ECE 220

Lecture x0016 - 04/11

Templates & iterators

Recap

Recap

- References vs. pointers

Recap

- References vs. pointers
- Classes vs. structs

Recap

- References vs. pointers
- Classes vs. structs
  - Friend functions

Recap

- References vs. pointers
- Classes vs. structs
  - Friend functions
- Inheritance (private/public/protected)

Recap

- References vs. pointers
- Classes vs. structs
    - Friend functions
- Inheritance (private/public/protected)
- Constructor in derived classes

Recap

- References vs. pointers
- Classes vs. structs
    - Friend functions
- Inheritance (private/public/protected)
- Constructor in derived classes
- Virtual functions

Recap

- References vs. pointers
- Classes vs. structs
    - Friend functions
- Inheritance (private/public/protected)
- Constructor in derived classes
- Virtual functions
- Pure virtual functions / abstract classes

Recap

- References vs. pointers
- Classes vs. structs
    - Friend functions
- Inheritance (private/public/protected)
- Constructor in derived classes
- Virtual functions
- Pure virtual functions / abstract classes
- Examples

Recap: virtual functions

Recap: virtual functions

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

class Animal{
public:
  void eat(){
    cout << "I'm eating generic food." << endl;
  }
};

class Cat : public Animal{
public:
  void eat(){
    cout << "I'm eating a mouse." << endl;
  }
};

void eat_lunch(Animal *a){
  a->eat();
}
```

Recap: virtual functions

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

class Animal{
public:
  void eat(){
    cout << "I'm eating generic food." << endl;
  }
};

class Cat : public Animal{
public:
  void eat(){
    cout << "I'm eating a mouse." << endl;
  }
};

void eat_lunch(Animal *a){
  a->eat();
}
```

```cpp
int main(){
  Animal *anim = new Animal();
  Cat *bruno = new Cat();
  anim->eat();
  bruno->eat();

  eat_lunch(anim);
  eat_lunch(bruno);
}
```

Recap: virtual functions

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

class Animal{
public:
  void eat(){
    cout << "I'm eating generic food." << endl;
  }
};

class Cat : public Animal{
public:
  void eat(){
    cout << "I'm eating a mouse." << endl;
  }
};

void eat_lunch(Animal *a){
  a->eat();
}
```

```
int main(){
  Animal *anim = new Animal();
  Cat *bruno = new Cat();
  anim->eat();
  bruno->eat();

  eat_lunch(anim);
  eat_lunch(bruno);
}
```

Why didn't Bruno eat a mouse for lunch ?

Recap: virtual functions

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

class Animal{
public:
  void eat(){
    cout << "I'm eating generic food." << endl;
  }
};

class Cat : public Animal{
public:
  void eat(){
    cout << "I'm eating a mouse." << endl;
  }
};

void eat_lunch(Animal *a){
  a->eat();
}
```

```
int main(){
  Animal *anim = new Animal();
  Cat *bruno = new Cat();
  anim->eat();
  bruno->eat();

  eat_lunch(anim);
  eat_lunch(bruno);
}
```

Why didn't Bruno eat a mouse for lunch ?

Need a way for the derived class to ***override*** the base class function,

… or ….

We will have to *overload* **`eat_lunch`** for each new species!

Recap: virtual functions

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

class Animal{
public:
  virtual void eat(){
    cout << "I'm eating generic food." << endl;
  }
};

class Cat : public Animal{
public:
  void eat(){
    cout << "I'm eating a mouse." << endl;
  }
};

void eat_lunch(Animal *a){
  a->eat();
}
```

Recap: virtual functions

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

class Animal{
public:
  virtual void eat(){
    cout << "I'm eating generic food." << endl;
  }
};

class Cat : public Animal{
public:
  void eat(){
    cout << "I'm eating a mouse." << endl;
  }
};

void eat_lunch(Animal *a){
  a->eat();
}
```

- A virtual function is a member function in the base class that we expect to redefine in derived classes

Recap: virtual functions

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

class Animal{
public:
  virtual void eat(){
    cout << "I'm eating generic food." << endl;
  }
};

class Cat : public Animal{
public:
  void eat(){
    cout << "I'm eating a mouse." << endl;
  }
};

void eat_lunch(Animal *a){
  a->eat();
}
```

- A virtual function is a member function in the base class that we expect to redefine in derived classes
- What if your colleagues forget to override a virtual function? How to ***ensure*** it?

Recap: pure virtual functions

Recap: pure virtual functions

Pure virtual functions are used

Recap: pure virtual functions

Pure virtual functions are used

- if a function doesn't have any use in the base class

Recap: pure virtual functions

Pure virtual functions are used

- if a function doesn't have any use in the base class

- but the function must be implemented by all its derived classes

Recap: pure virtual functions

Pure virtual functions are used

- if a function doesn't have any use in the base class
- but the function must be implemented by all its derived classes

A pure virtual function doesn't have a function body and it ends with “`=0`”

Recap: pure virtual functions

Pure virtual functions are used

- if a function doesn't have any use in the base class
- but the function must be implemented by all its derived classes

A pure virtual function doesn't have a function body and it ends with "=0"

```
class Animal{
public:
  virtual void eat()=0;
};

class Cat : public Animal{
public:
  void eat(){
    cout << "I'm eating a mouse." << endl;
  }
};
```

Recap: pure virtual functions

Pure virtual functions are used

- if a function doesn't have any use in the base class
- but the function must be implemented by all its derived classes

A pure virtual function doesn't have a function body and it ends with "=0"

```
class Animal{
public:
  virtual void eat()=0;
};

class Cat : public Animal{
public:
  void eat(){
    cout << "I'm eating a mouse." << endl;
  }
};
```

Adding a pure virtual function turns a normal class to an ***abstract*** class!

Recap: abstract class

- **Abstract class** is a class that contains one or more ***pure virtual functions.***

Recap: abstract class

- **Abstract class** is a class that contains one or more ***pure virtual functions.***
  - No objects of an abstract class can be created!

Recap: abstract class

- **Abstract class** is a class that contains one or more ***pure virtual functions.***
  - No objects of an abstract class can be created!
  - A pure virtual function that is not implemented in a derived class remains a pure virtual function, so the *derived class is also an abstract class!*

Recap: abstract class

- **Abstract class** is a class that contains one or more ***pure virtual functions.***
  - No objects of an abstract class can be created!
  - A pure virtual function that is not implemented in a derived class remains a pure virtual function, so the *derived class is also an abstract class!*
  - An abstract class is intended as an interface to objects accessed through pointers and references (e.g. `eat_lunch` function)

Copy constructor

Copy constructor

- Last time we implemented a linked list using the `Person` class and `LinkedList` class.

Copy constructor

- Last time we implemented a linked list using the `Person` class and `LinkedList` class.

- Now recall that we could implement a *Stack ADT* with a linked list

Copy constructor

- Last time we implemented a linked list using the `Person` class and `LinkedList` class.
- Now recall that we could implement a *Stack ADT* with a linked list
    - Push: add at head of linked list

Copy constructor

- Last time we implemented a linked list using the `Person` class and `LinkedList` class.
- Now recall that we could implement a *Stack ADT* with a linked list
  - Push: add at head of linked list
  - Pop: remove from head + *give popped value to caller*

Copy constructor

- Last time we implemented a linked list using the `Person` class and `LinkedList` class.

- Now recall that we could implement a *Stack ADT* with a linked list

    - Push: add at head of linked list

    - Pop: remove from head + *give popped value to caller*

    - How can we do the second part?

Copy constructor

- Last time we implemented a linked list using the `Person` class and `LinkedList` class.

- Now recall that we could implement a *Stack ADT* with a linked list

  - Push: add at head of linked list

  - Pop: remove from head + *give popped value to caller*

  - How can we do the second part?

Need a constructor that can generate a new instance of the object from a given instance, i.e. a copy constructor.

Copy constructor

Copy constructor

```
class Person{
  const char *name;
  unsigned int byear;

public:
  Person *next;
  Person(const char *name, unsigned int byear);
  Person(const Person &p);
};

Person::Person(const Person &p){
    this->name = p.name;
    this->byear = p.byear;
    this->next = NULL;
}
```

Copy constructor

```
class Person{
  const char *name;
  unsigned int byear;

public:
  Person *next;
  Person(const char *name, unsigned int byear);
  Person(const Person &p);
};

Person::Person(const Person &p){
    this->name = p.name;
    this->byear = p.byear;
    this->next = NULL;
}
```

Second constructor useful to copy an instance of Person.

Copy constructor

```
class Person{
  const char *name;
  unsigned int byear;

public:
  Person *next;
  Person(const char *name, unsigned int byear);
  Person(const Person &p);
};

Person::Person(const Person &p){
    this->name = p.name;
    this->byear = p.byear;
    this->next = NULL;
}
```

Second constructor useful to copy an instance of Person.

Called pass by constant reference.

Copy constructor

```
class Person{
  const char *name;
  unsigned int byear;

public:
  Person *next;
  Person(const char *name, unsigned int byear);
  Person(const Person &p);
};

Person::Person(const Person &p){
    this->name = p.name;
    this->byear = p.byear;
    this->next = NULL;
}
```

Second constructor useful to copy an instance of Person.

Called pass by constant reference.

- **Exercise:** Can we appropriately modify the `LinkedList` class definition and create a derived Stack class from it?

Copy constructor

```
class Person{
  const char *name;
  unsigned int byear;

public:
  Person *next;
  Person(const char *name, unsigned int byear);
  Person(const Person &p);
};

Person::Person(const Person &p){
    this->name = p.name;
    this->byear = p.byear;
    this->next = NULL;
}
```

Second constructor useful to copy an instance of Person.

Called pass by constant reference.

- **Exercise:** Can we appropriately modify the `LinkedList` class definition and create a derived Stack class from it?
- Stack should ***only*** expose the push and pop functions.

Exercise

Exercise

- How to modify the `LinkedList` class?

Exercise

- How to modify the `LinkedList` class?
    - Does `add_at_head` and `del_at_head` need to be public?

Exercise

- How to modify the `LinkedList` class?
  - Does `add_at_head` and `del_at_head` need to be public?
    - Can they be private?

Exercise

- How to modify the `LinkedList` class?
  - Does `add_at_head` and `del_at_head` need to be public?
    - Can they be private?
  - When popping, we need access to head pointer to call copy constructor - can it still be private?

Recall our swap function?

Recall our swap function?

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}
```

Recall our swap function?

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

void swap(float &a, float &b){
    float temp = a;
    a = b;
    b = temp;
}
```

- Okay, what if you want to swap two `float`s?

Recall our swap function?

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

void swap(float &a, float &b){
    float temp = a;
    a = b;
    b = temp;
}

void swap(char &a, char &b){
    char temp = a;
    a = b;
    b = temp;
}
```

- Okay, what if you want to swap two `float`s?
- How about `char`s?

Recall our swap function?

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

void swap(float &a, float &b){
    float temp = a;
    a = b;
    b = temp;
}

void swap(char &a, char &b){
    char temp = a;
    a = b;
    b = temp;
}
```

- Okay, what if you want to swap two `float`s?
- How about `char`s?
- Cool, how about two `Person`s?

Recall our swap function?

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

void swap(float &a, float &b){
    float temp = a;
    a = b;
    b = temp;
}

void swap(char &a, char &b){
    char temp = a;
    a = b;
    b = temp;
}
```

- Okay, what if you want to swap two `float`s?
- How about `char`s?
- Cool, how about two `Person`s?

```
class Person{
  const char *name;
  unsigned int byear;
  …
  …
};
```

Recall our swap function?

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

void swap(float &a, float &b){
    float temp = a;
    a = b;
    b = temp;
}

void swap(char &a, char &b){
    char temp = a;
    a = b;
    b = temp;
}
```

- Okay, what if you want to swap two `float`s?
- How about `char`s?
- Cool, how about two `Person`s?

```
class Person{
  const char *name;
  unsigned int byear;
  …
  …
};
```

Are we doomed to keep writing swaps?

Enter C++ templates

Enter C++ templates

- A template is a blueprint for creating a ***generic*** function or a class.

Enter C++ templates

- A template is a blueprint for creating a ***generic*** function or a class.
  - A mechanism to allow us to write code once with a *dummy type* (called a template) and then cast to the right kind when needed.

Enter C++ templates

- A template is a blueprint for creating a ***generic*** function or a class.
  - A mechanism to allow us to write code once with a *dummy type* (called a template) and then cast to the right kind when needed.

```
int Add(int a, int b){
  return a+b;
}

double Add(double a, double b){
  return a+b;
}
```

Enter C++ templates

- A template is a blueprint for creating a ***generic*** function or a class.
    - A mechanism to allow us to write code once with a *dummy type* (called a template) and then cast to the right kind when needed.

```
int Add(int a, int b){
  return a+b;
}

double Add(double a, double b){
  return a+b;
}
```

→

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

Example

Example

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

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

int main(){
  cout<<Add(1, 3)<<endl;
  cout<<Add(1.2, 3.5)<<endl;

}
```

Example

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

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

int main(){
  cout<<Add(1, 3)<<endl;
  cout<<Add(1.2, 3.5)<<endl;

}
```

Well … what if we want to be able to add 2 to ‘C’ and get “E”?

Example

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

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

int main(){
  cout<<Add(1, 3)<<endl;
  cout<<Add(1.2, 3.5)<<endl;

}
```

Well … what if we want to be able to add 2 to ‘C’ and get “E”?

You can specify more than one *typename.*

Example

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

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

int main(){
  cout<<Add(1, 3)<<endl;
  cout<<Add(1.2, 3.5)<<endl;

}
```

Well … what if we want to be able to add 2 to ‘C’ and get “E”?

You can specify more than one *typename.*

```
template <typename T1, typename T2>
T2 Add(T1 a, T2 b){
  return a+b;
}
```

Example

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

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

int main(){
  cout<<Add(1, 3)<<endl;
  cout<<Add(1.2, 3.5)<<endl;
  cout<<Add(2, ‘C’)<<endl;
}
```

Well … what if we want to be able to add 2 to ‘C’ and get “E”?

You can specify more than one *typename.*

```
template <typename T1, typename T2>
T2 Add(T1 a, T2 b){
  return a+b;
}
```

Exercise

Implement `myswap` so it works for any type of argument. Then use it to swap two instances of `Person`.

Note: It cannot be named swap, that will conflict with a templated `swap` function in the standard library.

Exercise

Implement `myswap` so it works for any type of argument. Then use it to swap two instances of `Person`.

```
class Person{
  const char *name;
  unsigned int byear;

public:
  Person *next;
  Person(const char *name, unsigned int byear);
  Person(const Person &p);
};

Person::Person(const Person &p){
    this->name = p.name;
    this->byear = p.byear;
    this->next = NULL;
}
```

Note: It cannot be named swap, that will conflict with a templated `swap` function in the standard library.

Class templates

Class templates

- Just like we can have function templates, we can also have class template.

Class templates

- Just like we can have function templates, we can also have class template.
- Here is a generic `node`.

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

template <typename T>
class Node{
  T data;
public:
  Node<T> * next;
  Node(T inval){
      data = inval;
      next = NULL;
  }
  void print(){ cout<<data; }
};
```

Class templates

- Just like we can have function templates, we can also have class template.
- Here is a generic `node`.
- Implement a linked list on this and test with `chars` and `ints`

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

template <typename T>
class Node{
  T data;
public:
  Node<T> * next;
  Node(T inval){
      data = inval;
      next = NULL;
  }
  void print(){ cout<<data; }
};
```

Class templates

- Just like we can have function templates, we can also have class template.
- Here is a generic `node`.
- Implement a linked list on this and test with `chars` and `ints`

```
template <class H>
class LinkedList{
H *head;

public:
  LinkedList(){
    this->head = NULL;
  }
  void print_list();
  void add_at_head(H &p);
  void del_at_head();
  ~LinkedList();
};
```

What would you need to make this work with our `Person` class?

C++ STL: Standard Template Library

C++ STL: Standard Template Library

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as *lists, stacks, arrays, etc.*

C++ STL: Standard Template Library

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as *lists, stacks, arrays, etc.*

- STL has five components

C++ STL: Standard Template Library

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as *lists, stacks, arrays, etc.*

- STL has five components
  - Algorithms

C++ STL: Standard Template Library

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as *lists, stacks, arrays, etc.*

- STL has five components
  - Algorithms
  - Containers

C++ STL: Standard Template Library

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as *lists, stacks, arrays, etc.*

- STL has five components
  - Algorithms
  - Containers
  - Iterators

C++ STL: Standard Template Library

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as *lists, stacks, arrays, etc.*

- STL has five components
  - Algorithms
  - Containers
  - Iterators
  - Functors

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

C++ STL: Standard Template Library

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as *lists, stacks, arrays, etc.*

- STL has five components
  - Algorithms
  - Containers
  - Iterators
  - Functors
  - Adaptors

C++ STL: Standard Template Library

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as *lists, stacks, arrays, etc.*

- STL has five components
  - Algorithms
  - Containers
  - Iterators
  - Functors
  - Adaptors

Left for later classes ←

Algorithms

Algorithms

- STL contains standard and vetted implementations of algorithms for sorting, searching, partitioning, etc.

Algorithms

- STL contains standard and vetted implementations of algorithms for sorting, searching, partitioning, etc.

```
#include <algorithm>
#include <iostream>

using namespace std;

void show(int a[], int array_size){
  int i=0;
  for (i = 0; i < array_size-1; ++i)
    cout << a[i] << ", ";
  cout<<a[i]<<endl;
}
```

Algorithms

- STL contains standard and vetted implementations of algorithms for sorting, searching, partitioning, etc.

```cpp
#include <algorithm>
#include <iostream>

using namespace std;

void show(int a[], int array_size){
  int i=0;
  for (i = 0; i < array_size-1; ++i)
    cout << a[i] << ", ";
  cout<<a[i]<<endl;
}
```

```cpp
int main(){
    int a[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };
    int asize = sizeof(a) / sizeof(a[0]);
    cout << "The array before sorting is: \n";
    show(a, asize);

    sort(a, a + asize);
    cout << "\n\nThe array after sorting is:\n";
    show(a, asize);
    return 0;
}
```

Containers

Containers

- Vectors

Containers

- Vectors
  - Dynamically sized but also contiguously stored

Containers

- Vectors
  - Dynamically sized but also contiguously stored
  - Fast traversal

Containers

- Vectors
  - Dynamically sized but also contiguously stored
  - Fast traversal
  - Insertion at beginning expensive, end … *variable*

Containers

- Vectors
  - Dynamically sized but also contiguously stored
  - Fast traversal
  - Insertion at beginning expensive, end … *variable*
- Lists

Containers

- Vectors
  - Dynamically sized but also contiguously stored
  - Fast traversal
  - Insertion at beginning expensive, end … *variable*
- Lists
  - Doubly linked lists

Containers

- Vectors
  - Dynamically sized but also contiguously stored
  - Fast traversal
  - Insertion at beginning expensive, end … *variable*
- Lists
  - Doubly linked lists
  - Non-contiguously stored

Containers

- Vectors
  - Dynamically sized but also contiguously stored
  - Fast traversal
  - Insertion at beginning expensive, end … *variable*
- Lists
  - Doubly linked lists
  - Non-contiguously stored
  - Slower traversal

Containers

- Vectors
  - Dynamically sized but also contiguously stored
  - Fast traversal
  - Insertion at beginning expensive, end … *variable*

- Lists
  - Doubly linked lists
  - Non-contiguously stored
  - Slower traversal
  - Insertion/deletion constant time

Containers

- Vectors
  - Dynamically sized but also contiguously stored
  - Fast traversal
  - Insertion at beginning expensive, end … *variable*
- Lists
  - Doubly linked lists
  - Non-contiguously stored
  - Slower traversal
  - Insertion/deletion constant time

There are many more, but we will talk about these two and deal with rest on need-to-know basis.

Vectors - common operations

- `push_back` – It push the elements into a vector from the back
- `pop_back` – It is used to pop or remove elements from a vector from the back.
- `insert` – It inserts new elements before the element at the specified position
- `assign` – It assigns new value to the vector elements by replacing old ones
- `swap` – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
- `clear` – It is used to remove all the elements of the vector container
- `front` – Returns a reference to the first element in the vector
- `back` – Returns a reference to the last element in the vector
- `size` – Returns size of the container

Example

Example

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size: " << g1.size() <<endl;
    cout << "Elements: ";

    for (int i = 0; i < 5; i++)
        cout<<g1[i]<<" ";

    cout<<endl;
    return 0;
}
```

Example

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size: " << g1.size() <<endl;
    cout << "Elements: ";

    for (int i = 0; i < 5; i++)
        cout<<g1[i]<<" ";

    cout<<endl;
    return 0;
}
```

Example

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size: " << g1.size() <<endl;
    cout << "Elements: ";

    for (int i = 0; i < 5; i++)
        cout<<g1[i]<<" ";

    cout<<endl;
    return 0;
}
```

This is traditionally how we have been taught to iterate over an array.

Example

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size: " << g1.size() <<endl;
    cout << "Elements: ";

    for (int i = 0; i < 5; i++)
        cout<<g1[i]<<" ";

    cout<<endl;
    return 0;
}
```

This is traditionally how we have been taught to iterate over an array.

But there are many containers in STL: `vector`, `list`, `queue`, `map`, `set`, etc.

Example

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size: " << g1.size() <<endl;
    cout << "Elements: ";

    for (int i = 0; i < 5; i++)
        cout<<g1[i]<<" ";

    cout<<endl;
    return 0;
}
```

This is traditionally how we have been taught to iterate over an array.

But there are many containers in STL: `vector`, `list`, `queue`, `map`, `set`, etc.

Need a consistent way to iterate over containers regardless of functionality!

Iterators - common methods

Iterators - common methods

- `begin()` – Used to return the beginning position of the container.

Iterators - common methods

- `begin()` – Used to return the beginning position of the container.
- `end()` – Used to return the position after the end of the container.

Iterators - common methods

- `begin()` – Used to return the beginning position of the container.
- `end()` – Used to return the position after the end of the container.
- `advance(itr, num)` – Used to increment the iterator itr position till the specified number num.

Iterators - common methods

- `begin()` – Used to return the beginning position of the container.
- `end()` – Used to return the position after the end of the container.
- `advance(itr, num)` – Used to increment the iterator itr position till the specified number num.
- `next(itr, num), prev(itr, num)` - Used to return **new** *iterators* after incrementing or decrementing itr by num positions.

Iterators

- Iterators point to the address of elements of a container.

Iterators

- Iterators point to the address of elements of a container.

```
#include<iostream>
#include<iterator> // for iterators
#include<vector>   // for vectors

using namespace std;

int main() {
    vector<int> ar = { 1, 2, 3, 4, 5 };
    vector<int>::iterator ptr;     // Declaring iterator to a vector

    cout << "The vector elements are : ";
    for (ptr = ar.begin(); ptr < ar.end(); ptr++)
        cout << *ptr << " ";

    return 0;
}
```

Vectors - More operations

- `begin()` – Returns an **iterator** pointing to the first element in the vector
- `end()` – Returns an **iterator** pointing to the theoretical element after last
- `rbegin()` – Returns a reverse **iterator** pointing to the last element in the vector
- `rend()` – Returns a reverse **iterator** pointing to the theoretical element before the first
- `cbegin()` – Returns a ***constant*** iterator pointing to the first element in the vector.
- `cend()` – Returns a ***constant*** iterator pointing to the element after last
- `crbegin()` – Returns a ***constant*** reverse iterator pointing to the last element in the vector
- `crend()` – Returns a ***constant*** reverse iterator pointing to the theoretical element before the first

Lists - common operations

- `front()` – Returns the value of the first element in the list.
- `back()` – Returns the value of the last element in the list.
- `push_front()` – Adds a new element at the beginning of the list.
- `push_back()` – Adds a new element at the end of the list.
- `pop_front()` – Removes the first element of the list
- `pop_back()` – Removes the last element of the list
- `insert()` – Inserts new elements in the list before the element at a specified position.
- `size()` – Returns the number of elements in the list.

Example

Example

```
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

template <typename T>
void showlist(list<T> g){
    typename list<T>::iterator it;
    for (auto it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << endl;
}
```

Example

```
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

template <typename T>
void showlist(list<T> g){
    typename list<T>::iterator it;
    for (auto it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << endl;
}
```

New keyword introduced in C++11, allows compiler to *deduce* the type.

Example

```
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

template <typename T>
void showlist(list<T> g){
    typename list<T>::iterator it;
    for (auto it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << endl;
}
```

New keyword introduced in C++11, allows compiler to *deduce* the type.

```
int main(){

    list<int> gqlist1, gqlist2;

    for (int i = 0; i < 10; ++i) {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }

    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);

    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);

    cout << "\ngqlist2.sort(): ";
    gqlist2.sort();
    showlist(gqlist2);

    return 0;
}
```