Introduction

The *polymorphism* refers to 'one name having many forms' 'different behaviour of an instance depending upon the situation'. C++ implements *polymorphism* through *overloaded functions* and *overloaded operators*. The term 'overloading' means a name having two or more distinct meanings. Thus, an 'overloaded function' refers to a function having (one name and) more than one distinct meanings. Similarly, when two or more distinct meanings are defined for an operator, it is said to be an 'overloaded operator'.

Function Overloading

A ***function*** name having several definitions that are differentiable by the number or types of their arguments.

OR

Function Overloading not only implements polymorphism but also reduces number of comparisons in a program and thereby makes the program run faster.

For example;

```
float divide (int a, int b);
float divide (float x, float y);
```

Declaration and Definition

The key to function overloading is a function's argument list which is also known as the function *signature.* It is the *signature,* not the function type that enables function overloading.

Note:

A function's argument list (i.e., number and type of argument) is known as the function's signature.

If two functions are having same number and types of arguments in the same order, they are said to have the same *signature*. Even if they are using distinct variable names, it doesn't matter. For instance, following two functions have same signature.

void squar (int *a*, float *b*); //function 1

void squar (int *x*, float *y*); //same function as that of function 1

To overload a function name, all you need to do is, declare and define all the functions with the same name but different signatures, separately. For instance, following code fragment overloads a function name **prnsqr( ).**

```
Void prnsqr (int i);          //overloaded for integer #1
Void prnsqr (char c);         //overloaded for character #2
Void prnsqr (float f);        //overloaded for floats #3
Void prnsqr (double d);       //overloaded for double floats #4
```

After declaring overloading functions, you must define them separately, as it is shown below for above given declarations.

```
void prnsqr (int i)
{cout<<"Integer"<<i<<"'s square is"<<i*i<<"\n";
}
void prnsqr (char c);
{cout<<c<<"is a character"<<"Thus No Square for
   it"<<"\n";
}
Void prnsqr (float f)
{cout<<"float"<<f<<"'s square is"<<f*f<<"\n";
}
void prnsqr (double d)
{cout <<"Double float"<<d<<"'s square
   is"<<d*d<<"\n';
}
```

Thus, we see that is not too much difficulty in declaring overloaded functions; they are declared as other functions are. Just one thing is to be kept in mind that the arguments are sufficiently different to allow the functions to be differentiated in use.

The argument types are said to be part of function's extended name. For instance, the name of above specified functions might be **`prnsqr()`** but their extended names are different. That is they have **prnsqr(int), prnsqr(char), prnsqr(float), and prnsqr(double)** extended names respectively.

When a function name is declared more than once in a program, the compiler will interpret the second (and subsequent) declaration(s) as follows:

1) If the signatures of subsequent functions match the previous function's, then the second is treated as a re-declaration of the first.
2) If the signatures of two functions match exactly but the return type differ, the second declaration is treated as an erroneous re-declaration of the first and is flagged at compile time as an error.

For example,

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

Functions with the same signature and same name but different return types are not allowed in C++. You can have different return types, but only if the signatures are also different:

```
float square (float f);          //different signatures, hence
double square (double d);        //allowed
```

3) If the signature of the two functions differ in either the number or type of their arguments, the two functions are considered to be overloaded.

Use function overloading only when a function is required to work for alternative argument types and there is a definite way of optimizing the function for the argument type.

Restrictions on Overloaded Functions

Several restrictions governs an acceptable set of overloaded functions:

- Any two functions in a set of overloaded functions must have different argument lists.
- Overloading functions with argument lists of the same types, based on return type alone, is an error.
- Member functions cannot be overloaded solely on the basis of one being static and the other nonstatic.

- Typedef declaration do not define new types; they introduces synonyms for existing types. They do not affect the overloading mechanism. Consider the following code:

```
typedef char* PSTR;
void Print (char * szToPrint);
void Print (PSTR szToPrint);
```

CALLING OVERLOADED FUNCTIONS

Overloaded functions are called just like other functions. The number and type of arguments determine which function should be invoked.

For instance consider the following code fragment:

```
prnsqr ('z');
prnsqr (13);
prnsqr (134.520000012);
prnsqr (12.5F);
```

Steps Involved in Finding the Best Match

A call to an overloaded function is resolved to a particular instance of the function through a process known as *argument matching,* which can be termed as a *process of disambiguation.* Argument matching involves comparing the actual arguments of the call with the formal arguments of each declared instance of the function. There are three possible cases, a function call may result in:

a) **A match.** A match is found for the function call.

b) **No match.** No match is found for the function call.

c) **Ambiguous Match.** **More than one defined instance for the function call.**

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

0 (zero) is of type **int** , thus the call exactly matches **afunc(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, enumerator, 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 afunc (int);
void afunc (float);
afunc ('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 afunc (char);
void afunc (double);
afunc (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 afunc (long);
void afunc (double);
afunc(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.

Default Arguments Versus Overloading

Using default argument gives the appearance of overloading, because the function may be called with an optional number of arguments. For instance, consider the following function prototype:

```
float amount (float principal, int time=2, float rate=0.08);
```

Now this function may be called by providing just one or two or all three argument values. A function call like as follows:

```
cout<<amount (3000);
```

will invoke the function **amount()** with argument values 3000, 2, and 0.08 respectively. Similarly a function call like

```
cout <<amount (3000, 4);
```

Will invoke **amount()** with argument values 2500, 5, and 0.12 respectively. That is if argument values are provided with the function call, then the function is invoked with the given values. But if any value is missing and there has been default values specified for it, then the function is invoked using the default value.

However if you skip the middle argument **time** but C++ makes no attempt at this type of interpretation. C++ will take 0.13 to be the argument value for **time** and hence invoke **amount()** with values 2000, 0 (0.13 converted to **int,** thus 0) and 0.08 (the default rate). That is, with default arguments C++ expects that only the arguments on the right side can be defaulted. If you want to default a middle argument, then all the arguments on its right must also be defaulted.