Object-oriented Programming

Function Overloading

Function Overloading

- C++ enables several functions of the same name to be defined, as long as they have different parameter signatures

- The compiler selects the proper function to call by examining the number, types and order of the arguments in the call

Function Overloading

- Function overloading allows the program to follow the principle of *“Polymorphism”*
- The *return type* or *name* of parameters don’t help the compiler in selecting which overloaded function to call
- Overloading the functions that perform closely related tasks can make programs more readable and understandable

Example

void findPerson(string name) { . . . }

void findPerson(int ID) { . . . }

void findPerson(int ID, string addr) { . . . }

void findPerson(string addr, int ID) { . . . }

All of above are valid overloaded functions

Example

```
void showVal(int a)
{  cout << a;  }

void showVal(char a)
{  cout << a;  }

void showVal(float a)
{  cout << a;  }
```

```
int main()
{
    showVal('M');
    showVal(20);
    showVal(3.5);
}
```

Pitfall

- Avoid using default arguments with overloaded functions as it can be dangerous i.e. can lead to error conditions that are hard to trace

For example:

- In a program, a function that takes no argument and another function of the same name that takes all default arguments

Name Mangling

- The compiler encodes each function identifier with the number and types of its parameters

- This is called **name mangling** or **name decoration**

Name Mangling

- Each mangled name (other than main) begins with two underscores (__) followed by the letter Z, a number and the function name
- The number that follows Z specifies how many characters are in the function's name
- The function name is then followed by an encoding of its parameter list

Name Mangling

```
void func(int i, char c) { . . . }
int func2(string x, float y) { . . . }
void func3(int a, float b, int& c) { . . . }
```

Mangled Names:

```
__Z2funcic
__Z2func2sf
__Z3func3ifRi
```