ACA Summer School 2014
Advanced C++

Pankaj Prateek

ACA, CSE, IIT Kanpur

July 4, 2014

Templates

- Generic Programming: style of programming which is free of types
- Templates are foundations of generic programming
- Templates allow us to write a generic function with types known dynamically (Remember function overloading!)

Templates

- Generic Programming: style of programming which is free of types
- Templates are foundations of generic programming
- Templates allow us to write a generic function with types known dynamically (Remember function overloading!)

Templates

- Generic Programming: style of programming which is free of types
- Templates are foundations of generic programming
- Templates allow us to write a generic function with types known dynamically (Remember function overloading!)

Function Templates

```
void PrintInt(int n) {
  cout<<''Data=''<<n<<endl;
}
void PrintFloat(float n) {
  cout<<''Data=''<<n<<endl;
}
```

Apart from the input data type, both functions are identical

Function Templates

```
void PrintInt(int n) {
  cout<<''Data=''<<n<<endl;
}
void PrintFloat(float n) {
  cout<<''Data=''<<n<<endl;
}
```

Apart from the input data type, both functions are identical

Function Templates

```
Template<typename T>
void Print(T n) {
  cout<<''Data=''<<n<<endl;
}
```

- The first line tells the compiler what follows is a function template
- Actual meaning of T, template-type parameter, would be deduced by the compiler from the arguement to the function

Function Templates

```
Template<typename T>
void Print(T n) {
  cout<<''Data=''<<n<<endl;
}
```

- The first line tells the compiler what follows is a function template
- Actual meaning of T, template-type parameter, would be deduced by the compiler from the arguement to the function

Function Templates

```
Template<typename T>
void Print(T n) {
  cout<<''Data=''<<n<<endl;
}
```

- The first line tells the compiler what follows is a function template
- Actual meaning of T, template-type parameter, would be deduced by the compiler from the arguement to the function

Function Templates: Why???

- The functions are generated by the compiler
- The real benefit is that the programmer does not have to copy-paste the code, he does not need to write down a new overload for each new data-type which is created.
- Most compilers make optimizations to the code from templates which would not be possible in their absence

Function Templates: Why???

- The functions are generated by the compiler
- The real benefit is that the programmer does not have to copy-paste the code, he does not need to write down a new overload for each new data-type which is created.
- Most compilers make optimizations to the code from templates which would not be possible in their absence

Function Templates: Why???

- The functions are generated by the compiler
- The real benefit is that the programmer does not have to copy-paste the code, he does not need to write down a new overload for each new data-type which is created.
- Most compilers make optimizations to the code from templates which would not be possible in their absence

Class Templates

- Helpful in defining types whose behaviour is generic and reusable

```
class item {
  int data;
  ....
};
class item {
  float data;
  ....
};
```

- If similar functionality for other data-types is needed, need to duplicate code or maybe even entire class. It incurs code maintenance issues, increases code size at the source code as well as at binary level

Class Templates

- Helpful in defining types whose behaviour is generic and reusable

```
class item {
  int data;
  ....
};
class item {
  float data;
  ....
};
```

- If similar functionality for other data-types is needed, need to duplicate code or maybe even entire class. It incurs code maintenance issues, increases code size at the source code as well as at binary level

Class Templates

- Helpful in defining types whose behaviour is generic and reusable

```
class item {
  int data;
  ....
};
class item {
  float data;
  ....
};
```

- If similar functionality for other data-types is needed, need to duplicate code or maybe even entire class. It incurs code maintenance issues, increases code size at the source code as well as at binary level

Class Templates

- Instead define a class using a template

```
template<typename T>
class item{
  T data;
  ....
};
item<int> item1;
item<float> item2;
```

- With function templates, the type of arguements were sufficient for the compiler to call the correct function, but with class templates, the template type should be explicitly passes in angle brackets

Class Templates

- Instead define a class using a template

```
template<typename T>
class item{
  T data;
  ....
};
item<int> item1;
item<float> item2;
```

- With function templates, the type of arguements were sufficient for the compiler to call the correct function, but with class templates, the template type should be explicitly passes in angle brackets

Class Templates

- Instead define a class using a template

```
template<typename T>
class item{
  T data;
  ....
};
item<int> item1;
item<float> item2;
```

- With function templates, the type of arguements were sufficient for the compiler to call the correct function, but with class templates, the template type should be explicitly passes in angle brackets

Class Templates: Example

item<int> instantiated

```
item<int> item1;
item1.set(29);
item1.print();
```

item<float> instantiated

```
item<float> item1;
item1.set(29.5);
item1.print();
```

- No relation between the two instantiations. For the compiler and the linker, they are two different entities, say two different classes

Class Templates: Example

item<int> instantiated

```
item<int> item1;
item1.set(29);
item1.print();
```

item<float> instantiated

```
item<float> item1;
item1.set(29.5);
item1.print();
```

- No relation between the two instantiations. For the compiler and the linker, they are two different entities, say two different classes

Class Templates: Example

item<int> instantiated

```
item<int> item1;
item1.set(29);
item1.print();
```

item<float> instantiated

```
item<float> item1;
item1.set(29.5);
item1.print();
```

- No relation between the two instantiations. For the compiler and the linker, they are two different entities, say two different classes

Templates: A note

- Instead of keyword `typename` in the template definition, keyword `class` can also be used without any change in the sematics of the code

```
template<typename T>
// can be replaced by
template<class T>
```

- I prefer typename to class because it shows that the template parameter need not be a class

Templates: A note

- Instead of keyword `typename` in the template definition, keyword `class` can also be used without any change in the sematics of the code

```
template<typename T>
// can be replaced by
template<class T>
```

- I prefer typename to class because it shows that the template parameter need not be a class

Templates: A note

- Instead of keyword `typename` in the template definition, keyword `class` can also be used without any change in the sematics of the code

```
template<typename T>
// can be replaced by
template<class T>
```

- I prefer typename to class because it shows that the template parameter need not be a class

Library

- In computer science, a library is a collection of subroutines or classes used to develop software quickly and efficiently
- Usually libraries have functions/constructs which are used very frequently by programmers
- Examples : STL, Boost C++
- When you create a software, and want others to use it too, roll it out in the form of a library
- How to use a particular library depends completely on the implementation of the library
- Except STL, all other C++ libraries are usually non-standard and hence need to be installed separately from g++

Library

- In computer science, a library is a collection of subroutines or classes used to develop software quickly and efficiently
- Usually libraries have functions/constructs which are used very frequently by programmers
- Examples : STL, Boost C++
- When you create a software, and want others to use it too, roll it out in the form of a library
- How to use a particular library depends completely on the implementation of the library
- Except STL, all other C++ libraries are usually non-standard and hence need to be installed separately from g++

Library

- In computer science, a library is a collection of subroutines or classes used to develop software quickly and efficiently
- Usually libraries have functions/constructs which are used very frequently by programmers
- Examples : STL, Boost C++
- When you create a software, and want others to use it too, roll it out in the form of a library
- How to use a particular library depends completely on the implementation of the library
- Except STL, all other C++ libraries are usually non-standard and hence need to be installed separately from g++

Library

- In computer science, a library is a collection of subroutines or classes used to develop software quickly and efficiently
- Usually libraries have functions/constructs which are used very frequently by programmers
- Examples : STL, Boost C++
- When you create a software, and want others to use it too, roll it out in the form of a library
- How to use a particular library depends completely on the implementation of the library
- Except STL, all other C++ libraries are usually non-standard and hence need to be installed separately from g++

Library

- In computer science, a library is a collection of subroutines or classes used to develop software quickly and efficiently
- Usually libraries have functions/constructs which are used very frequently by programmers
- Examples : STL, Boost C++
- When you create a software, and want others to use it too, roll it out in the form of a library
- How to use a particular library depends completely on the implementation of the library
- Except STL, all other C++ libraries are usually non-standard and hence need to be installed separately from g++

Library

- In computer science, a library is a collection of subroutines or classes used to develop software quickly and efficiently
- Usually libraries have functions/constructs which are used very frequently by programmers
- Examples : STL, Boost C++
- When you create a software, and want others to use it too, roll it out in the form of a library
- How to use a particular library depends completely on the implementation of the library
- Except STL, all other C++ libraries are usually non-standard and hence need to be installed separately from g++

Standard Template Library

- Simple repeatedly used procedures are needed to be coded by the programmer each time he wants to use them
- For eg : Sorting, searching, string handling, etc
- Some repeatedly used data structures are also needed to be developed from scratch each time
- For eg : Stack, List, Linked List, Queue, Set, Tree, Heap, Map...
- A secondary aim of C++ is also to enable the programmer to focus more on the larger picture, eliminating the details quickly
- Hence, C++ comes with pre-defined fast, generic, template based collection of classes which are also very efficient

Standard Template Library

- Simple repeatedly used procedures are needed to be coded by the programmer each time he wants to use them
- For eg : Sorting, searching, string handling, etc
- Some repeatedly used data structures are also needed to be developed from scratch each time
- For eg : Stack, List, Linked List, Queue, Set, Tree, Heap, Map...
- A secondary aim of C++ is also to enable the programmer to focus more on the larger picture, eliminating the details quickly
- Hence, C++ comes with pre-defined fast, generic, template based collection of classes which are also very efficient

Standard Template Library

- Simple repeatedly used procedures are needed to be coded by the programmer each time he wants to use them
- For eg : Sorting, searching, string handling, etc
- Some repeatedly used data structures are also needed to be developed from scratch each time
- For eg : Stack, List, Linked List, Queue, Set, Tree, Heap, Map...
- A secondary aim of C++ is also to enable the programmer to focus more on the larger picture, eliminating the details quickly
- Hence, C++ comes with pre-defined fast, generic, template based collection of classes which are also very efficient

Standard Template Library

- Simple repeatedly used procedures are needed to be coded by the programmer each time he wants to use them
- For eg : Sorting, searching, string handling, etc
- Some repeatedly used data structures are also needed to be developed from scratch each time
- For eg : Stack, List, Linked List, Queue, Set, Tree, Heap, Map...
- A secondary aim of C++ is also to enable the programmer to focus more on the larger picture, eliminating the details quickly
- Hence, C++ comes with pre-defined fast, generic, template based collection of classes which are also very efficient

Standard Template Library

- Simple repeatedly used procedures are needed to be coded by the programmer each time he wants to use them
- For eg : Sorting, searching, string handling, etc
- Some repeatedly used data structures are also needed to be developed from scratch each time
- For eg : Stack, List, Linked List, Queue, Set, Tree, Heap, Map...
- A secondary aim of C++ is also to enable the programmer to focus more on the larger picture, eliminating the details quickly
- Hence, C++ comes with pre-defined fast, generic, template based collection of classes which are also very efficient

Standard Template Library

- Simple repeatedly used procedures are needed to be coded by the programmer each time he wants to use them
- For eg : Sorting, searching, string handling, etc
- Some repeatedly used data structures are also needed to be developed from scratch each time
- For eg : Stack, List, Linked List, Queue, Set, Tree, Heap, Map...
- A secondary aim of C++ is also to enable the programmer to focus more on the larger picture, eliminating the details quickly
- Hence, C++ comes with pre-defined fast, generic, template based collection of classes which are also very efficient

Standard Template Library

- The Standard Template Library (STL) is a C++ software library
- The STL was created as the first library of generic algorithms and data structures for C++
- Idea behind STL: generic programming, abstractness without loss of efficiency, the Von Neumann computation model
- The STL achieves its results through the use of templates.
- Modern C++ compilers are optimized to minimize any abstraction penalty arising from heavy use of the STL.
- The STL provides a ready-made set of common classes for C++, that can be used with any built-in type and any user-defined type

Standard Template Library

- The Standard Template Library (STL) is a C++ software library
- The STL was created as the first library of generic algorithms and data structures for C++
- Idea behind STL: generic programming, abstractness without loss of efficiency, the Von Neumann computation model
- The STL achieves its results through the use of templates.
- Modern C++ compilers are optimized to minimize any abstraction penalty arising from heavy use of the STL.
- The STL provides a ready-made set of common classes for C++, that can be used with any built-in type and any user-defined type

Standard Template Library

- The Standard Template Library (STL) is a C++ software library
- The STL was created as the first library of generic algorithms and data structures for C++
- Idea behind STL: generic programming, abstractness without loss of efficiency, the Von Neumann computation model
- The STL achieves its results through the use of templates.
- Modern C++ compilers are optimized to minimize any abstraction penalty arising from heavy use of the STL.
- The STL provides a ready-made set of common classes for C++, that can be used with any built-in type and any user-defined type

Standard Template Library

- The Standard Template Library (STL) is a C++ software library
- The STL was created as the first library of generic algorithms and data structures for C++
- Idea behind STL: generic programming, abstractness without loss of efficiency, the Von Neumann computation model
- The STL achieves its results through the use of templates.
- Modern C++ compilers are optimized to minimize any abstraction penalty arising from heavy use of the STL.
- The STL provides a ready-made set of common classes for C++, that can be used with any built-in type and any user-defined type

Standard Template Library

- The Standard Template Library (STL) is a C++ software library
- The STL was created as the first library of generic algorithms and data structures for C++
- Idea behind STL: generic programming, abstractness without loss of efficiency, the Von Neumann computation model
- The STL achieves its results through the use of templates.
- Modern C++ compilers are optimized to minimize any abstraction penalty arising from heavy use of the STL.
- The STL provides a ready-made set of common classes for C++, that can be used with any built-in type and any user-defined type

Standard Template Library

- The Standard Template Library (STL) is a C++ software library
- The STL was created as the first library of generic algorithms and data structures for C++
- Idea behind STL: generic programming, abstractness without loss of efficiency, the Von Neumann computation model
- The STL achieves its results through the use of templates.
- Modern C++ compilers are optimized to minimize any abstraction penalty arising from heavy use of the STL.
- The STL provides a ready-made set of common classes for C++, that can be used with any built-in type and any user-defined type

Standard Template Library

- The Standard Template Library (STL) is composed of 4 parts:
  - Containers: Stack, Queue, List etc
  - Algorithms: Sort, Search, etc
  - Functors: Function Objects
  - Iterators: Random, Forward, etc

Standard Template Library

- The Standard Template Library (STL) is composed of 4 parts:
  - Containers: Stack, Queue, List etc
  - Algorithms: Sort, Search, etc
  - Functors: Function Objects
  - Iterators: Random, Forward, etc

Standard Template Library

- The Standard Template Library (STL) is composed of 4 parts:
  - Containers: Stack, Queue, List etc
  - Algorithms: Sort, Search, etc
  - Functors: Function Objects
  - Iterators: Random, Forward, etc

Standard Template Library

- The Standard Template Library (STL) is composed of 4 parts:
  - Containers: Stack, Queue, List etc
  - Algorithms: Sort, Search, etc
  - Functors: Function Objects
  - Iterators: Random, Forward, etc

Standard Template Library

- The Standard Template Library (STL) is composed of 4 parts:
  - Containers: Stack, Queue, List etc
  - Algorithms: Sort, Search, etc
  - Functors: Function Objects
  - Iterators: Random, Forward, etc

STL: Containers

Deque

- Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends
- Unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.
- Unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations.
- Thus deques do not allow direct access by offsetting pointers to elements like arrays or vectors.
- The elements of a deque can be scattered in different chunks of storage. Memory is allocated in chunks to avoid over scattering.
- Internally, more complex than vectors, but can be more efficient if the sequences are large

STL: Containers

Deque

- Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends
- Unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.
- Unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations.
- Thus deques do not allow direct access by offsetting pointers to elements like arrays or vectors.
- The elements of a deque can be scattered in different chunks of storage. Memory is allocated in chunks to avoid over scattering.
- Internally, more complex than vectors, but can be more efficient if the sequences are large

STL: Containers

Deque

- Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends
- Unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.
- Unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations.
- Thus deques do not allow direct access by offsetting pointers to elements like arrays or vectors.
- The elements of a deque can be scattered in different chunks of storage. Memory is allocated in chunks to avoid over scattering.
- Internally, more complex than vectors, but can be more efficient if the sequences are large

STL: Containers

Deque

- Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends
- Unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.
- Unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations.
- Thus deques do not allow direct access by offsetting pointers to elements like arrays or vectors.
- The elements of a deque can be scattered in different chunks of storage. Memory is allocated in chunks to avoid over scattering.
- Internally, more complex than vectors, but can be more efficient if the sequences are large

STL: Containers

Deque

- Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends
- Unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.
- Unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations.
- Thus deques do not allow direct access by offsetting pointers to elements like arrays or vectors.
- The elements of a deque can be scattered in different chunks of storage. Memory is allocated in chunks to avoid over scattering.
- Internally, more complex than vectors, but can be more efficient if the sequences are large

STL: Containers

Deque

- Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends
- Unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.
- Unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations.
- Thus deques do not allow direct access by offsetting pointers to elements like arrays or vectors.
- The elements of a deque can be scattered in different chunks of storage. Memory is allocated in chunks to avoid over scattering.
- Internally, more complex than vectors, but can be more efficient if the sequences are large

STL: Containers

List

- Sequence containers allowing constant time insert and erase operations within the sequence, and iteration in both directions.
- List containers are implemented as doubly-linked lists
- They are very similar to forward_list: The main difference being that forward_list objects are single-linked lists
- The main drawback of lists is that they lack direct access to the elements by their position
- Lists perform generally better in inserting, extracting and moving elements in any position

STL: Containers

List

- Sequence containers allowing constant time insert and erase operations within the sequence, and iteration in both directions.
- List containers are implemented as doubly-linked lists
- They are very similar to forward_list: The main difference being that forward_list objects are single-linked lists
- The main drawback of lists is that they lack direct access to the elements by their position
- Lists perform generally better in inserting, extracting and moving elements in any position

STL: Containers

List

- Sequence containers allowing constant time insert and erase operations within the sequence, and iteration in both directions.
- List containers are implemented as doubly-linked lists
- They are very similar to forward_list: The main difference being that forward_list objects are single-linked lists
- The main drawback of lists is that they lack direct access to the elements by their position
- Lists perform generally better in inserting, extracting and moving elements in any position

STL: Containers

List

- Sequence containers allowing constant time insert and erase operations within the sequence, and iteration in both directions.
- List containers are implemented as doubly-linked lists
- They are very similar to forward_list: The main difference being that forward_list objects are single-linked lists
- The main drawback of lists is that they lack direct access to the elements by their position
- Lists perform generally better in inserting, extracting and moving elements in any position

STL: Containers

List

- Sequence containers allowing constant time insert and erase operations within the sequence, and iteration in both directions.
- List containers are implemented as doubly-linked lists
- They are very similar to forward_list: The main difference being that forward_list objects are single-linked lists
- The main drawback of lists is that they lack direct access to the elements by their position
- Lists perform generally better in inserting, extracting and moving elements in any position

STL: Containers

Stack

- Implements the LIFO (Last In First Out) sequencing on the elements, with only one end for all insertion and extraction of data
- Needs 2 arguments : Type of data, and type of container
- Container is the type of the stack and should support the usual operations like push(), pop() etc
- Standard containers of vector, deque and list can be used. By default, the standard container deque is used.
- template < class T, class Container = deque<T> > class stack;

STL: Containers

Stack

- Implements the LIFO (Last In First Out) sequencing on the elements, with only one end for all insertion and extraction of data
- Needs 2 arguments : Type of data, and type of container
- Container is the type of the stack and should support the usual operations like push(), pop() etc
- Standard containers of vector, deque and list can be used. By default, the standard container deque is used.
- template < class T, class Container = deque<T> > class stack;

STL: Containers

Stack

- Implements the LIFO (Last In First Out) sequencing on the elements, with only one end for all insertion and extraction of data
- Needs 2 arguments : Type of data, and type of container
- Container is the type of the stack and should support the usual operations like push(), pop() etc
- Standard containers of vector, deque and list can be used. By default, the standard container deque is used.
- template < class T, class Container = deque<T> > class stack;

STL: Containers

Stack

- Implements the LIFO (Last In First Out) sequencing on the elements, with only one end for all insertion and extraction of data
- Needs 2 arguments : Type of data, and type of container
- Container is the type of the stack and should support the usual operations like push(), pop() etc
- Standard containers of vector, deque and list can be used. By default, the standard container deque is used.
- template < class T, class Container = deque<T> > class stack;

STL: Containers

Stack

- Implements the LIFO (Last In First Out) sequencing on the elements, with only one end for all insertion and extraction of data
- Needs 2 arguments : Type of data, and type of container
- Container is the type of the stack and should support the usual operations like push(), pop() etc
- Standard containers of vector, deque and list can be used. By default, the standard container deque is used.
- template < class T, class Container = deque<T> > class stack;

STL: Containers

Maps

- Maps are associative containers that store elements formed by a combination of a (key, value), following a specific order.
- In a map, the key values are generally used to sort and uniquely identify the elements
- While the mapped values store the content associated to this key.
- A unique feature of maps is that they implement the operator[], which allows for direct access of the mapped value.
- Maps are typically implemented as binary search trees

STL: Containers

Maps

- Maps are associative containers that store elements formed by a combination of a (key, value), following a specific order.
- In a map, the key values are generally used to sort and uniquely identify the elements
- While the mapped values store the content associated to this key.
- A unique feature of maps is that they implement the operator[], which allows for direct access of the mapped value.
- Maps are typically implemented as binary search trees

STL: Containers

Maps

- Maps are associative containers that store elements formed by a combination of a (key, value), following a specific order.
- In a map, the key values are generally used to sort and uniquely identify the elements
- While the mapped values store the content associated to this key.
- A unique feature of maps is that they implement the operator[], which allows for direct access of the mapped value.
- Maps are typically implemented as binary search trees

STL: Containers

Maps

- Maps are associative containers that store elements formed by a combination of a (key, value), following a specific order.
- In a map, the key values are generally used to sort and uniquely identify the elements
- While the mapped values store the content associated to this key.
- A unique feature of maps is that they implement the operator[], which allows for direct access of the mapped value.
- Maps are typically implemented as binary search trees

STL: Containers

Maps

- Maps are associative containers that store elements formed by a combination of a (key, value), following a specific order.
- In a map, the key values are generally used to sort and uniquely identify the elements
- While the mapped values store the content associated to this key.
- A unique feature of maps is that they implement the operator[], which allows for direct access of the mapped value.
- Maps are typically implemented as binary search trees