# Creative Software Design

## 12 – Template

Yoonsang Lee
Fall 2022

# Final Exam

- Date & time: **Dec 13, 09:30 - 10:30 am**
- Place: **IT.BT 207**
- Scope: Lecture 8 ~ 13

- **You cannot leave until 30 minutes after the start of the exam** even if you finish the exam earlier.

- That means, **you cannot enter the room after 30 minutes from the start of the exam** (do not be late, never too late!).

- Please bring your **student ID card** to the exam.

- Problem types: true/false, single choice, multiple choices, short answer, fill-in-blank, ...

# Today's Topics

- Intro to Generic Programming
- Function Template
- Class Template
- Review Standard Template Library (STL)
  - A set of C++ template classes
- Templates and Inheritance

# Generic Programming

- Generic programming is a style of computer programming in which
  - Algorithms are written in terms of ***to-be-specified-later*** **types** (or ***generic*** **types**) that are then instantiated when needed for specific types provided as parameters.[wikipedia]

- C++ Template
  - A C++ feature that allows functions and classes to operate with ***generic types***.
  - This allows a function or class to **work on many different data types without being rewritten** for each one.[wikipedia]

- C++ Standard Template Library (STL).
  - Best known example
  - Data containers such as vector, list, map, etc.
  - Algorithms such as sorting, searching, hashing, etc.

# Direct Approach

- Need ***K*** sorting algorithms to handle ***K*** different data types

```
//Suppose we want to sort an integer array.
void SelectionSort(int* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j =i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    int tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

```
// We also want to sort a double array.
void SelectionSort(double* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    double tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

# Generic Approach

- C++ template allows us to avoid this repeated code.
- ***Functions*** and ***classes*** can be ***templated***.

```
// Suppose we want to sort an array of type T.
template <typename T>
void SelectionSort(T* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    T tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

# Function Template

- A generic function description
  - defines a ***function*** in terms of a ***generic type***
    - A specific type, such as *int* or *double*, can be substituted.

- Pass a specific type as an argument to a template
  - Compiler generates a function for that particular type

- Write functions of the same algorithm once for various types.

# Function Template

- Example
  - Swap function

```
template <typename T> //let compiler know that you're about to define a template. Programmers use simple names such as T.
void Swap(T &a, T &b)
{
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

- **The template does not create any functions**
  - It just lets the compiler know how to define a function

# Function Template

```
template <typename T>
void Swap(T &a, T &b)
{
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

**Output:**

i, j = 10, 20.
compiler-generated int swapper:
Now i, j = 20, 10.
x, y = 24.5, 81.7.
compiler-generated double swapper:
Now x, y = 81.7, 24.5.

```
template <typename T> // or class T
void Swap(T &a, T &b);

int main()
{
  int i = 10;
  int j = 20;
  cout << "i, j = " << i << ", " << j << ".\n";
  cout << "compiler-generated int swapper:\n";
  Swap<int>(i,j); // generates void Swap(int &, int &)
  cout << "Now i, j = " << i << ", " << j << ".\n";
  double x = 24.5;
  double y = 81.7;
  cout << "x, y = " << x << ", " << y << ".\n";
  cout << "compiler-generated double swapper:\n";
  Swap<double>(x,y); // generates void Swap(double &, double &)
  cout << "Now x, y = " << x << ", " << y << ".\n";
  return 0;
}
```

# Function Template Instance

- Templates are "instantiated" at compile time.
- "Function template instance"

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
  cout << myMax<int>(3, 7) << endl;
  cout << myMax<char>('g', 'e') << endl;
  return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

# Template Argument Deduction

- You can **omit** any template argument that the compiler can deduce by the usage and context of that function template call.

```
int i = 10;
int j = 20;
Swap<int>(i,j);
```

=

```
int i = 10;
int j = 20;
Swap(i,j);
```

# Overloading Function Templates

```
template <typename T>
void Swap(T &a, T &b)
{
  T temp;
  temp = a;
  a = b;
  b = temp;
}

template <typename T>
void Swap(T* a, T* b, int
n)
{
  T temp;
  for (int i = 0; i < n; i++)  {
    temp = a[i];
    a[i] = b[i];
    b[i] = temp;
  };
}
```

```
int main()
{
  int i = 10, j = 20;
  cout << "i, j = " << i << ", " << j << endl;
  cout << "Swap scalars" << endl;
  Swap(i,j); // generates Swap(int &, int &)
  cout << "i, j = " << i << ", " << j << ".\n";
  cout << "*****************\n" ;
  int d1[] = {1,2};
  int d2[] = {3,4};
  int n = 2;
  cout << "d1[0]=" << d1[0] << ", d1[1]=" << d1[1] << endl;
  cout << "d2[0]=" << d2[0] << ", d2[1]=" << d2[1] << endl;
  cout << "Swap arrays" << endl;
  Swap(d1,d2, n); // generates void Swap(int *, int *, int)
  cout << "d1[0]=" << d1[0] << ", d1[1]=" << d1[1] << endl;
  cout << "d2[0]=" << d2[0] << ", d2[1]=" << d2[1] << endl;
  return 0;
}
```

# Overloading Function Templates

**Output:**

i, j = 10, 20
Swap scalars
i, j = 20, 10.
*******************
d1[0]=1, d1[1]=2
d2[0]=3, d2[1]=4
Swap arrays
d1[0]=3, d1[1]=4
d2[0]=1, d2[1]=2

# Quiz #1

- Go to https://www.slido.com/
- Join #**csd-ys**
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

# Class Template

- Class members can be templated
  - Define a class in a generic fashion (type-independent)
  - Allow to reuse code
    - Inheritance & containment aren't always the solution

```
class Stack1 {
  private:
    Item1 items[10]; // holds stack items
    int top; // index for top stack item
  public:
    Stack();
};
```

```
class Stack2 {
  private:
    Item2 items[10]; // holds stack items
    int top; // index for top stack item
  public:
    Stack();
};
```

# Class Template

- How to use

```
template <typename T>//let the compiler know that you're about to define a template
class Stack
{
  private:
    T items[10]; // holds stack items (type-independent)
    int top; // index for top stack item
  public:
    Stack();
};
```

  - When a template is invoked, ***T*** will be replaced with a specific type.
    - E.g., *int* or *string*
  - In this example, the generic type name ***T*** identifies the type to be stored in the stack.

# Class Template

```
template <typename T>
class mypair {
  T a, b;
  public:
    mypair (T first, T second) {
      a=first; b=second;
    }
    T getmax ();
};

template < typename T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}
```

```
int main ()
{
  int a_i = 100, b_i = 75;
  mypair <int> myobject_i (a_i, b_i);
  cout << "max("<<a_i <<"," << b_i <<")=" <<
myobject_i.getmax() << endl;

  double a_d = 1.5, b_d = -3.5;
  mypair <double> myobject_d (a_d, b_d);
  cout << "max("<<a_d <<"," << b_d <<")=" <<
myobject_d.getmax() << endl;

  return 0;
}
```

**Output:**

max(100,75)=100

max(1.5,-3.5)=1.5

# Class Template Instance

- Templates are "instantiated" at compile time.
- "Class template instance"

```
mypair <int> myobject_i (a_i, b_i);
```

```
class mypair {
  int a, b;
  public:
    mypair (int first, int second) {
      a=first; b=second;
    }
    int getmax ();
};

int mypair<int>::getmax ()
{
  int retval;
  retval = a>b? a : b;
  return retval;
}
```

- A class template instance is a class.
- A class template is **not** a class!

# Class Template: Closer Look at

- Types for the **mypair** <T>
  - Both built-in types and classes are allowed.
  - How about pointers?
    - Won't work very well without major modifications
    - Need to take care

```
int main () {

  int a_i = 100, b_i = 75;
  mypair <int*> myobject_i (&a_i, &b_i);
  cout << "max("<<a_i <<"," << b_i <<")=" << myobject_i.getmax()
<< endl;

  return 0;
}
```

**Output:**

max(100,75)=0x22fe
2c

# Member Function Template

- Can be used to provide additional template parameters other than those of the class template.

```
template <typename T>
class X
{
public:
    template <typename U>
    void mf(const U &u);
};

template <typename T>
template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
}
```

# typename & class keyword

- ‘typename’ can always be replaced by keyword ‘class’.

```
template <class First, class Second> // Same as <typename First, typename Second>.
class Pair {
        First first;
        Second second;
};

template <class First, class Second>
Pair<First, Second> MakePair(const First& first, const Second& second) {
        return Pair<First, Second>(first, second);
}
```

```
int main (){
 Pair<int, int> p = MakePair(10, 10); // == MakePair<int, int>(10, 10);
 Pair<int, int> q = Pair<int, int>(20, 20);
   return 0;
}
```

# Non-type Template Parameter

- A non-type template parameter
  - is a special type of parameter that is **replaced by a value**.
  - e.g., `template<class T, int size>`

- A non-type template argument
  - is provided within a template argument list
  - is **an expression whose value can be determined at compile time**
    - ***constant expressions***
  - is treated as **const**
  - e.g. `Myfilebuf<double, 200>`

# Non-type Template Parameter

```
template<class T, int size>
class Myfilebuf {
        T* filepos;
        int array[size];
public:
        Myfilebuf() { /* ... */ }
        ~Myfilebuf() {}
        ...
};
```

```
int main (){
        Myfilebuf<double,200> x; // create object x of class
        Myfilebuf<double,200.0> y; // error, 200.0 is a double, not an int
        return 0;
}
```

# Non-type Template Parameter

```
template<int i> class C
{
 public:
        Int array[i];
        int k;
        C() { k = i; }
};
```

```
int main (){
    C<100> a; //can be instantiated
    C<200> b;//can be instantiated
    return 0;
}
```

# Quiz #2

- Go to https://www.slido.com/
- Join #**csd-ys**
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

# STL Revisit

- STL defines powerful, **template-based**, reusable components
- STL uses **template-based generic programming**
- **A collection of useful templates** for handling various kinds of data structure and algorithms **with generic types**
  - Containers
    - Data structures that store objects of any type
  - Iterators
    - Used to manipulate container elements
  - Algorithms
    - Operations on containers for searching, sorting and many others

# Containers Revisit

- Sequential container: contiguous blocks of objects
  - Vectors: insertion at end, random access
  - List: insertion anywhere, sequential access
  - Deque (double-ended queue): insertion at either end, random access
- Container adapter
  - Stack: Last In Last Out
  - Queue: First In First Out
- Associative container: a generalization of sequence
  - Indexed by any type (vs. sequential containers are indexed by integers)
  - Set: add or delete elements, query for membership…
  - Map: a mapping from one type (key) to another type (value)

# vector - a resizable array

```
#include <iostream>
#include <vector>
using namespace std;

int main(void){

    vector<int> intVec(10);

    for(int i=0; i< 10; i++){
            cout << "input!";
            cin >> intVec[i];
    }

     for(int i=0; i< 10; i++){
            cout << intVec[i] << " " ;
    }
    cout << endl;
    return 0;
}
```

# STL: vector

- Standard library header <vector>
  - A class template
  - Templated member functions/variables

```
template <class T, class Allocator = allocator<T> >
class vector {
public:
    // types:
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef T value_type;
    typedef Allocator allocator_type;
    typedef typedef allocator_traits<Allocator>::pointer pointer;
    typedef typedef allocator_traits<Allocator>::const_pointer const_pointer;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator>
    const_reverse_iterator;
}
```

# STL: vector

- Standard library header <vector>
  - Constructors/destructor

```
template <class T, class Allocator = allocator<T> >
class vector {
public:
    // construct/copy/destroy:
    explicit vector(const Allocator& = Allocator());
    explicit vector(size_type n);
    vector(size_type n, const T& value,const Allocator& = Allocator());
    template <class InputIterator>
        vector(InputIterator first, InputIterator last,const Allocator& =
Allocator());
    vector(const vector<T,Allocator>& x);
    vector(vector&&);
    vector(const vector&, const Allocator&);
    vector(vector&&, const Allocator&);
    vector(initializer_list<T>, const Allocator& = Allocator());
    ~vector();
}
```

# STL: vector

- Standard library header <vector>
  - Assignment operators / member functions

```
template <class T, class Allocator = allocator<T> >
class vector {
public:
    vector<T,Allocator>& operator=(const vector<T,Allocator>& x);
    vector<T,Allocator>& operator=(vector<T,Allocator>&& x);
    vector& operator=(initializer_list<T>);
    template <class InputIterator>
        void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const T& t);
    void assign(initializer_list<T>);
    allocator_type get_allocator() const noexcept;
}
```

# STL: vector

- Standard library header <vector>
  - Iterators
    - begin(), end(), rbegin(), rend(), …
  - Capacity
    - size(), resize(), capacity(), capacity(), empty(), reserve(), …
  - Element access
    - [], at(), front(), back()
  - Modifiers
    - push_back(), pop_back(), insert(), erase(), swap(), clear(), …

  - Everything is templated!!

# Class template vs. Template class

- C++ standard only uses the term "**class template**".
- Some people interchangeably use these two terms, but I recommend you to only use **"class template"**.
  - E.g., This is a class template, but not a class:

```
template<typename T>
class MyClassTemplate { ... };
```

  - E.g., This is a class, but not a class template:

```
MyClassTemplate<int>
```

# Templates and Inheritance

- Inheritance works the same as with ordinary classes.

```
template<class T>
class CountedQue : public QueType<T> {

  public:
    CountedQue();
    void Enqueue (T newItem);
    void Dequeue (T& item);
    int LengthIs() const;
  private:
    int length;
};
```

# Templates and Inheritance

- Overidding

```
template<class T>
class Base
{
	public:
		void set(const T& val) {data = val;}
	private:
		T data;
};
template<class T>
class Derived : public Base<T>//no class Base, only
class Base<T>
{
	public:
		void set(const T& val);
};
template<class T>
void Derived<T>::set(const T& v){
	Base<T>::set(v);//no class Base, only class Base<T>
}
```

# Templates and Inheritance

- A **derived class template** may have its own template parameters.

```
template<class T>
class Base
{
	public:
		void set(const T& val) {data = val;}
	private:
		T data;
};
template<class T, class U>
class Derived : public Base<T>//no class Base, only class Base<T>
{
	public:
		void set(const T& val);
	private:
		U derived_data;
};
```

# Templates and Inheritance

- A **derived class** may inherit from an explicit instance of the base class template.

```
template<class T>
class Base
{
	public:
		void set(const T& val) {data = val;}
		T get(){ return data;}
	private:
		T data;
};

class Derived : public Base<int>//explicit instance of the base class
{
	public:
		int get(){   return Base<int>::get();        }
};
```

# Templates and Inheritance

- Parameterized inheritance

```
class Shape
{
	public:
			void display() { cout << "show" << endl; }
};

template<class T>
class Rectangle : public T //base class is the template parameter
{
	public:
			void display(){		T::display();		}
};

int main()
{
	Rectangle<Shape> rect;
};
```

# Quiz #3

- Go to https://www.slido.com/
- Join #**csd-ys**
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

# Templates and Static Members

- General classes
  - **Static** member variables are shared between all instances of the same class.

- Classes template instances
  - Each class template instance (== each instantiated class) (e.g., MyTemplate<int>, MyTemplate<double>) has its own copy of **static** member variables.
  - **Static** member variables are shared between all instances of the same instantiated class.

# Templates and Static Members

- Example

```
template <class T> class TemplatedClass{
        public:
                static T x;
};
template <class T> T TemplatedClass<T> ::x = 0;  // static member variables should be defined outside of the class
int main()
{
        TemplatedClass<int>::x = 1;
        cout << TemplatedClass<int>::x << endl;
        cout << TemplatedClass<float>::x << endl;
        return 1;
}
```

**Output**:

```
1
0
```

# Next Time

- Labs for this lecture:
  - Lab1 (next Tue): Assignment 12-1
  - Lab2 (next Thur): Assignment 12-2

- Next lecture:
  - 13 - Exception Handling (The last lecture)