ST. CHARLES COMMUNITY COLLEGE

CPT-182 - Programming in C++

Module 11

Templates, C++ Standard Template Library (STL)

Dayu Wang

- **Function Templates**

→ **Multiple functions may be identical (or nearly identical), differing only in their data types.**

```cpp
void _swap(int& x1, int& x2) {
    int temp = x1;
    x1 = x2;
    x2 = temp;
}
```

```cpp
void _swap(char& c_1, char& c_2) {
    char temp = c_1;
    c_1 = c_2;
    c_2 = temp;
}
```

```cpp
void _swap(string& s_1, string& s_2) {
    string temp = s_1;
    s_1 = s_2;
    s_2 = temp;
}
```

→ **Can we write one function that can swap whatever data type?**

```cpp
template<typename Item_Type>
void _swap(Item_Type& x, Item_Type& y) {
    Item_Type temp = x;
    x = y;
    y = temp;
}
```

- **A function template is a function definition having a special type parameter that may be used in place of types in the function.**

→ Syntax

```
template<typename Item_Type>
// Or...
template<class Item_Type>
```

Item_Type can be any identifier.

Item_Type is known as a type parameter and can be used throughout the function for any parameter types, return types, or local variable types.

The identifier is known as a template parameter, and may be various items such as an int, double, char, or string, or a pointer or reference, or even another template parameter.

```
template<class T>
T min_of_three(const T& x, const T& y, const T& z) {
    if (x < y) {
        if (x < z) { return x; }
        else { return z; }
    } else {
        if (y < z) { return y; }
        else { return z; }
    }
}
```



```
int main() {
    int x_1 = 10, x_2 = -3, x_3 = 150;
    string s_1 = "CPT-182-83", s_2 = "CPT-106-02", s_3 = "CPT-281-42";
    cout << min_of_three(x_1, x_2, x_3) << endl;
    cout << min_of_three(s_1, s_2, s_3) << endl;
    system("pause");
    return 0;
}
```

Console

```
-3
CPT-106-02
```

→ How about user defined classes?

Can we use the same function to return the smallest Rectangle object?

```
class Rectangle {
private:
    unsigned int width, height;
public:
    Rectangle(unsigned int = 0, unsigned int = 0);
    unsigned int area() const;
    // Must overload the "<" operator.
    bool operator < (const Rectangle&) const;
    friend ostream& operator << (ostream&, const Rectangle&);
};
```

```
Rectangle::Rectangle(unsigned int width, unsigned int height) {
    this->width = width;
    this->height = height;
}
unsigned int Rectangle::area() const { return width * height; }
bool Rectangle::operator < (const Rectangle& other) const {
    return area() < other.area();
}
ostream& operator << (ostream& out, const Rectangle& rectangle) {
    out << "Width:\t" << rectangle.width << endl;
    out << "Height:\t" << rectangle.height;
    return out;
}
```

```
int main() {
    Rectangle r_1(10, 6), r_2(3, 3), r_3(1, 8);
    cout << min_of_three(r_1, r_2, r_3) << endl;
    system("pause");
    return 0;
}
```

Console

```
Width:  1
Height: 8
```


- **Class Templates**

→ **Multiple classes may be identical (nearly identical), differing only in their data types.**

```
class Pair {
private:
    int first;
    int second;
// Public section
};
```

```
class Pair {
private:
    string first;
    string second;
// Public section
};
```

```
class Pair {
private:
    int first;
    string second;
// Public section
};
```

```
class Pair {
private:
    string first;
    int second;
// Public section
};
```

→ **Can we use templates to group any two variables into a Pair object?**

```cpp
template<typename Type_1, typename Type_2>
class Pair {
private:
    Type_1 first;
    Type_2 second;
public:
    Pair(const Type_1&, const Type_2&);
    virtual ~Pair();  // To avoid warning messages
    Type_1 get_first() const;
    void set_first(const Type_1&);
    Type_2 get_second() const;
    void set_second(const Type_2&);
};
```

You can create multiple type parameters.

→ **Normally, we define a `class` in a <u>header file</u> (`.h`). Then, implement the `class` in a `.cpp` file.**

- **[Important] However, if the `class` is a <u>template class</u>, then we have to implement the `class` in the same header file, instead of creating a `.cpp` file.**
- **We need to put the "`template<typename Type_1, typename Type_2>`" at the beginning of each function implement.**
- **For each class reference, you need to specify the types in "<>", except for the constructors.**


```cpp
template<typename Type_1, typename Type_2>
Pair<Type_1, Type_2>::Pair(const Type_1& first, const Type_2& second) {
    this->first = first;
    this->second = second;
}

template<typename Type_1, typename Type_2>
Pair<Type_1, Type_2>::~Pair() {}

template<typename Type_1, typename Type_2>
Type_1 Pair<Type_1, Type_2>::get_first() const { return first; }

template<typename Type_1, typename Type_2>
void Pair<Type_1, Type_2>::set_first(const Type_1& first) {
    this->first = first;
}

template<typename Type_1, typename Type_2>
Type_2 Pair<Type_1, Type_2>::get_second() const { return second; }

template<typename Type_1, typename Type_2>
void Pair<Type_1, Type_2>::set_second(const Type_2& second) {
    this->second = second;
}
```

- **Friend Functions in Template Classes**

→ **A friend function in template class should have its own type parameters.**

```
template<typename Type_1, typename Type_2>
class Pair {
private:
    Type_1 first;
    Type_2 second;
public:
    Pair();
    virtual ~Pair();
    Type_1 get_first() const;
    void set_first(const Type_1&);
    Type_2 get_second() const;
    void set_second(const Type_2&);

    template<typename T1, typename T2>
    friend ostream& operator << (ostream&, const Pair<T1, T2>&);
};
```

```
template<typename T1, typename T2>
ostream& operator << (ostream& out, const Pair<T1, T2>& pair) {
    out << '(' << pair.first << ", " << pair.second << ')';
    return out;
}
```



```
int main() {
    Pair<int, int> p_1(3, 3);
    Pair<string, double> p_2("PI", 3.14);
    cout << p_1 << endl << p_2 << endl;
    system("pause");
    return 0;
}
```

Console

```
(3, 3)
(PI, 3.14)
```

→ **[Good Practice] Avoid ~~friend functions~~ in template classes.**

You can use getters and setters to access/update the private data fields of the class.

```
// Stream insertion operator
template<class Type_1, class Type_2>
ostream& operator << (ostream& out, const Pair<Type_1, Type_2>& p) {
    return out << '(' << p.get_first() << ", " << p.get_second() << ')';
}
```

- **Standard Template Library (STL)**

Class	Header
`pair<Type_1, Type_2>`	`<utility>`
`vector<Item_Type>`	`<vector>`
`list<Item_Type>`	`<list>`
`stack<Item_Type>`	`<stack>`
`queue<Item_Type>`	`<queue>`
`priority_queue<Item_Type>`	`<queue>`
`set<Item_Type>`	`<set>`
`unordered_set<Item_Type>`	`<unordered_set>`
`map<Key_Type, Value_Type>`	`<map>`
`unordered_map<Key_Type, Value_Type>`	`<unordered_map>`

- **Linked Lists**
  - → **Linked list is an abstract data type (ADT).**
    - **You need `#include <list>` to use the `list` template class.**
    - **List is a sequential (linear) data container.**
  - → **Although lists behave similar to vectors, they are completely different data structures.**
    - **Items stored in a vector are physically connected in the memory.**
    - **Items stored in a list are logically connected in the memory.**
  - → **Common class-member functions in `list<Item_Type>`:**

Function	Behavior
`size_t size() const;`	Returns the number of elements in the list.
`bool empty() const;`	Tests whether the list is empty.
`Item_Type& front();` `const Item_Type& front() const;`	Returns the element at the front end of the list.
`Item_Type& back();` `const Item_Type& back() const;`	Returns the element at the rear end of the list.
`void push_front(const Item_Type&);`	Inserts an element to the front end of the list.
`void push_back(const Item_Type&);`	Inserts an element to the rear end of the list.
`void pop_front();`	Deletes an element from the front end of the list.
`void pop_back();`	Deletes an element from the rear end of the list.

• **Linked Lists**

→ [Important] `list` does not support ~~index~~.

You cannot use index to iterate over a list.

→ If you need to iterate over a list, you must use iterators.

Function	Behavior
`iterator begin();` `const_iterator begin() const;`	Generates an iterator positioned on the first element in the list.
`iterator end();` `const_iterator end() const;`	Generates an iterator positioned just after the last element in the list.

```
list<char> li;
li.push_back('x');
li.push_front('y');
li.push_back('z');
li.push_front('p');

for (list<char>::const_iterator it = li.begin(); it != li.end(); it++) {
    cout << *it << '\t';
}
```

Console
```
p	y	x	z
```

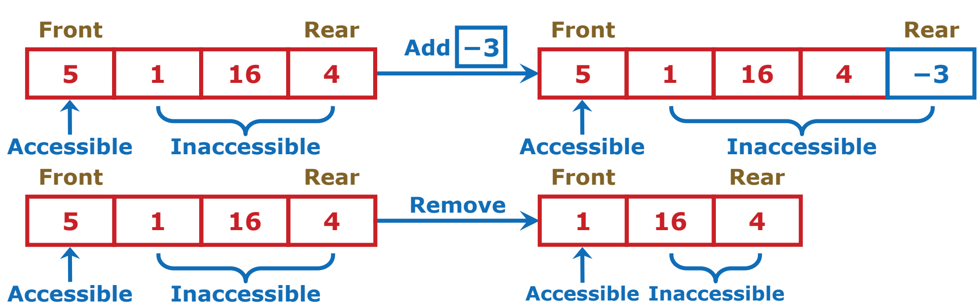

• **Queues**

→ Queue is an abstract data type (ADT).

- You need `#include <queue>` to use the `queue` template class.
- Queue is a sequential (linear) data container which has two ends: front end and rear end.

→ Queue implements the first-in-first-out feature.

- When you insert an element into a queue, the element will always be inserted to rear end.
- When you delete an element from a queue, you will always delete the element at front end.
- In a queue, only the front element is accessible.



• **Queues**

→ Common class-member functions in `queue<Item_Type>`:

Function	Behavior
`size_t size() const;`	Returns the number of elements in the queue.
`bool empty() const;`	Tests whether the queue is empty.
`Item_Type& front();` `const Item_Type& front() const;`	Returns the element at the front end of the queue.
`void push(const Item_Type&);`	Inserts an element to the rear end of the queue.
`void pop();`	Deletes an element from the front end of the queue.

→ Queue does not support ~~index~~, nor ~~iterator~~.

You cannot ~~iterate over a queue~~.

• **Priority Queues**

→ Template class `priority_queue<Item_Type>` in the `<queue>` library pushes/pops elements to/from the queue in priority order.

• **Stacks**

→ Template class `stack<Item_Type>` in the `<stack>` library implements the last-in-first-out feature.

• **Pairs**

→ Pairs are abstract data type (ADT).

- You need `#include <utility>` to use the `pair` template struct.
- A `pair<Type_1, Type_2>` object groups two variables of `Type_1` and `Type_2` together.

→ Since `pair` is a struct, you can access its data fields directly.

- First value in the pair can be accessed via `.first`.
- Second value in the pair can be accessed via `.second`.

→ To form a `pair` object by grouping two values, you need to use the `make_pair()` function.

• **Search and Sort**

→ The `find()` function can be used to search for an element in a vector or list.

```
iterator find(iterator begin, iterator end, const Item_Type& target);
```

- If `target` is found, then the function will return an iterator on that element.
- If `target` is not found, then the function will return an iterator just after the last element.

You can use `find(target) == end()` to test whether an element is in a vector or list.

→ The `sort(iterator begin, iterator end)` function can sort the elements between the iterators.

Use `sort(v.begin(), v.end())`/`sort(li.begin(), li.end())` to sort the entire vector/list.

- **Set**
  - → A set is a collection of objects.
  - → Characteristics: membership, unordered, unique
- **Set Operations**
  - → Membership testing
  - → Inserting elements into a set
  - → Removing elements from a set
  - → Union

    $\{1, 2, 5, 7\} \cup \{2, 3, 4, 5\} = \{1, 2, 3, 4, 5, 7\}$
  - → Intersection

    $\{1, 2, 5, 7\} \cap \{2, 3, 4, 5\} = \{2, 5\}$
  - → Difference

    $\{1, 2, 5, 7\} - \{2, 3, 4, 5\} = \{1, 7\}$
  - → Subset

    $\{1, 2, 5, 7\} \subset \{1, 2, 3, 4, 5, 7\} =$ `true`

- **Set in C++ Standard Template Library**
  - → The `set` class in `<set>` library
    - A `set` is implemented by balanced binary search tree (a hierarchical data structure).
    - A `set` does not allow ~~duplicate element keys~~.
    - To iterate over a `set`, you need to use iterators.
  - → Common class-member functions of `set<Item_Type>`

Function	Behavior
`size_t size() const;`	Returns the number of elements in the set.
`bool empty() const;`	Tests whether the set is empty.
`pair<iterator, bool> insert(const Item_Type&);`	Inserts an element into the set.
`void erase(iterator);`	Deletes the element at iterator position.
`size_t erase(const Item_Type&);`	Deletes an element.
`void clear();`	Deletes all the elements in the set.
`iterator begin();` `const_iterator begin() const;`	Generates an iterator positioned at the first element in the set.
`iterator end();` `const_iterator end() const;`	Generates an iterator positioned just after the last element in the set.
`iterator find(const Item_Type&) const;`	Tests whether an element is in the set.

• **Insertion and Search**

→ **The `insert()` function**

- **The function inserts a new key into the set.**
- **The function returns a pair object containing an iterator and a boolean.**
- **If insert was successful, the function returns an iterator on the newly inserted key and `true`.**
- **If failed (the key is already in the set), the function returns the `end()` iterator and `false`.**

→ **The `find()` function**

- **The function searches for a key in the set.**
- **If the key was found, the function returns an iterator on the found key.**
- **If the key was not found, the function returns the `end()` iterator.**

• **Unordered Set**

→ **The `unordered_set` class in `<unordered_set>` library**

- **An `unordered_set` is implemented by hash table (not required to understand).**
- **An `unordered_set` does not allow ~~duplicate element keys~~.**
- **To iterate over an `unordered_set`, you need to use iterators.**

→ **Common class-member functions of `unordered_set<Item_Type>`**

Function	Behavior
`size_t size() const;`	Returns the number of elements in the set.
`bool empty() const;`	Tests whether the set is empty.
`pair<iterator, bool> insert(const Item_Type&);`	Inserts an element into the set.
`iterator erase(iterator);`	Deletes the element at iterator position.
`size_t erase(const Item_Type&);`	Deletes an element.
`void clear();`	Deletes all the elements in the set.
`iterator begin();` `const_iterator begin() const;`	Generates an iterator positioned at the first element in the set. There is **no** ~~guaranteed first element~~ in an unordered set.
`iterator end();` `const_iterator end() const;`	Generates an iterator positioned just passed all the elements in the set.
`iterator find(const Item_Type&) const;`	Tests whether an element is in the set.

- **[Sample Question] Write a function that returns all the unique values in a vector of integers.**
  - **The input vector is not sorted in any way.**
  - **Your function can return the unique values in any order.**

→ We need to use a set to store all the "discovered values".

Since order does not matter, we can just use `unordered_set`**.**

```
/** Finds all the unique values in a vector.
    @param vec: input vector (not sorted in any way)
    @return: a vector containing all the unique values in the input vector
*/
vector<int> unique_values(const vector<int>& vec) {
    unordered_set<int> us;  // Create an unordered_set to store all the discovered values.
    vector<int> result;  // Stores the unique values.
    for (size_t i = 0; i < vec.size(); i++) {
        if (us.find(vec.at(i)) == us.end()) {
            result.push_back(vec.at(i));
            us.insert(vec.at(i));
        }
    }
    return result;
}
```


- **Maps**

→ The map (also called associative array) is related to the set.

Mathematically, it is a set of ordered pairs whose elements are known as the key and the value.

→ The key is required to be unique, but the value is not necessarily unique.

Example: { (J, Jane), (B, Bill), (S, Sam), (B1, Bob), (B2, Bill) }

→ A map can be used to enable efficient storage and retrieval of information in a table.

Type of Item	Key	Value
University student	Student ID	Student name, address, major, GPA
Online store customer	E-mail address	Customer name, shopping cart
Inventory item	Part ID	Description, quantity, price

→ What are the differences between map and indexed collection (array)?

Maps can use user-specified keys to access the information stored in it.

Arrays can only use indices as keys to access the information stored in it.

→ A map is essentially a set of (key, value) pairs.

- **Maps in C++ Standard Template Library**

→ **The `map` class in `<map>` library**

- **A `map` is implemented by balanced binary search tree (a hierarchical data structure).**
- **A `map` does not allow ~~duplicate keys~~, but allow duplicate values.**
- **To iterate over a `map`, you need to use iterators.**

→ **Common class-member functions of `map<Key_Type, Value_Type>`**

Function	Behavior
`size_t size() const;`	Returns the number of **entries** (key-value pairs) in the map.
`bool empty() const;`	Tests whether the map is empty.
`pair<iterator, bool> insert(const pair<Key_Type, Value_Type>&);`	Inserts an entry into the map.
`void erase(iterator);`	Deletes the entry at iterator position.
`size_t erase(const Key_Type&);`	Deletes an entry with the given key.
`void clear();`	Deletes all the entries in the map.
`iterator begin();` `const_iterator begin() const;`	Generates an iterator positioned at the first entry in the map.
`iterator end();` `const_iterator end() const;`	Generates an iterator positioned just after the last entry in the map.
`iterator find(const Key_Type&) const;`	Tests whether a key is in the map.

- **Insertion and Search**

→ **The `insert()` function**

- **The function inserts a new entry (key-value pair) into the map.**
- **The function returns a pair object containing an iterator and a boolean.**
- **If insert was successful, the function returns an iterator on the newly inserted entry and `true`.**
- **If failed (the key is already in the map), the function returns the `end()` iterator and `false`.**

→ **The `find()` function**

- **The function searches for a key (not a value) in the map.**
- **If the key was found, the function returns an iterator on the found entry (key-value pair).**
- **If the key was not found, the function returns the `end()` iterator.**

- **Unordered Map**

→ **The `unordered_map` class in `<unordered_map>` library**

- **An `unordered_map` is implemented by hash table (not required to understand).**
- **An `unordered_map` does not allow ~~duplicate keys~~, but allow duplicate values.**
- **To iterate over an `unordered_map`, you need to use iterators.**

→ **Common class-member functions of `unordered_map<Key_Type, Value_Type>`**

Function	Behavior
`size_t size() const;`	Returns the number of **entries** (key-value pairs) in the map.
`bool empty() const;`	Tests whether the map is empty.
`pair<iterator, bool> insert(const pair<Key_Type, Value_Type>&);`	Inserts an entry into the map.
`iterator erase(iterator);`	Deletes the entry at iterator position.
`size_t erase(const Key_Type&);`	Deletes an entry with the given key.
`void clear();`	Deletes all the entries in the map.
`iterator begin();` `const_iterator begin() const;`	Generates an iterator positioned at the first entry (**no** guarantee) in the map.
`iterator end();` `const_iterator end() const;`	Generates an iterator positioned just passed all the entries in the map.
`iterator find(const Key_Type&) const;`	Tests whether a key is in the map.

- **Member Accessing in Maps and Unordered Maps**

→ **The [] operator in `map` and `unordered_map`**

```
Value_Type& operator [] (Key_Type&);
```

- **It returns the value associated with the given key.**
- **If the given key does not exist in the map, then a new entry is inserted into the map.**
- **This works for both lvalue and rvalue.**

→ **The `.at()` class-member function in `map` and `unordered_map`**

```
Value_Type& at(const Key_Type&);
const Value_Type& at(const Key_Type&) const;
```

- **It returns the value associated with the given key.**
- **If the given key does not exist in the map, then an exception will be thrown.**
- **For lvalue, it returns a reference; for rvalue, it returns a `const` reference.**

- **[Sample Question] Frequency Count**

  → **Given a vector of integers, please find the element which has the largest frequency.**

  [Example] In vector [4, 0, 9, 2, 9, 1, 2, 2, 4, 3, 2, 9, 0, 0, 1], value 2 appears 4 times, which is the highest frequency.

  → **[Solution] Use a map to store the frequency of all the discovered values in the vector.**

```
int most_frequent_value(const vector<int>& vec) {
    // Store the frequency of each value into an unordered map.
    unordered_map<int, unsigned int> m;
    for (size_t i = 0; i < vec.size(); i++) {
        if (m.find(vec.at(i)) == m.end()) { m[vec.at(i)] = 1; }
        else { m[vec.at(i)]++; }
    }
    // Find the max value in the unordered map.
    int result = 0;
    unsigned int f = 0;
    for (unordered_map<int, unsigned int>::iterator it = m.begin(); it != m.end(); it++) {
        if (it->second > f) {
            result = it->first;
            f = it->second;
        }
    }
    return result;
}  // Time complexity: O(n) -> Average
```