cross is the typing of this program. Though any editor can be used to do so, we have preferred to use the editor called 'KWrite'. This is because it is a very simple yet elegant editor compared to other editors like 'vi' or 'emacs'. Note that KWrite is a text editor and is a part of K Desktop environment (KDE). Once KDE is started select the following command from the desktop panel to start KWrite:

K Menu | Accessories | More Accessories | KWrite

After that, carry out the following steps.

(a) Type the program and save it under the name 'hello.c'.

(b) At the command prompt switch to the directory containing 'hello.c' using the **cd** command.

(c) Now compile the program using the **gcc** compiler as shown below:

```
# gcc hello.c
```

(d) On successful compilation **gcc** produces a file named 'a.out'. This file contains the machine code of the program which can now be executed.

(e) Execute the program using following command.

```
# ./a.out
```

Processes

Kernel assigns each process running in memory a unique ID to distinguish it from other running processes. This ID is often known as processes ID or simply PID. It is very simple to print the PID of a running process programmatically.

In the first program, **getpid( )** is a library function which returns the process ID of the calling process. When the execution of the program comes to an end the process stands terminated. Every time we run the program a new process is created. Hence the kernel assigns a new ID to the process each time. This can be verified by executing the program several times—each time it would produce a different output.

As we know, our running program is a process. From this process we can create another process. This is done in the 2nd program. There is a parent-child relationship between the two processes. The way to achieve this is by using a library function called **fork( )**. This function splits the running process into two processes, the existing one is known as parent and the new process is known as child.

Watch the output of the program in the slide. You can notice that all the statements after the **fork( )** are executed twice—once by the parent process and second time by the child process. In other words **fork( )** has managed to split our process into two.

Why fork( )

At times we want our program to perform two jobs simultaneously. Since these jobs may be inter-related we may not want to create two different programs to perform them. Suppose we want to perform two jobs—copy contents of source file to target file and display an animated GIF file indicating that the file copy is in progress. The GIF file should continue to play till file copy is taking place. Once the copying is over the playing of the GIF file should be stopped. Since both these jobs are inter-related they cannot be performed in two different programs. Also, they cannot be performed one after another. Both jobs should be performed simultaneously. We would want to use **fork( )** to create a child process and then write the program in such a manner that file copy is done by the parent and displaying of animated GIF file is done by the child process.

As we know, **fork( )** creates a child process and duplicates the code of the parent process in the child process. There onwards the execution of the **fork( )** function continues in both the processes. Thus the duplication code inside **fork( )** is executed once, whereas the remaining code inside it is executed in both the parent as well as the child process. Hence control would come back from **fork( )** twice, even though it is actually called only once. When control returns from **fork( )** of the parent process it returns the PID of the child process, whereas when control returns from **fork( )** of the child process it always returns a 0. This can be exploited by our program to segregate the code that we want to execute in the parent process from the code that we want to execute in the child process. We have done this in our program using an **if** statement. In the parent process the 'else block' would get executed, whereas in the child process the 'if block' would get executed.

Parent & Child

This program would use the **fork( )** call to create a child process. In the child process we would print the PID of child and its parent, whereas in the parent process we would print the PID of the parent and its child.

In addition to **getpid( )** there is another related function that we have used in this program—**getppid( )**. As the name suggests, this function returns the PID of the parent of the calling process.

You can tally the PIDs from the output and convince yourself that you have understood the **fork( )** function well. A lot of things that follow use the **fork( )** function.

Signals

Consider the program given in the slide. Here we have used an infinite **while** loop to print the message "Program Running" on the screen. When the program is running we can terminate it by pressing the Ctrl + C. When we press Ctrl + C the keyboard device driver informs the Linux kernel about pressing of this special key combination. The kernel reacts to this by sending a signal to our program. Since we have done nothing to handle this signal the default signal handler gets called. In

this default signal handler there is code to terminate the program. Hence on pressing Ctrl + C the program gets terminated.

But how would the default signal handler get called? There are several signals that can be sent to a program. A unique number is associated with each signal. To avoid remembering these numbers, they have been defined as macros like **SIGINT**, **SIGKILL**, **SIGCONT**, etc. in the file 'signal.h'. Every process contains several 'signal ID - function pointer' pairs indicating for which signal which function should be called. If we do not decide to handle a signal then against that signal ID the address of the default signal handler function is present. It is precisely this default signal handler for **SIGINT** that got called when we pressed Ctrl + C when the above program was executed. INT in **SIGINT** stands for interrupt.

Customized Signal Handling

In the program given in the slide, we have registered a signal handler for the SIGINT signal by using the **signal( )** library function. The first parameter of this function specifies the ID of the signal that we wish to register. The second parameter is the address of a function that should get called whenever the signal is received by our program. This address has to be typecasted to a **void *** before passing it to the **signal( )** function.

Now when we press Ctrl + C the registered handler, namely, **sighandler( )** would get called. This function would display the message 'SIGINT received. Inside sighandler' and return the control back to **main( )**. Note that unlike the default handler, our handler does not terminate the execution of our program. So, only way to terminate it is to kill the running process from a different terminal. For this we need to open a new instance of command prompt (terminal). Next do a **ps –a** to obtain the list of processes running at all the command prompts that we have launched. Note down the process id of **a.out**. Finally kill 'a.out' process by saying

kill 3276

Here, **3276** happens to be a process id but it would be different number in your case.

If we wish we can abort the execution of the program in the signal handler itself by using the **exit (0)** beyond the **printf( )**.

Note that signals work asynchronously. That is, when a signal is received no matter what our program is doing, the signal handler would immediately get called. Once the execution of the signal handler is over the execution of the program is resumed from the point where it left off when the signal was received.

Handling Multiple Signals

Let us now try to handle multiple signals. Consider the program given in the slide.

In the program apart from **SIGINT** we have additionally registered two new signals, namely, **SIGTERM** and **SIGCONT**. The **signal( )** function is called thrice to register a different handler for each of the three signals. After registering the signals we enter a infinite **while** loop to print the 'Program running' message on the screen.

As in the previous program, here too, when we press Ctrl + C the handler for the SIGINT i.e. **f1( )** is called. However, when we try to kill the program from the second terminal using the **kill** command the program does not terminate. This is because when the **kill** command is used it sends the running program a **SIGTERM** signal. The default handler for the message terminates the program. Since we have handled this signal ourselves, the handler for **SIGTERM** i.e. **f2( )** gets called. As a result the **printf( )** statement in the **f2( )** function gets executed and the message 'SIGTERM Received' gets displayed on the screen. Once the execution of **f2( )** function is over the program resumes its execution and continues to print 'Program Running'. Then how are we supposed to terminate the program? Simple. Use the following command from another terminal:

kill –SIGKILL 3276

As the command indicates, we are trying to send a **SIGKILL** signal to our program. A **SIGKILL** signal terminates the program.

The process may catch most signals, but there are a few signals that the process cannot catch, and they cause the process to terminate. Such signals are often known as un-catchable signals. The **SIGKILL** signal is an un-catchable signal that forcibly terminates the execution of a process.

Note that even if a process attempts to handle the **SIGKILL** signal by registering a handler for it still the control would always land in the default **SIGKILL** handler which would terminate the program. As explained earlier the process id may be different in each case.

The **SIGKILL** signal is to be used as a last resort to terminate a program that gets out of control. One such process that makes uses of this signal is a system shutdown process. It first sends a **SIGTERM** signal to all processes, waits for a while, thus giving a 'grace period' to all the running processes. However, after the grace period is over it forcibly terminates all the remaining processes using the **SIGKILL** signal.

Common Handler

Consider the program given in the slide. Here, during each call to the **signal( )** function we have specified the address of a common signal handler named **f( )**. Thus the same signal handler function would get called when one of the three signals are received. This does not lead to a problem since in the **f( )** we can figure out inside the signal ID using the first parameter of the function. In our program we have made use of the **switch-case** construct to print a different message for each of the three signals.

Note that we can easily afford to mix the two methods of registering signals in a program. That is, we can register separate signal handlers for some of the signals and a common handler for some other signals. Registering a common handler makes sense if we want to react to different signals in exactly the same way.

Widget Programming

Having understood the mechanism of signal processing let us now see how signaling is used by Linux – based libraries to create event driven GUI programs. As you know, in a GUI program events occur typically when we click on the window, type a character, close the window, repaint the window, etc. We have chosen the GTK library version 2.0 to create the GUI applications. Here, GTK stands for Gimp's Tool Kit.

The GTK library provides a large number of functions that makes it very easy for us to create GUI programs. Every window under GTK is known as a widget.

Let us now see a program to create a simple window. Consider the program given in the slide.

Here, we have Initialized the GTK library with a call to **gtk_init( )** function. This function requires the addresses of the command line arguments received in **main( )**. Next, we have called the **gtk_window_new( )** function to create a top level window. The only parameter this function takes is the type of windows to be created. A top level window can be created by specifying the **GTK_WINDOW_TOPLEVEL** value. This call creates a window in memory and returns a pointer to the widget object. The widget object is a structure (**GtkWidget**) variable that stores lots of information including the attributes of window it represents. We have collected this pointer in a **GtkWidget** structure pointer called **p**. We have set the title for the window by making a call to **gtk_window_set_title( )** function. The first parameter of this function is a pointer to the **GtkWidget** structure representing the window for which the title has to be set. The second parameter is a string describing the text to be displayed in the title of the window.

Then we need to register a signal handler for the destroy signal. The **destroy** signal is received whenever we try to close the window. The handler for the **destroy** signal should perform clean up activities and then shutdown the application. GTK provides a ready-made function called **gtk_main_quit( )** that does this job. We only need to associate this function with the destroy signal. This can be achieved using the **g_signal_connect( )** function. The first parameter of this function is the pointer to the widget for which destroy signal handler has to be registered. The second parameter is a string that specifies the name of the signal. The third parameter is the address of the signal handler routine. We have not used the fourth parameter.

To resize the window to the desired size we have called **gtk_widget_set_size_request( )** function. The second and the third parameters passed to this function specify the height and the width of the window respectively. Lastly, to display the window on the screen we have called function **gtk_widget_show( )**.

In order to wait in a loop to receive events for the window, we have called **gtk_main( )** function.

Slide Number 11

With this knowledge of creating windows let us now try a program that draws a few shapes in the window. Consider the program given in the slide.

This program is similar to the first one. The only difference is that in addition to the destroy signal we have registered a signal handler for the **expose_event** using the **g_signal_connect( )** function. This signal is sent to our process whenever the window needs to be redrawn. By writing the code for drawing shapes in the handler for this signal we are assured that the drawing would never vanish if the windows is dragged outside the screen and then brought back in, or some other window uncovers a portion of our window which was previously overlapped, and so on. This is because a **expose_event** signal would be sent to our application which would immediately redraw the shapes in our window.

The way in Windows we have a device context, under Linux we have a graphics context. In order to draw in the window we need to obtain a graphics context for the window using the **gdk_gc_new( )** function. This function returns a pointer to the graphics context structure. This pointer must be passed to the drawing functions like **gdk_draw_line( )**, **gdk_draw_rectangle( )**, **gdk_draw_arc( )**, **gdk_draw_polygon( )**, etc. Once we are through with drawing we should release the graphics context using the **gdk_gc_unref( )** function.