Week 11 Lecture 2

NWEN 241
Systems Programming

Jyoti Sahni

jyoti.sahni@ecs.vuw.ac.nz

Templates

Templates

- Templates are a feature of C++ that allows writing generic programs.
- Templates allow a functionality to be adapted to more than one type or class **without** repeating the entire code for each type.
- In 1990, Alex Stepanov and Meng Lee of Hewlett Packard Laboratories extended C++ with a library of class and function templates which has come to be known as the Standard Template Library (STL).
- In 1994, STL was adopted as part of ANSI/ISO Standard C++

Templates

- Template parameters are used to pass data types as an argument

```
<class TYPE>
```

OR

```
<typename TYPE>
```

- The concept of templates can be used in different ways:
  - Function Templates
  - Class Templates
  - Variable Templates (Since C++14)

Function Templates

- Function templates represents a family of functions.
- A ***function template*** behaves like any other function except that the template can have arguments of many different types
- The actual meaning of TYPE would be deduced by compiler depending on the argument passed to this function.

```
template <class TYPE> function_declaration;
```

```
template <typename TYPE> function_declaration;
```

Example

```
void printMaxInt(int a, int b)
{
     cout<<"Max "<< (a > b ? a : b);
}
```

```
void printMaxFloat(float a, float b)
{
    cout<<"Max "<< (a > b ? a : b);
}
```

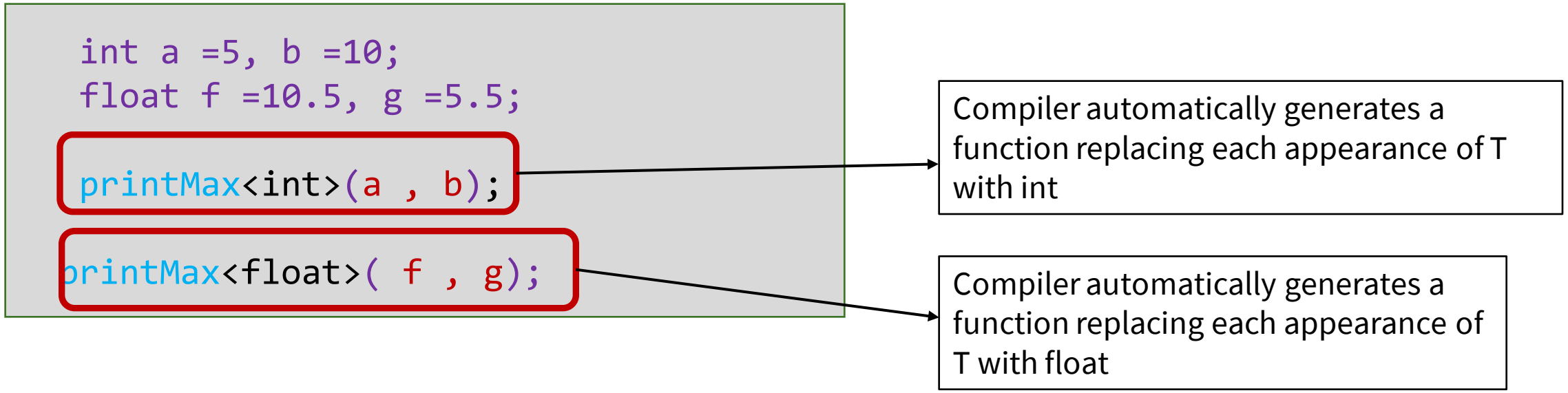
```
template <class T>

void printMax(T a, T b){

  cout<<"Max "<< (a > b ? a : b);
}
```

```
printMax<int>(a , b);
printMax<float>(a , b);
```

Example

- When a compiler sees the definition of a template, it does not **generate code**.

- It generates code only when we **instantiate** a specific instance of the template.



- If a template function is instantiated for a particular data-type, compiler would re-use the same function' instance , if the function is invoked again for same data-type.

Example

```
template <class T>

void printMax(T a , T b){

  cout<<"Max: "<< (a > b ? a : b);}
}
```

```
printMax(5,10);
```

Instantiates `printMax(int a, int b)`

```
printMax(5.0, 10.0);
```

Instantiates `printMax(float a, float b)`

```
printMax(5,10.0);
```

Complier dependent. May generate error.

```
printMax<float> printMax(5,10.0);
```

Function Templates

- STL provides template definitions for many programming tasks
- The header `<algorithm>` defines a collection of function templates. (http://www.cplusplus.com/reference/algorithm/)
- Use them! Do not reinvent the wheel!
- **Searching and sorting** : `find(), find_if(), count(), count_if(), min(), max(), binary_search(),lower_bound(), upper_bound(), sort()`
- **Comparing**: `equal()`
- **Rearranging and copying**: `unique(), replace(), copy(), remove(),reverse(), random_shuffle(), merge()`
- **Iterating**: `for_each()`

Class Templates

- Class Templates allow a class to have members that use **template parameters** as types.
- Class templates represents a family of classes.
- Each parameter in a Template declaration may be:
  - a **type** template parameter
    - `template <class T>`
  - a **non-type** template parameter
    - Integrals are the most used non-types
    - `template<class T, int SIZE>`
  - a **template template** parameter
    - Pass templates as arguments

```
template <parameter list> class_declaration;
```

Type Parameter

```
template <class T>
class Item
{
     T Data;
public:
     Item() : Data(T())
     {}
     void SetData(T nValue)
         {Data = nValue;}

     T GetData() const
         {return Data;}

     void PrintData()
        {cout << Data;}
};
```

```
Item <int> i1;
```

class template Item instantiates Item<int>

```
Item <float> f1;
```

class template Item instantiate Item<float>

Note: We require to explicitly pass template type, unlike function template instantiation, where arguments of function itself helps the compiler to deduce template type arguments; with class templates you must.

Template Instantiation

- Template **instantiation** involves generating a compiled code for a particular combination of template arguments .

- Instantiation of a class template doesn't instantiate any of its member functions unless they are used.

```
template < class T >
class item{
    T a;
public:
    item():a(T()){}
    void disp(){
        cout <<a % 10;
    }};
```

```
item <float> f1; //fine

f1.disp(); //generates error
```

Defining member functions outside class template

```
template < class T >
class item{
    T a;
public:
    item():a(T()){}
    void disp();
    T getData();
};

template < class T >
    void item<T>::disp(){
        cout<<"a ="<<a;}

template < class T >
    T item<T>::getData(){
        return a;}
```

To define member functions outside the declaration of the class template, we must always precede the function definition with the **`template <...>`** prefix

Non Type parameter

```
template < class T, int N >
class item{
    T a[N];
public:
    void setmember(int l, T v)
    {
      a[l] =v;
    }
    T getmember(int l)
    {
      return a[l];
    }
};
```

```
item<int ,2> item1;
```

```
item1.setmember(0,90);
```

Variable Template (C++14)

- A variable template defines a family of variables.

```
template < parameter-list >
variable-declaration;
// The declared variable name becomes a template name.
```

```
template<class T>
T pi = T(3.14159265358979323851);
```

```
int main(){

    cout<<pi<int> <<endl;
    cout<<pi<float> <<endl;
}
```

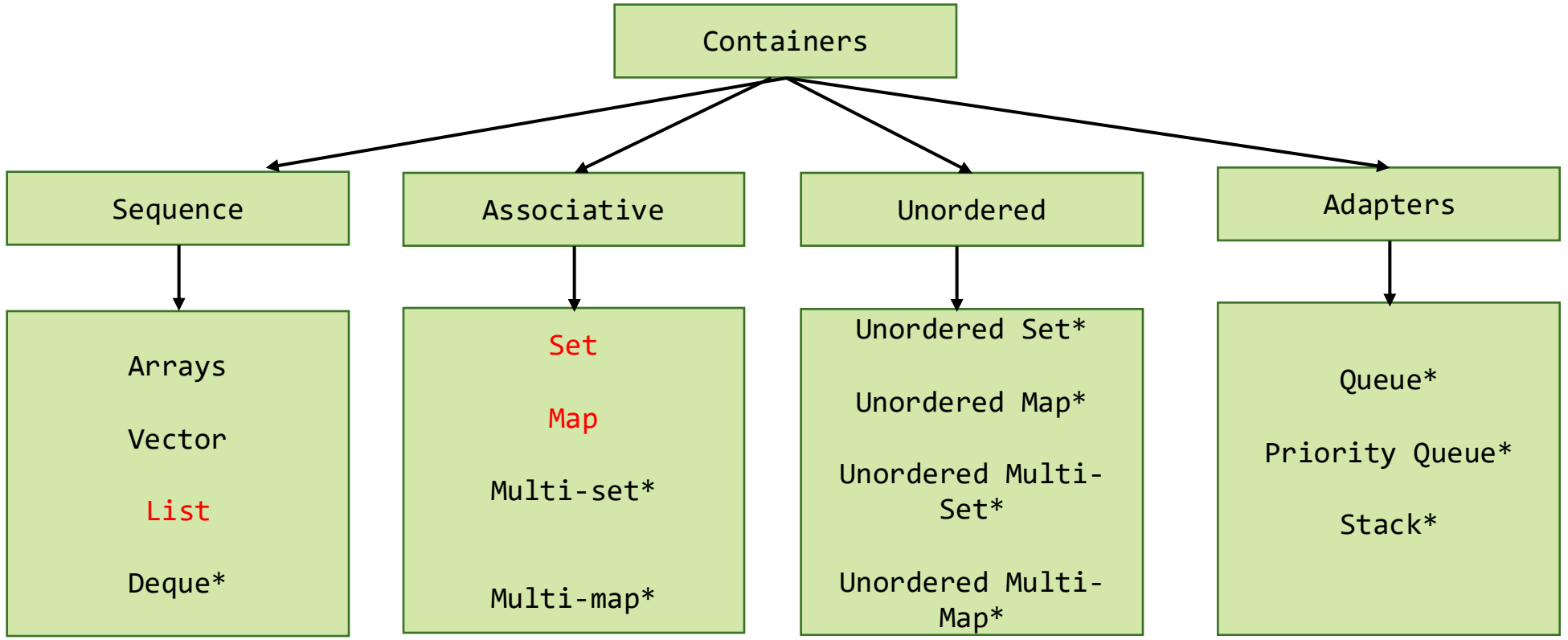
Output:

```
3
3.14159
```

Containers

Recap: Containers

- Containers or container classes store objects and data.



can be accessed sequentially.

Sorted; can be quickly searched ($O(\log n)$ complexity).

unsorted (hashed); can be quickly searched ($O(1)$ amortized, $O(n)$ worst-case complexity)

Have constraints on storage and retrieval

List

- **Forward list** - Defined in `<forward _list>` (Singly linked list)
- **list** - Defined in `<list>` (Doubly linked list)

```
template< class T, class Allocator = std::allocator<T>> class list/forward_list;
```

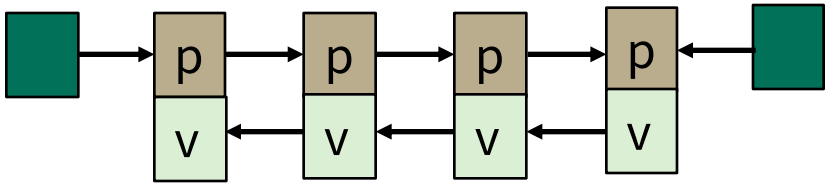
The type of the elements.

Default template parameter.

Is an handle to a heap. It used to acquire/release memory and to construct/destroy the elements in that memory.

List

- A list is a doubly linked list.



- Allows non-contiguous memory allocation.
- We use a list for sequences where we want to insert and delete elements without moving other elements.
- It supports both forward and backward traversal. As compared to vector, list has slow traversal. Unlike vectors, list does **not** support random access
- Involves constant time insertion and removal of elements at the beginning or the end, or in the middle.

List Iterators

- Bi-directional iterator

`iterator`	`Iterator used to iterate through a list.`
`const_iterator`	`Const iterator used to iterate through a list.`
`reverse_iterator`	`Iterator used to iterate backwards through a list.`
`const_reverse_iterator`	`Const iterator used to iterate backwards through a list.`

Basic List Operations

Operation	Description
Iterators	begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend() {c – constant, r –reverse (end to beg)}
Element Access	~~at(), operator[ ],~~ front(), back() {front() / back()– reference to the first / last element}
Capacity	empty(), size(), max_size(), ~~resize(),capacity(), reserve(), shrink_to_fit()~~
Modifiers	clear(), insert(iter_pos, value), emplace(pos, args), erase(pos), push_back(value), emplace_back(args), pop_back(), swap(list_to_swap_with), push_front(value), pop_front() {clear() erases all elements erase() deletes element at a given position push_back() / pop_back() inserts / deletes element at the end, insert() inserts *by copying* an object at a given location, emplace() inserts *in-place* at a given location swap() swap contents of list with another list }

Example

```
class A{
   int a;
   int b;

public:
   A(int x, int y):a(x),b(y){}

   A(){}

void show(){
   cout<<"a = "<<a<<" "<<"b = "<<b<<endl;
    }
};
```

```
list<A> listA;
```

```
int i = 1;
int j = 2;
listA.push_back(A(i,j));
```

```
listA.size()
```

```
listA.at(2).show(); //not allowed
```

```
//Arithematic operations on iterator
//not allowed
listA.insert(listA.begin()+2,{1,2});
```

```
listA.emplace(listA.begin(),-1,-2);
```

```
int main(){
    list<A> listA;
    for (int i = 0; i <= 4; i++)
        listA.push_back(A(i,i));

    auto it=listA.begin();
    cout<<"Element at Loc 2"<<endl;
    int count;
    for(count =0; count<2; count++)
        it++;
    (*it).show();

    cout<<"Inserting a new element {-1, -1} at loc "
<<count<<" using insert: "<<endl;

     listA.insert(it,{-1,-1});

    cout<<"Inserting a new element {-1 , -2} at begin
using emplace: "<<endl;
    listA.emplace(listA.begin(),-1,-2);
    for(auto i =listA.begin();i!=listA.end();i++)
        (*i).show();
    return 0; }
```

```
Element at Loc 2
a = 2 b = 2
Inserting a new element{ -1 -1 } using insert:
Inserting a new element{-1 , -2} using
emplace:
a = -1 b = -2
a = 0 b = 0
a = 1 b = 1
a = -1 b = -1
a = 2 b = 2
a = 3 b = 3
a = 4 b = 4
```

Next Lecture

- **This week** (Tutorial Style Lecture):
  - Friend functions, Templates and Operator overloading

- **Next Week :**
  - Associative containers
  - Smart pointers