Overloads and templates

Overloaded functions

- In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. For example:

Example

```
int square (int);
double square (double);
int main() {
        int n;
        cin>>n;
        cout<<square(n)<<endl;
        double m;
        cin>>m;
        cout<<square(m)<<endl;
        return 0;
}

int square (int x){
        return x*x;
}
double square (double x){
        return x*x;
}
```

- In the above code, the square is overloaded with different parameters.
- The function square can be overloaded with other arguments too, which requires the same name and different arguments every time.
- To reduce these efforts, C++ has introduced a generic type called function template
- We can not overload a function just by only its return type

Function templates

- Defining a function template follows the same syntax as a regular function, except that it is preceded by the template keyword and a series of template parameters enclosed in angle-brackets <>:
  - template <template-parameters> function-declaration
- The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the class or typename keyword followed by an identifier. This identifier can then be used in the function declaration as if it was a regular type. For example, a generic sum function could be defined as:

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

- It makes no difference whether the generic type is specified with keyword class or keyword typename in the template argument list (they are 100% synonyms in template declarations).
- In the code above, declaring SomeType (a generic type within the template parameters enclosed in angle-brackets) allows SomeType to be used anywhere in the function definition, just as any other type; it can be used as the type for parameters, as return type, or to declare new variables of this type.
- In all cases, it represents a generic type that will be determined on the moment the template is instantiated.
- Instantiating a template is applying the template to create a function using particular types or values for its template parameters.
- This is done by calling the function template, with the same syntax as calling a regular function, but specifying the template arguments enclosed in angle brackets:
  - name <template-arguments> (function-arguments)
- For example, the sum function template defined above can be called with:
  - x = sum<int>(10,20);

Example

```
template <class T>
T square(T x){
        T result;
        result=x*x;
        return result;
}

int main() {
        int n;
        cin>>n;
        cout<<square<int>(n)<<endl;
        double m;
        cin>>m;
        cout<<square<double>(m)<<endl;
        return 0;
}
```