Chp25.

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer, then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the **cast operator** as follows: `(type_name) expression` Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation: `#include <stdio.h> main() { int sum = 17, count = 5; double mean; mean = (double) sum / count; printf("Value of mean : %f\n", mean ); }` When the above code is compiled and executed, it produces the following result: `Value of mean : 3.400000` It should be noted here that the cast operator has precedence over division, so the value of **sum** is first converted to type **double** and finally it gets divided by count yielding a double value. Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the **cast operator**. It is considered good programming practice to use the cast operator

whenever type conversions are necessary.

Integer Promotion Integer promotion is the process by which values of integer type "smaller" than **int** or **unsigned int** are converted either to **int** or **unsigned int**. Consider an example of adding a character with an integer: `#include <stdio.h> main() { int i = 17; char c = 'c'; /* ascii value is 99 */ int sum; sum = i + c; printf("Value of sum : %d\n", sum ); }` When the above code is compiled and executed, it produces the following result: `Value of sum : 116` Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of 'c' to ASCII before performing the actual addition operation. **Usual Arithmetic Conversion** The **usual arithmetic conversions** are implicitly performed to cast their values to a common type. The compiler first performs *integer promotion*; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy:

The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||. Let us take the following example to understand the concept: `#include <stdio.h> main() { int i = 17; char c = 'c'; /* ascii value is 99 */ float sum; sum = i + c; printf("Value of sum : %f\n", sum ); }`

When the above code is compiled and executed, it produces the following result:

```
Value of sum : 116.000000
```

Here, it is simple to understand that first c gets converted to integer, but as the final value is double, usual arithmetic conversion applies and the compiler converts i and c into 'float' and adds them yielding a 'float' result.

As such, C programming does not provide direct support for error handling but being a sytem programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code **errno**. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in <error.h> header file. So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice to set errno to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program. **errno, perror(), and strerror()** The C programming language provides **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**. • The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value. • The **strerror()** function, which returns a pointer to the textual representation of the current errno value. Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use **stderr** file stream to output all the errors.

```
#include <stdio.h> #include <errno.h> #include <string.h> extern int errno ; int main () { FILE * pf; int errnum; pf = fopen ("unexist.txt", "rb"); if (pf == NULL) { errnum = errno; fprintf(stderr, "Value of errno: %d\n", errno); perror("Error printed by perror"); fprintf(stderr, "Error opening file: %s\n", strerror( errnum )); } else { fclose (pf); } return 0; }
```

When the above code is compiled and executed, it produces the following result:

```
Value of errno: 2 Error printed by perror: No such file or directory Error opening file: No such file or directory
```

Divide by Zero Errors It is a common problem that at the time of dividing any number, programmers do not check if a divisor is zero and finally it creates a runtime error. The code below fixes this by checking if the divisor is zero before dividing:

```
#include <stdio.h> #include <stdlib.h> main() { int dividend = 20; int divisor = 0; int quotient; if( divisor == 0){
```

```
fprintf(stderr, "Division by zero! Exiting...\n"); exit(-1); } quotient = dividend / divisor; fprintf(stderr, "Value of quotient : %d\n", quotient ); exit(0); }
```

When the above code is compiled and executed, it produces the following result:

```
Division by zero! Exiting...
```

Program Exit Status It is a common practice to exit with a value of EXIT_SUCCESS in case of program coming out after a successful operation. Here, EXIT_SUCCESS is a macro and it is defined as 0. If you have an error condition in your program and you are coming out then you should exit with a status EXIT_FAILURE which is defined as -1. So let's write above program as follows: `#include <stdio.h> #include <stdlib.h> main() { int dividend = 20; int divisor = 5; int quotient; if( divisor == 0) { fprintf(stderr, "Division by zero! Exiting...\n"); exit(EXIT_FAILURE); } quotient = dividend / divisor; fprintf(stderr, "Value of quotient : %d\n", quotient );`

`exit(EXIT_SUCCESS); }` When the above code is compiled and executed, it produces the following result: `Value of quotient : 4`

Chp26.

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function. `void recursion() { recursion(); /* function calls itself */ } int main() { recursion(); }` The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc. **Number Factorial** The following example calculates the factorial of a given number using a recursive function: `#include <stdio.h> int factorial(unsigned int i) { if(i <= 1) { return 1; } return i *`

`factorial(i - 1);`

`} int main() { int i = 15; printf("Factorial of %d is %d\n", i, factorial(i)); return 0; }` When the above code is compiled and executed, it produces the following result: `Factorial of 15 is 2004310016` **Fibonacci Series** The following example generates the Fibonacci series for a given number using a recursive function: `#include <stdio.h> int fibonaci(int i) { if(i == 0) { return 0; } if(i == 1) { return 1; } return fibonaci(i-1) + fibonaci(i-2); } int main() { int i; for (i = 0; i < 10; i++) {`

`printf("%d\t%n", fibonaci(i)); } return 0; }` When the above code is compiled and executed, it produces the following result: `0 1 1 2 3 5 8 13 21 34`