FUNCTION OVERLOADING

Function 1:

```
void sum( );
```

Function 2:

```
void sum( int);
```

- When several function declaration are specified for a single function name in the same program , the function name is said to be overloaded.

- C++ allows functions to have the same name, they are distinguished by their number or type of arguments.

Function overloading

- A function name having several definitions that are differentiable by the number or types of their arguments is known as function overloading.

eg : void divide(int); // F1

void divide(int, int); //F2

- Function call

divide(10);

// will invoke F1

divide(12,13);

// will invoke F2

Need for function Overloading

- Needed to reduce the number of comparisons in a program.
- Function overloading implements polymorphism.

- Polymorphism refers to ‘One name having different forms’
- Ability of an object to behave differently in different circumstances.

Declaration and definition

- A function argument list (ie number and type of parameter) is known as function signature.

- The key to function overloading is function signature

- If 2 functions have the same number and type of arguments in the same order, they are said to have the same signature. Even if they are using different variable it will not matter.

```
void square(int a, float b);
void square(int x, float y);
```

C++ allows you to overload the function name

- To overload a function name, all you have to do is declare and define all the functions with the same name but different signatures.
- For eg
- void psqrt(int); // F1
- void psqrt(char c); //F2
- void psqrt(float); //F3
- void psqrt(int, float); //F4

- After declaring overloading function, you must define them separately
- void psqrt(int a)

{ cout << a;}

- void psqrt(char c)

{ c++ ; cout<<c; }

- void psqrt(float x)

{ x=x/10; cout<<x;}

- void psqrt(int x, float y)

{ cout<< x <<“ “<<y;}

Calling overloaded functions

- Write the function call for
- F1
- F2
- F3
- F4

When a function name is declared more than once in the program, the compiler will determine the second declaration as follows:

1. If the signature of the second function matches the first function, then the second one is treated as the re-declaration of the first one.
2. If the signature of the two function match but the return type differ, the second is treated as error.

- For eg:

```
float square( float f);
double square(float f); // error
```

- Functions with the same signature and same name but different return types are not allowed in c++. Different return type is allowed only if argument list is different.

```
float  square( float f);
int  square( double x);
```

- If the signatures of the two functions differ in either the number or type of their arguments, then the two functions are considered to be overloaded.

Restrictions on overloaded functions

1. Any 2 functions in set of overloaded functions must have different argument list.
2. Overloaded functions with same argument list, but different return type is an error
3. typedef declaration do not define new type, they only introduce synonyms for the existing types. They do not affect overloading.

POLYMORPHISM

- Function overloading implements polymorphism in an Object oriented language.
- That is the ability of an object to behave differently in different circumstances , which is implemented through function overloading.
- Function overloading reduces number of comparisons in a program and thereby makes the program run faster.

STEPS INVOVLED IN FINDING THE EXACT MATCH

There are three possible cases, a function call may result in :

1. A match : A match is found in the function call.

2. No match : No match is found for the function call. Match can be found through promotion

ie conversion of char to int, int to float.

3. Ambiguous match : More than one function match

1. Search for an Exact Match

If the type of the actual argument exactly matches the type of one defined instance, the compiler invokes that particular instance. For example,

```
void fun(int);                    //overloaded functions
void fun(double);
fun(0);                           //exactly match. Matches afunc(int)
```

0 (zero) is of type **int** , thus the call exactly matches **fun(int).**

2. A match through promotion

If no exact match is found, an attempt is made to achieve a match through promotion of the actual argument.

Recall that the conversion of integer types **(char, short int)** into **int** (if all values of the type can be represented by **int**) or into **unsigned int** (if all values can't be represented by **int**) is called *integral promotion*.

For example, consider the following code fragment:

```
void fun (int);
void fun (float);
fun ('c');                    //match through the promotion;
                                matches afunc (int)
```

3. A match through application of standard C++ conversion rules

If no exact match or match through a promotion is found, an attempt is made to achieve a match through a standard conversion of the actual argument. Consider the following example,

```
void fun (char);
void  fun(double);
fun(471);                    //match through standard
                                conversion matches afunc
                                (double)
```

The **int** argument 471 can be converted to a **double** value 471 using C++ standard conversion rules and thus the function call matches (through standard conversion) **func(double).**

But if the actual argument may be converted to multiple formal argument types, the compiler wil generate an error message as it will be ambiguous match. For example,

```
void fun (long);
void fun (double);
fun (15);
                        //Error !! Ambiguous match
```

Here the **int** argument 15 can be converted either **long** or **double**, thereby creating an ambiguous situation as to which **afunc()** should be used.

4. A match through application of a user-defined conversion.

If all the above mentioned steps fail, then the compiler will try the user-defined conversion in the combinations to find a unique match.

Any function, whether it is a class member or just an ordinary function can be overloaded in C++, provided it is required to work for distinct argument types, numbers and combinations.

Using default arguments

```
void amount( float price=1000, int time=2, float rate=0.08)
{ cout<< price<<time<<rate;}
```

Consider the following function call cases:

1. amount(2000) // price= ? , time = ?, rate =?
2. amount(2500,3) // price= ? , time = ?, rate =?
3. amount(2300,3,0.11) // price= ? , time = ?, rate =?
4. amount(2300,0.11) // price= ? , time = ?, rate =?
5. amount(5,0.11) // price= ? , time = ?, rate =?

Same program illustrated through Function overloading

```
void amount( float price, int time, float rate); // f1
void amount( float price, int time); // f2
void amount( float price, float rate); // f3
void amount( int time, float rate); // f4
void amount( float price); // f5
```

Consider the following function call cases:

1. amount(2000) // Which Function ?
2. amount(2500,3) // Which Function ?
3. amount(2300,3,0.11) // Which Function ?
4. amount(2300,0.11) // Which Function ?
4. amount(5,0.11) // Which Function ?

Advantages of Function overloading over default arguments

i. Default arguments might not work for all possible combinations of arguments whereas a function may be overloaded for all possible combination of arguments.
ii. With function overloading, multiple function definitions can be executed but with default arguments exactly one function definition is executed.

iii. By declaring an overloaded function, you save the compiler from the trouble of pushing the default argument value on the function call stack .

Programs

1. Using function overloading write the program to find the area of circle , rectangle, triangle.

2. W.A.P that uses a function to check whether a given number is divisible by another number or not. However, if the second number is missing, the function should check whether the given number is prime or not (USE FUNCTION OVERLOADING PROPERTY)

String Functions

The string functions are present in the **string.h** header file. Some string functions are given below:

strlen(S)	It gives the no. of characters including spaces present in a string S.
strcat(S1, S2)	It concatenates the string S2 onto the end of the string S1. The string S1 must have enough locations to hold S2.
strcpy(S1, S2)	It copies character string S2 to string S1. The S1 must have enough storage locations to hold S2.
strcmp((S1, S2)==0) strcmp((S1, S2)>0) strcmp((S1, S2) <0)	It compares S1 and S2 and finds out whether S1 equal to S2, S1 greater than S2 or S1 less than S2.
strcmpi((S1, S2)==0) strcmpi((S1, S2)>0) strcmpi((S1, S2) <0)	It compares S1 and S2 ignoring case and finds out whether S1 equal to S2, S1 greater than S2 or S1 less than S2.
strrev(s)	It converts a string s into its reverse
strupr(s)	It converts a string s into upper case
strlwr(s)	It converts a string s into lower case

Character Functions

All the character functions require **ctype.h** header file. The following table lists the function.

Function	Meaning
isalpha(c)	It returns True if C is an uppercase letter and False if c is lowercase.
isdigit(c)	It returns True if c is a digit (0 through 9) otherwise False.
isalnum(c)	It returns True if c is a digit from 0 through 9 or an alphabetic character (either uppercase or lowercase) otherwise False.
islower(c)	It returns True if C is a lowercase letter otherwise False.
isupper(c)	It returns True if C is an uppercase letter otherwise False.
toupper(c)	It converts c to uppercase letter.
tolower(c)	It converts c to lowercase letter.

General purpose standard library functions

The following are the list of functions are in **stdlib.h**

randomize()	It initializes / seeds the random number generator with a random number
random(n)	It generates a random number between o to n-1
atoi(s)	It converts string s into a numerical representation.
itoa(n)	It converts a number to a string