*Hochiminh City University of Technology*
*Computer Science and Engineering*
*[CO1011 - 501127] - Fundamentals of C++ Programming*

# Polymorphism & Template

Lecturer: Dr. Duc Dung Nguyen
Dr. Rang Nguyen

Credits: 4

## Outcomes

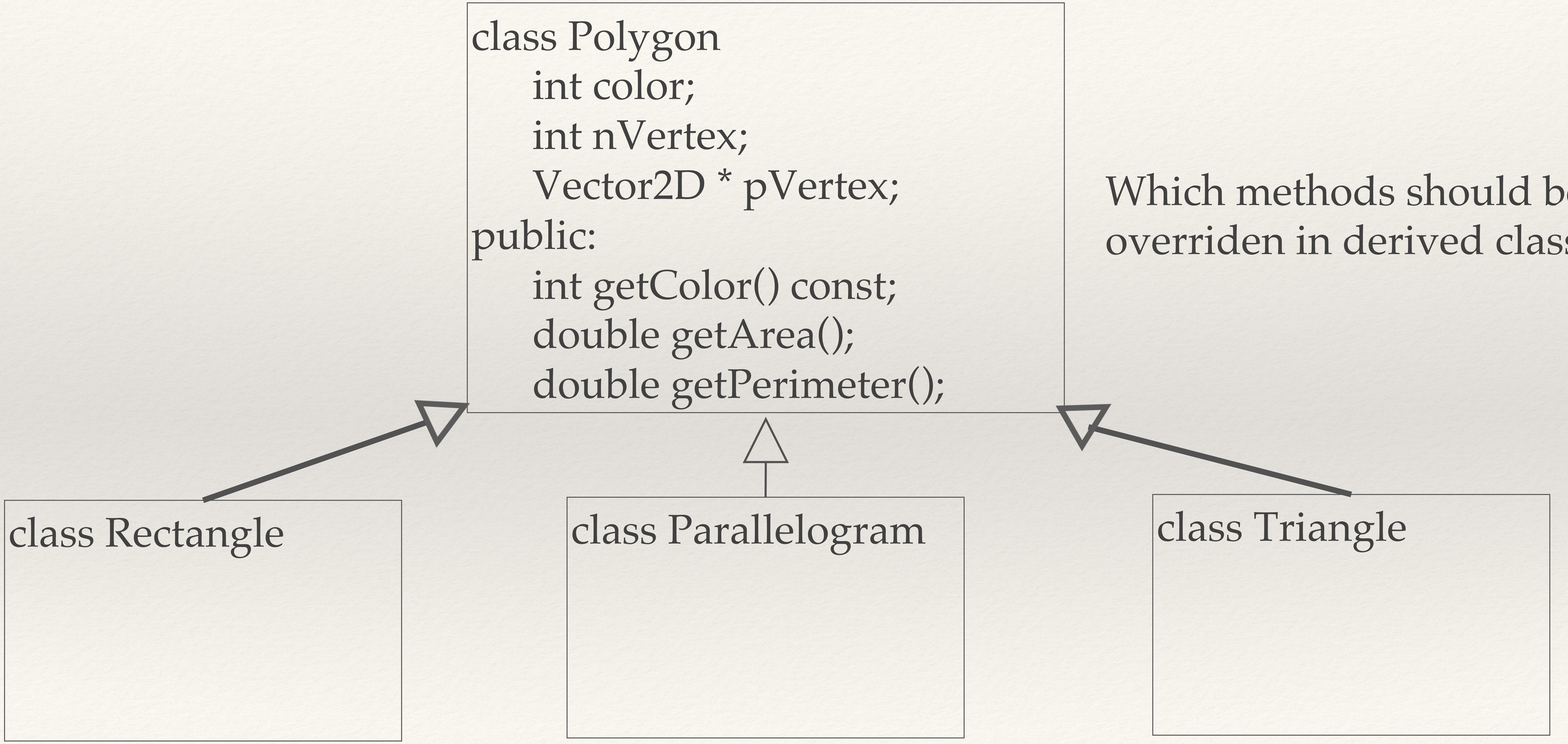
- Explain the concept of polymorphism in OOP.
- Explain the concept of template in C++.

## Outline

- Polymorphism
  - Motivation
  - Polymorphism
  - Dynamic Binding
- Templates
  - Function Templates
  - Class Templates
  - Class Templates and Non-type Parameters
  - Some other issues

## Polymorphism

# Hand-on Exercise



```
class Polygon
    int color;
    int nVertex;
    Vector2D * pVertex;
public:
    int getColor() const;
    double getArea();
    double getPerimeter();
```

```
class Rectangle
```

```
class Parallelogram
```

```
class Triangle
```

Which methods should be overriden in derived classes?

## Hand-on Exercise

Polygon *p = new Rectangle(…);

cout << p->getArea();

- which getArea() method will be called?

## Motivation

- How can p->getArea() call the simple version getArea() of Rectangle?

⇒ Switch the target to the corresponding method of the object p really points to

⇒ Easy extend your program with new classes

## Virtual Functions

- **`virtual`** functions
  - – Declaration:
    - Keyword **`virtual`** before method prototype in base class

```
virtual int getArea() const;
```

  - – <pointer of base class> -> <virtual method> will call to <virtual method> of the object <pointer of base class> points to

    **`p -> getArea()`** will call **`getArea()`** of object p points to
  - – If a derived class does not define a **`virtual`** function, it is inherited from the base class.

## Virtual Functions (cont)

- `PolygonPtr->getArea();`
  - –Compiler implements dynamic binding (late binding)
  - –Target method is determined during execution time

- `PolygonObject.getArea();`
  - –Compiler implements static binding (early binding)
  - –Target method is determined during compile-time

## Abstract and Concrete Classes

- Abstract classes
  - Sole purpose is to provide a base class for other classes
  - No objects of an abstract base class can be instantiated
    - Too generic to define real objects, i.e. **`TwoDimensionalShape`**
    - Can have pointers and references
  - Concrete classes - classes that can instantiate objects
    - Provide specifics to make real objects , i.e. **`Square`**, **`Circle`**

## Abstract and Concrete Classes (cont)

- Making abstract classes
  - Declare one or more **`virtual`** functions as “pure” by initializing the function to zero

```
virtual double earnings() const = 0;
```

    - Pure **`virtual`** function

# Polymorphism

- Polymorphism:
  - Ability for objects of different classes to respond differently to the same function call
  - Base-class pointer (or reference) calls a **`virtual`** function
    - C++ chooses the correct overridden function in object
  - Suppose **`print`** not a **`virtual`** function

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print();   //call base-class print function
hPtr->print();   //call derived-class print function
ePtr=&h;         //allowable implicit conversion
ePtr->print();   // still calls base-class print
```

# New Classes and Dynamic Binding

- Dynamic binding (late binding )
  - – Object's type not needed when compiling **`virtual`** functions
  - – Accommodate new classes that have been added after compilation
  - – Important for ISV’s (Independent Software Vendors) who do not wish to reveal source code to their customers

## Virtual Destructors

- Problem:
  - If base-class pointer to a derived object is **`delete`**d, the base-class destructor will act on the object
- Solution:
  - Declare a **`virtual`** base-class destructor
  - Now, the appropriate destructor will be called

## Hand-on Exercise

- Write class Polygon as an abstract class and classes Triangle and Rectangle as its concrete derived classed. Implement getPerimeter() and getArea() in these classes.

## Template

## Introduction

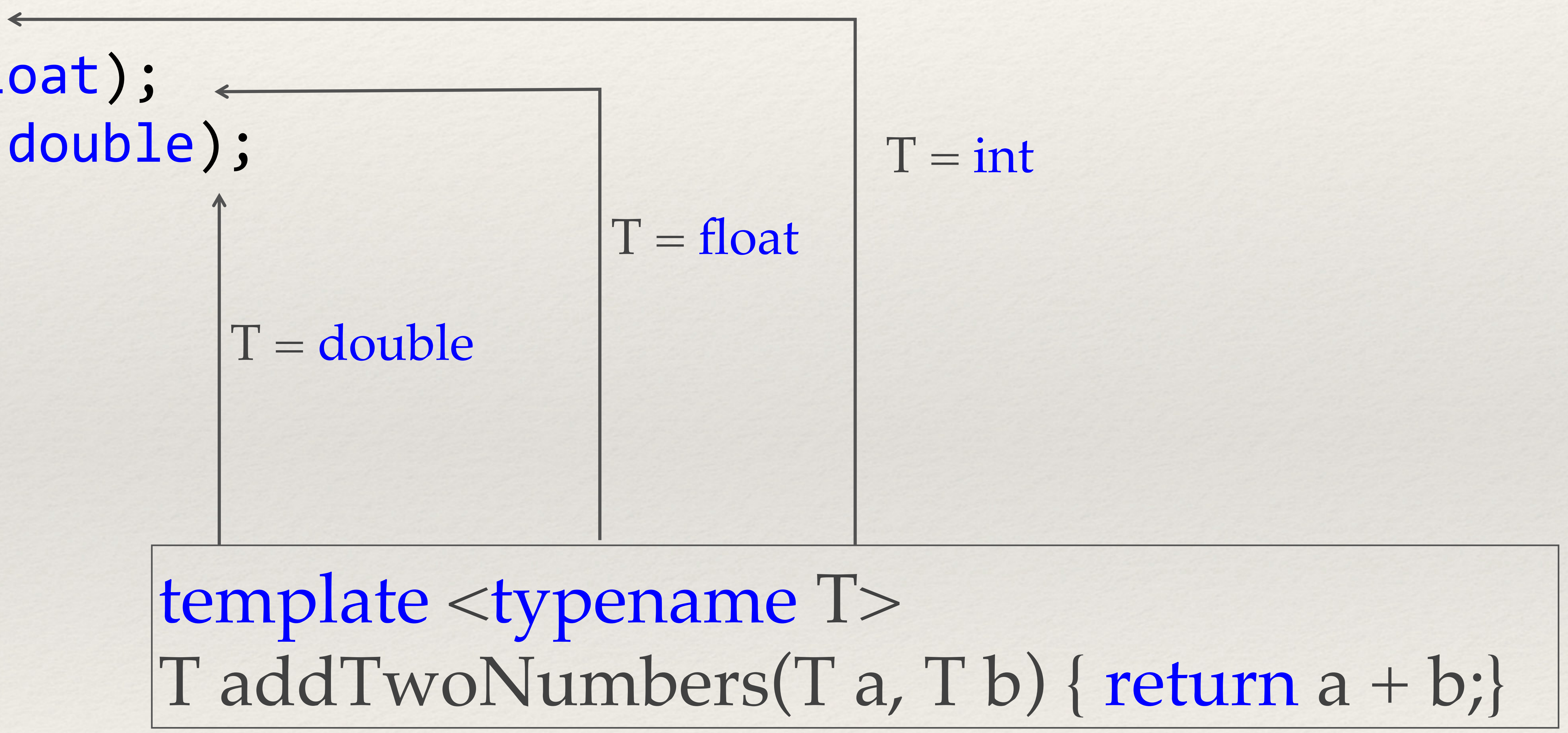
- Templates
  - Easily create a large range of related functions or classes
    - Function template
    - Class template

# Motivation of function template

- Using function overloading:

```
int addTwoNumbers(int, int);
float addTwoNumbers(float, float);
double addTwoNumbers(double, double);
```

- Disadvantages:



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

# Function Templates Syntax

- The function template declaration:

```
template <typename T>  //tell the compiler we are using a template

//T represents the variable type: any fundamental type or user-defined type
T  functionName(T parameter1, T parameter2, ...);
```

- The function template definition:

```
template <typename T>
T functionName(T  parameter1, T  parameter2, ...)
{
    function statements;
}
```

- Function call: as normal, but make sure the type constraints

## Example

```
template <typename T>

T addTwoNumbers(T a, T b) { return a + b;}

cout << addTwoNumbers(1,2) << addTwoNumbers(1.2,3.4) <<
addTwoNumber(1,3.4);
```

## Hand-on Exercise

- Write a function that can return the maximum value of an array of any type and test this function with at least two calls in main function with two different arrays on different element types?

# Class Templates

- Class templates
  - Allow type-specific versions of generic classes
- Format:

  **template <class T>**

  **class** *ClassName*{

  **definition**

  }
- To create an object of the class, type

  *ClassName*< *type* > **myObject**;

  Example: **Stack< double > doubleStack**;

```
template <class T>
class Stack {
   static const int MAX_BUFF = 20;
   int _top;
   T buff[MAX_BUFF];
   public:
      Stack():_top(0){}
      void push(T const&);
      void pop();
      T top();
      bool empty() {return _top == 0;}
```

```
template <class T>
void Stack<T>::push(T const& ele) {
   if (_top == MAX_BUFF)
      throw out_of_range(“Stack full!”);
   buff[_top++] = ele;
```

## Hand-on Exercise

- Implement the methods pop and top of class Stack and write the main function where Stack is initialized with different types?

## Class Templates and Non-type Parameters

- Can use non-type parameters in templates
  - Default argument
  - Treated as `const`
- Example:

```
template< class T, int elements >
Stack< double, 100 > mostRecentSalesFigures;
```

  - Declares object of type `Stack< double, 100>`

  - This may appear in the class definition:

```
T stackHolder[ elements ]; //array to hold stack
```

  - Creates array at compile time, rather than dynamic allocation at execution time

## Hand-on Exercise

- Redefine class Stack such that the MAX_BUFF is initialized together with element type?

## Templates and Inheritance

- A class template can be derived from a class-template specialization.
- A class template can be derived from a non-template class.
- A class-template specialization can be derived from a class-template specialization.
- A non-template class can be derived from a class-template specialization.

# Templates and Friends

- Friendships allowed between a class template and
  - Global function
  - Member function of another class
  - Entire class
- **`friend`** functions
  - Inside definition of class template **`X`**:
  - **`friend void f1();`**
    - **`f1()`** a **`friend`** of all template classes
  - **`friend void f2( X< T > & );`**
    - **`f2( X< int > & )`** is a **`friend`** of **`X< int >`** only. The same applies for **`float`**, **`double`**, etc.
  - **`friend void A::f3();`**
    - Member function **`f3`** of class **`A`** is a **`friend`** of all template classes

# Templates and Friends (cont)

- **`friend void C< T >::f4( X< T > & );`**
  - **`C<float>::f4( X< float> & )`** is a **`friend`** of **`class X<float>`** only

- **`friend`** classes
  - **`friend class Y;`**
    - Every member function of **`Y`** is a friend with every template class made from **`X`**
  - **`friend class Z<T>;`**
    - Class **`Z<float>`** a **`friend`** of class **`X<float>`**, etc.

# Templates and static Members

- Non-template class
  - **`static`** data members shared between all objects
  - **`static`** data members initialized at global scope
- Template classes
  - Each class (**`int`**, **`float`**, etc.) has its own copy of **`static`** data members
  - **`static`** data members initialized at global scope
  - Each template class gets its own copy of **`static`** member functions

## Reference

- www.geeksforgeeks.org/templates-cpp
- www.programiz.com/cpp-programming/templates