Advanced Programming Concepts with C++

Templates and **Generic Programming,** and some other cool features!

Fall 2021
CSI2372[A]

Amir EAMAN
Postdoc Researcher
Faculty of Engineering
uOttawa

Templates. → can make returntype.
function parameter
by define by user

O.O Programming VS. Generic Programming

- In generic programming types become known **during compilation**.
- In **O.O programming** types become known **during run time**.

- Generic programming is a style of computer programming.
- In generic programming, we write codes with ***to-be-specified-later*** **types** which will be **instantiated** (as required) when we will provide them specific types as extra information through parameters.

- Templates are the basis of generic programming in C++.
- The Standard Template Library (**STL**) is a good example of generic programming.

Templates

In function
In class

- Templates are the foundation of **generic programming** in C++.
- A generic class or a generic function transform into a specific class or function during compilation. We have to provide the information needed to do the transformation.

- A template is the structure for creating specific classes or functions.
- All the types which we use in templates must be known when we write the program.
- The compiler will generates specific instances of templates when we instantiate the templates by providing template arguments.

- The Standard Library is made up of templates.

Templates - Example

```cpp
#include <iostream>

template <typename T>
T sum( T a,  T b)
{
    return (a + b);
}

int main()
{
    int res = sum<int>(3, 4);
    double res_double = sum<double>(1.0, 3.0);
    std::string res_string = sum<std::string>("one", "two");
    int res2 = sum(8, 5); //the compiler find out the T is int

    std::cout << res << std::endl;
    std::cout << res_double << std::endl;
    std::cout << res_string << std::endl;
    std::cout << res2 << std::endl;

    return 0;
}
```

can be anything

```
7
4
onetwo
13
```

Templates - instantiation

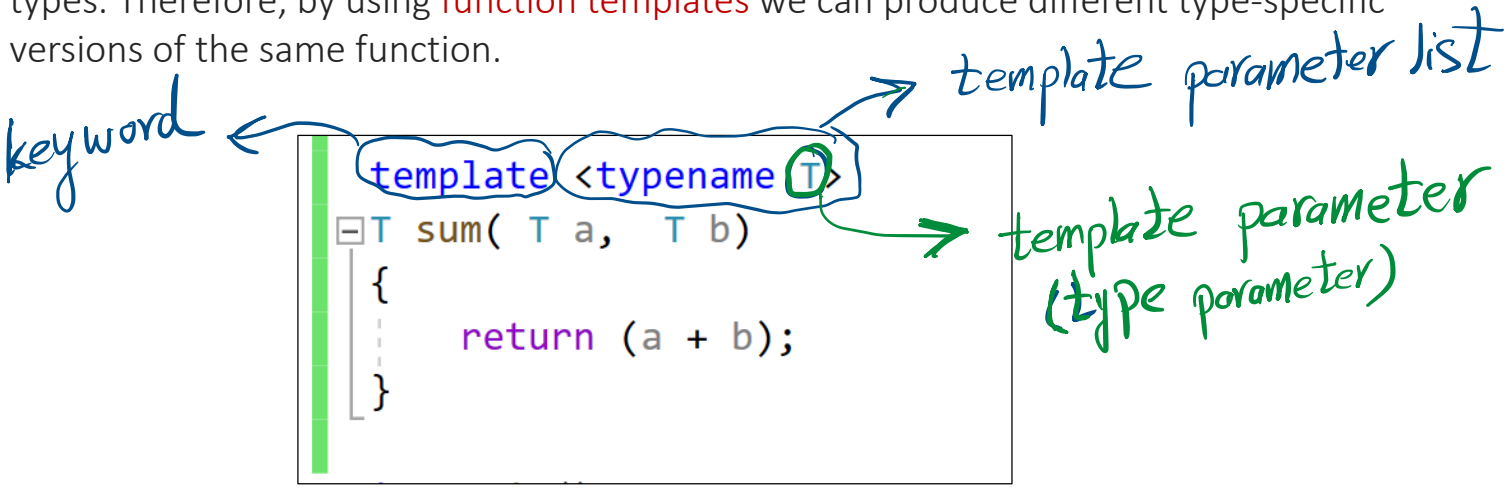
Compiler will generate *type-specific version* of the templates for us. Producing type-specific versions of templates are referred to as instantiation of templates.

Example- A specific version of sum (from previous slide) with T replaced by int

```
int sum(int a, int b)
{
    return(a + b);
}
```

Function Templates

- Function templates help us to avoid repeating the body of the functions for every possible types. Therefore, by using function templates we can produce different type-specific versions of the same function.



Function Templates- template parameter list

- Template parameter list is a comma-separated list of one or more template parameters.
- Template parameters are similar to functions parameters. We must provide template arguments either implicitly or explicitly for the template parameters.

can be define list of type.

```
double res_double = sum<double>(1.0, 3.0);
```

template argument

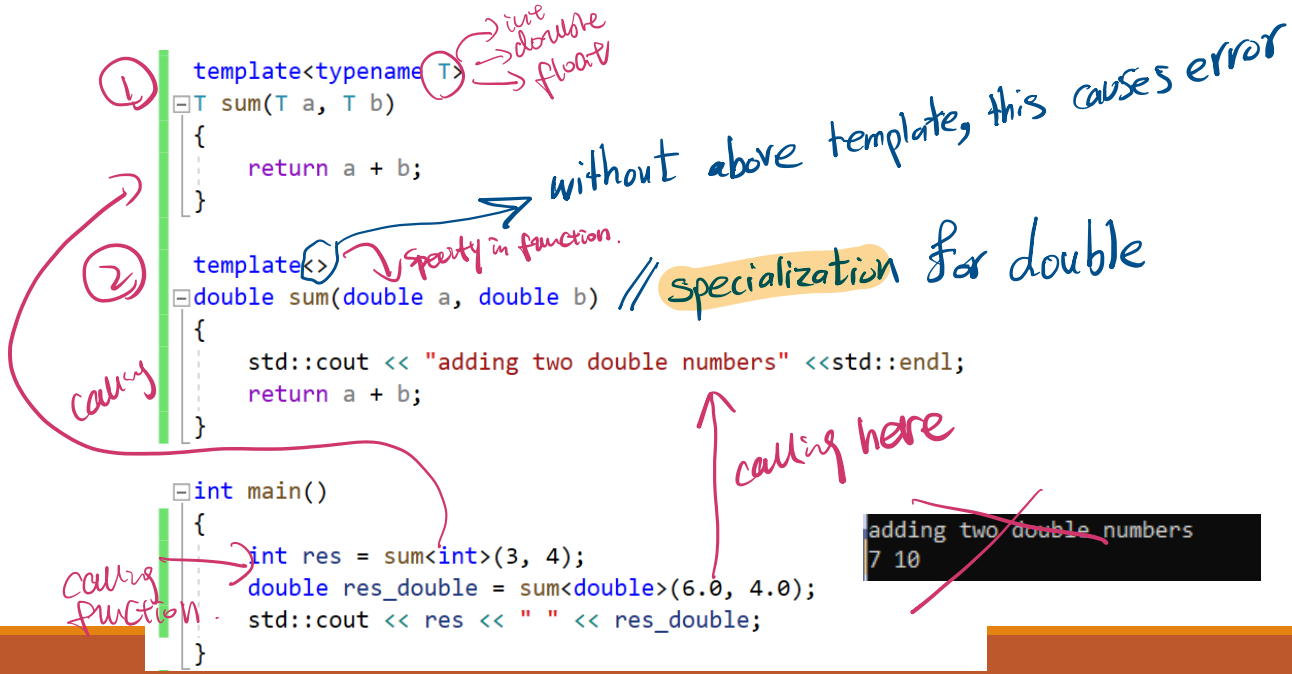
// in the above statement, double will bind to the T template parameter

```
int res2 = sum(8, 5);    // template argument is int and the compiler deduces this type
```

sum<int>(8,5)

Function Templates- specialization of function templates

In function templates, we can define a different implementation for a specific data type:



Templates Parameters – more details

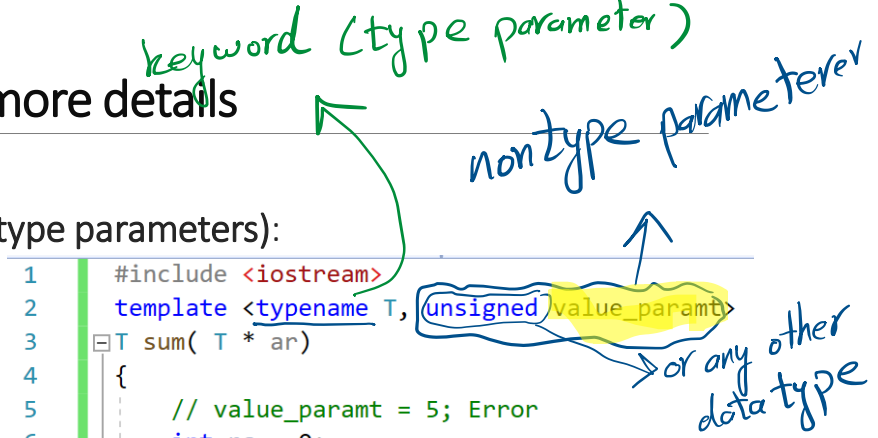
1. Template Type Parameters:

```
1   #include <iostream>
2
3   template <typename T, typename U>
4   T sum( T a,  U b)
5   {
6       return (a + b);
7   }
8
9   int main()
10  {
11      int res = sum<int,double>(3, 4.5);
12
13      std::cout << res << std::endl;
14
15      return 0;
16  }
```

Templates Parameters – more details

2- nontype Template Parameters (nontype parameters):

Nontype parameters are ***constant values*** rather than types.



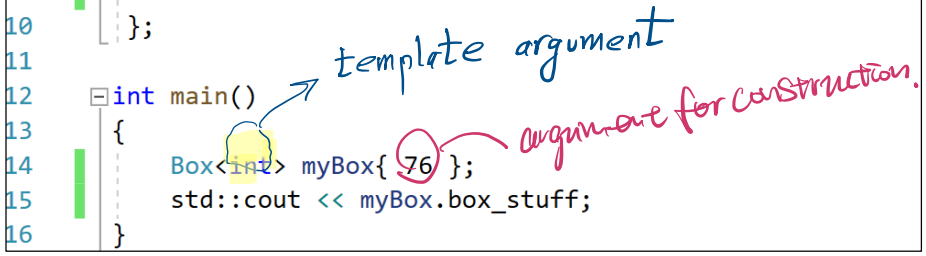

```
1  #include <iostream>
2  template <typename T, unsigned value_paramt>
3  T sum( T * ar)
4  {
5      // value_paramt = 5; Error
6      int rs = 0;
7      for (size_t i = 0; i < value_paramt; i++)
8      {
9          rs += ar[i];
10     }
11     return rs;
12 }
13 int main()
14 {
15     int arr[4] = {1,2,3,4};
16     int res = sum<int,4>(arr);
17     std::cout << res;
18 }
```

Class Templates

- Class templates are formulas to generate type-specific classes.
- Class templates are similar to **function templates**. however, the **template parameters** of class templates **cannot** be deduced, and so we must provide template arguments as additional information (in contrast to **templates parameters** of function templates which compiler **can** deduce).

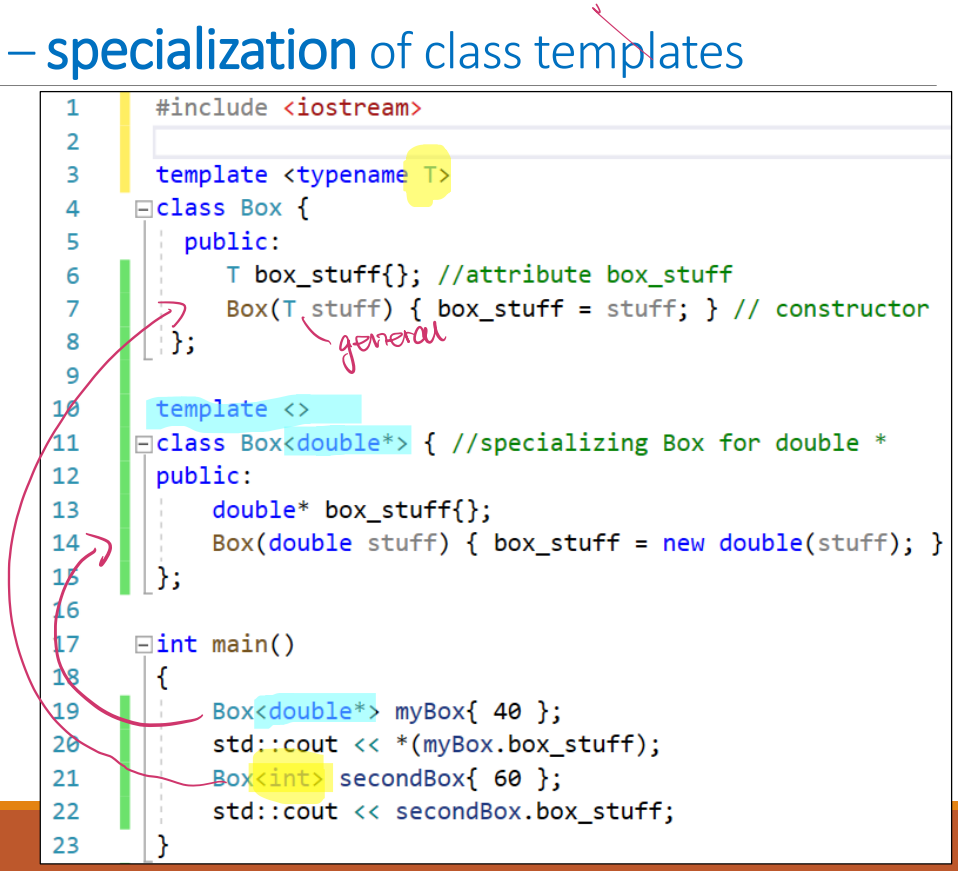
Class Templates - Example

```
1   #include <iostream>
2
3   template <typename T>
4   class Box {
5     public:
6       T box_stuff{}; //attribute box_stuff
7
8       Box(T stuff) { box_stuff = stuff; } // constructor
9
10  };
11
12  int main()
13  {
14      Box<int> myBox{ 76 };
15      std::cout << myBox.box_stuff;
16  }
```



Class Templates – specialization of class templates

In class templates, we can define a different implementation for a specific data type:




```
1  #include <iostream>
2
3  template <typename T>
4  class Box {
5    public:
6      T box_stuff{}; //attribute box_stuff
7      Box(T stuff) { box_stuff = stuff; } // constructor
8  };
9
10 template <>
11 class Box<double*> { //specializing Box for double *
12 public:
13     double* box_stuff{};
14     Box(double stuff) { box_stuff = new double(stuff); }
15 };
16
17 int main()
18 {
19     Box<double*> myBox{ 40 };
20     std::cout << *(myBox.box_stuff);
21     Box<int> secondBox{ 60 };
22     std::cout << secondBox.box_stuff;
23 }
```

some other C++ subjects

Conversion Constructor – the compiler uses them for implicit conversion!

- Constructors with one parameter implicitly converts the input arguments to the class of the constructor.
- The compiler uses them as conversion functions.

```
1   class Simple {
2       int part1;
3       int part2;
4        public:
5           Simple(int a) { part1 = a; part2 = a; }
6           ~Simple() {}
7
8   };
9
10  int main()
11  {
12      Simple s = 6;
13  }
```

constructor . int → simple.

Name	Value
s	{part1=6 part2=6 }
part1	6
part2	6

sign a privilege type to object.
there have hidden constructor

explicit keyword – stop conversion using conversion constructor!

explicit keyword prevents the compiler from using the conversion constructor for implicit conversions

```
1   class Simple {
2       int part1;
3       int part2;
4        public:
5           explicit Simple(int a) { part1 = a; part2 = a; }
6           ~Simple() {}
7
8   };
9
10  int main()
11  {
12      Simple s = 6; //ERROR
13      Simple simple = Simple(6);
14  }
```

stop to use broken constructor to Convertion.

Code	Description
C2440	'initializing': cannot convert from 'int' to 'Simple'
E0415	no suitable constructor exists to convert from "int" to "Simple"

→ can't convert number to object

Friend classes and Friend methods -Example

For a class:

- A friend method can access the private and protected members of the class in which it is declared as a friend. The friend method of a class is not considered as the member function of the class.

- All of the members of a friend class can access the private and protected members of the class in which it is declared as a friend.

Friend classes and Friend methods - example

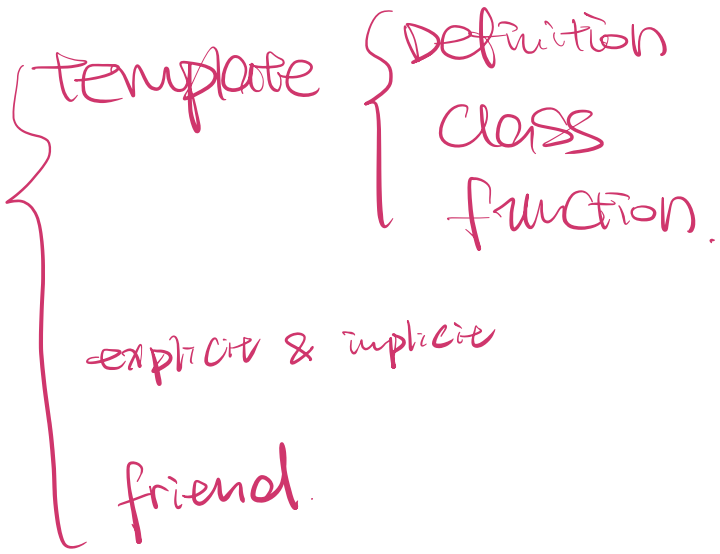
```
1   #include <iostream>
2
3   class Simple {
4       int part1;
5       int part2;
6   public:
7       Simple() :part1(1), part2(1) { };
8       Simple(int a) { part1 = a; part2 = a; }
9       ~Simple() {}
10      friend class VerySimple;
11      friend int myFriendMethod(Simple);
12  };
13
```

private

can access to Simple class variable

Friend classes and Friend methods - example

```
14  class VerySimple {
15      Simple se;
16  public:
17      VerySimple() {}
18      void VerySimpleMethod(int a) { se.part1 = a; se.part2 = a; }
19  };
20
21  int myFriendMethod(Simple se)
22  {
23      return se.part1;
24  }
25
26  int main()
27  {
28      Simple si = 9;
29      std::cout << myFriendMethod(si) << std::endl;
30      VerySimple vs;
31
32  }
```


template
definition
class
function.
explicit & implicit
friend.