Function Templates

- Overloaded functions are normally used to perform similar operations that involve different program logic on different data types.
- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates.
- A single function template definition is written. Given the argument types provided in calls to the function, C++ automatically generates separate function template specializations to handle each type of call appropriately
- Thus, defining a single function template essentially defines a whole family of overloaded functions!
- Here is a maximum function template that determines the largest of three values.

```
template <typename T>            // or template<class T>
T maximum(T valuel, T value2, T value3)
{
    T maximumValue = valuel;  // assume valuel is maximum

    // determine whether value2 is greater than maximumValue
    if (value2 > maximumValue)
    {
        maximumValue = value2;
    }

    // determine whether value3 is greater than maximumValue
    if (value3 > maximumValue)
    {
        maximumValue = value3;
    }

    return maximumValue;
}
```

- All function template definitions begin with the template keyword followed by a template parameter list enclosed in angle brackets(< >).
- Every parameter in the template parameter list is preceded by keyword `typename` or keyword `class` (they are synonyms in this context).
- The type parameters are placeholders for fundamental types. These placeholders, in this case, `T`, are used to specify the types of the function's parameters, to specify the function 's return type, and to declare variables within the body of the function definition.
- A function template is defined like any other function, but uses the type parameters as *placeholders* for actual data types.
- The function template declares a single type parameter `T` as a placeholder for the type of the data to be tested by function `maximum`.
- The name of a type parameter must be *unique* in the template parameter list for a particular template definition. When the compiler detects a `maximum()` invocation in the program source code, the *type* of the arguments in the `maximum()` call is substituted for T throughout the template definition, and C++ creates a complete function for determining the maximum of three values of the specified type—all three must have the same type, since we use only one

type parameter in this example. Then the newly created function is compiled—templates are a means of *code generation*.

- Here is an example of using the `maximum` function template to determine the largest of three `int` values, three `double` values, and three `char` values, respectively.

```
int main()
{
    // demonstrate maximum() int values
    cout << "Input three integer values: ";

    int int1, int2, int3;
    cin >> int1 >>  int2 >> int3;

    // invoke int version of maximum
    cout << "The maximum integer value is: " <<
        maximum(int1, int2, int3) << endl;
    cout << endl;

    // demonstrate maximum() double values
    cout << "Input three double values: ";

    double double1, double2, double3;
    cin >> double1 >>  double2 >> double3;

    // invoke int version of maximum
    cout << "The maximum double value is: " <<
        maximum(double1, double2, double3) << endl;
    cout << endl;

    // demonstrate maximum() char values
    cout << "Input three characters: ";

    char char1, char2, char3;
    cin >> char1 >>  char2 >> char3;

    // invoke char version of maximum
    cout << "The maximum char value is: " <<
        maximum(char1, char2, char3) << endl;

    return 0;
}
```

- Because each call uses arguments of a different type, the compiler creates a separate function definition for each—one expecting three `int` values, one expecting three `double` values an one expecting three `char` values, respectively.

- Templates may have more than one template parameter, in which case you just separate the template parameters using commas. Here is a simple `printTwoTypes()` template that uses two template parameters in its template list.

```
// simple example template using two types
template <typename T1, typename T2>
void printTwoTypes(T1 t1, T2 t2)
{
    cout << "t1: " << t1 << endl;
    cout << "t2: " << t2 << endl;
}

int main()
{
    int i = 1;
    char ch = 'a';

    // try two type template
    printTwoTypes(i, ch);

    return 0;
}
```