Week 10 Lecture 1

NWEN 241
Systems Programming

Jyoti Sahni

jyoti.sahni@ecs.vuw.ac.nz

Announcement

- Weekly revision questions available at the course wiki at:

https://ecs.wgtn.ac.nz/Courses/NWEN241_2022T1/LectureSchedule

Content

Inheritance

Containers

(Topics relevant to Assignment #4)

Inheritance - Extending Classes

- Just like Java, C++ supports class **inheritance**
- **Sub Class or Derived class** – a class that inherits member fields from another class
- **Super Class or Base Class** – a class whose fields are inherited by sub class
- **The sub class is said to extend the base class**

Inheritance - Extending Classes

- Syntax of extending a single base class:

```
class subclass_name : access_mode baseclass_name {
      class_member_list
};
```

  - *subclass_name* is the identifier given to the sub class being declared
  - *access_mode* controls the access of inherited fields
  - *baseclass_name* is the identifier of the super class being extended

Example

Base Class:

```
class Animal {
public:
  const char *getName() const;
  void sleep();
  void eat(int food);
  Animal();
  Animal(const char *);

protected:
  int age;

private:
  char name[100]; };
```

Sub Class:

```
class Dog : public Animal {
public:
  int bark(int loudness);
  int bite(int strength);
  void run(int speed);
  void eat(int food);
  Dog(const char *n) : Animal(n) {}

private:
  int skills;
};
```

Recap: Member Access Specifier

- **Private members** – can only be accessed by member functions (and friends) and not accessible by descendant classes
- **Public members** – can be accessed outside the class and inherited by descendant classes
- **Protected members** – can only be accessed by member functions (and friends) and inherited by descendant classes
- When member access specifier is not indicated, default access is `private`

Example

```
class Dog : public Animal {

public:
  int bark(int loudness);
  int bite(int strength);
  void run(int speed);
  void eat(int food);
  Dog(const char *n) : Animal(n) {}

private:
  int skills;
};
```

Access mode – controls access to inherited fields

Base class member access specifier	Access mode		
	public	Protected	private
public	public	protected	private
protected	protected	protected	private
private	(Not accessible)		

If the access mode is not used, then it is private by default, in case of class and public in case of structures.

Example

```
class Dog : public Animal {
public:
  int bark(int loudness);
  int bite(int strength);
  void run(int speed);

  void eat(int food);

  Dog(const char *n) : Animal(n) {}

  private:
  int skills;

};
```

Member functions specific to sub class Dog

Member function present in base class - will be **overridden** by sub class

Member variable specific to sub class Dog

Example

```
class Dog : public Animal {
public:

  int bark(int loudness);
  int bite(int strength);
  void run(int speed);
  void eat(int food);

  Dog(const char *n) : Animal(n) {}

private:
  int skills;
};
```

Constructor of Dog invokes appropriate super class constructor

Unlike Java, C++ does not have super keyword for invoking super class constructor

When an object of derived class is instantiated, first the *base portion* of Derived is constructed (using the Base class constructor). Once the Base portion is finished, the Derived portion is constructed (using the Derived class constructor).

Overriding Member Functions

- Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class.

- It is like creating a new version of an old function, in the child class.

```
class Dog : public Animal {
public:
  …
  // Overridden function
  void eat(int food){

  }
};
```

```
Dog d;
d.eat(10);
```

Invokes function definition in child class

Prototype same as base class function.
Overridden member function.

Overriding Member Functions

How to invoke base class function:

```
class Dog : public Animal {
public:
  …
  // Overridden function
  void eat(int food);
  …
};
```

```
Animal::eat(10);
```

Within member
Function of derived class

```
Dog d;
d.Animal::eat(10);
```

From an instance

Pointers and Inheritance

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

};

class Dog : public Animal {
public:
    void display()
  {
  cout << "display Dog" << endl;
  }
};
```

Valid, Dog *is an* Animal

```
int main (){

   Animal* bptr;
   Dog d;

   bptr = &d;

  // Binded at compile time
   bptr->display();
}
```

Output:
display Animal

Rum time Polymorphism

- Allow a member function to be called **based on the content** of the base class pointer rather than its **type**.
- Implemented using Virtual functions

```
class Animal {
public:
    virtual void display()
  {
   cout << "display Animal" << endl;   }
  };
class Dog : public Animal {
public:
    void display()
  {
  cout << "display Dog" << endl;   }
};
```

```
int main (){
   Animal* bptr;
   Dog d;

   bptr = &d;

  // Binded at run time
   bptr->display();  }
```

```
Output:
display Dog
```

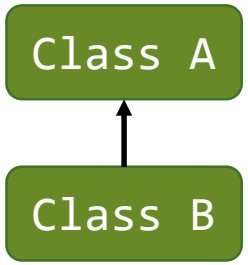
Pure virtual function and Abstract Classes

- **Pure virtual function** is declared in the base class without any implementation.
- Pure virtual functions must be implemented by a sub class that needs to be instantiated (**concrete).**
- A class that contains at least one **pure virtual function** member is called as an **Abstract class**.
- Abstract classes cannot be instantiated

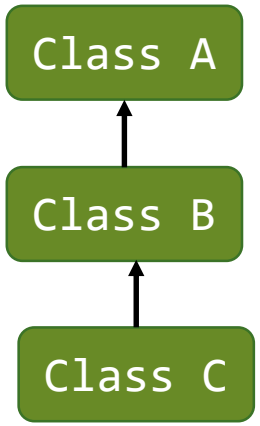
```
class Shape {
public:
  // Pure virtual function
  virtual float draw() = 0;

  // Virtual function
  virtual int getSides() {
    return 1;
  }
};
```

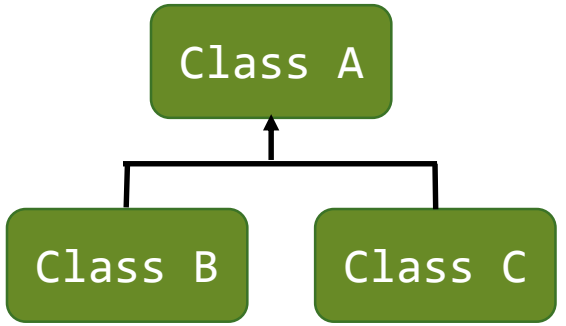
Types of Inheritance



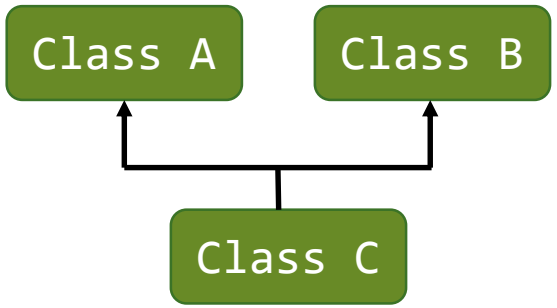
Single



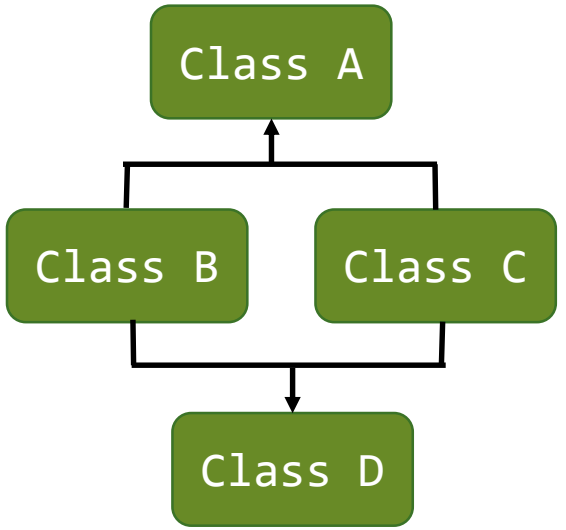
Multi-level



Hierarchical



Multiple



Hybrid

Multiple Inheritance

- C++ supports multiple inheritance

- Syntax for extending multiple classes:

```
class subclass_name : access_mode1 baseclass_name1, … ,
  access_modeN baseclass_nameN {
      class_member_list
};
```

Example

- Suppose that Human and Dog are existing classes

```
class Werewolf : public Human, public Dog {
public:
  void transform();
  …
  Werewolf(const char *n) : Human(n), Dog(n) {}

private:
  int transformCount;
  …
};
```

Member Function Clash

- What happens if base classes of a derived class have a common member function?

```
class A : {
public:
  void foo();
  …
};

class B : {
public:
  void foo();
  …
};
```

```
class C : public A, public B {
  …
};
```

Class C must override foo(), example:

```
class C : public A, public B {
public:
  void foo() {
    A::foo(); // Use A's implementation of foo!
  }
};
```

Containers

Generic Programming

- Generic programming involves writing code in a way that is **independent** of any particular type.

- It allows ***type*** as a parameter to methods and classes.

- A *type* parameter may be a primitive / built-in type such as `int` or `double` or a user defined type such as `class` or `structure.`

- Generics eliminates the need to create different algorithms if the data type is an integer, string or a character.

- Generics can be implemented in C++ using **Templates**. Templates allow us to create a single **function** or a **class** to work with different data types.

Standard Template Library

- Standard Template Library (STL) is a library for C++.

STL has with three basic components:

Containers

- Containers are used to manage collections of objects of a certain kind

Algorithms

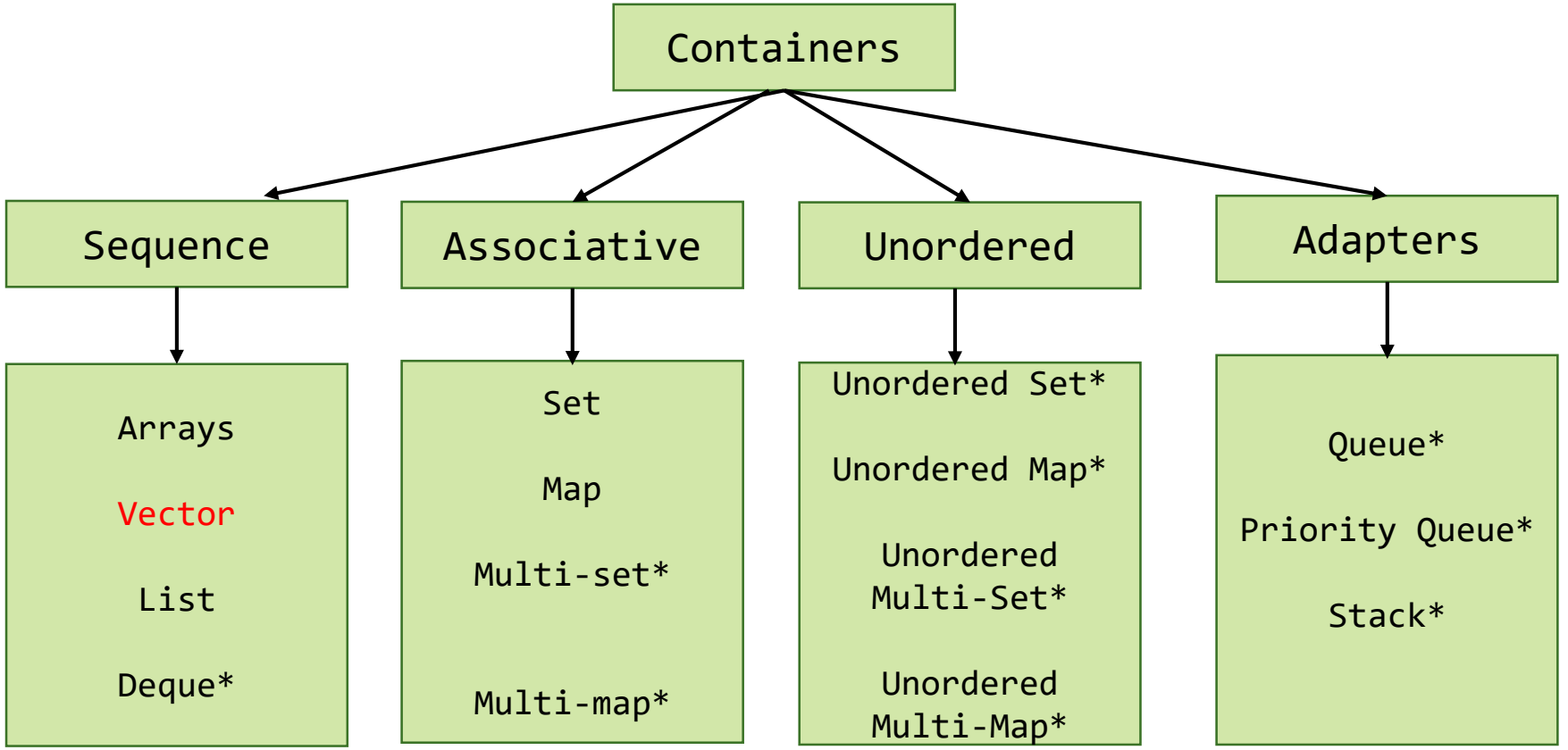
- Algorithms act on containers.
- Independent of containers

Iterators

- Generalized pointers that facilitate use of containers
- Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers

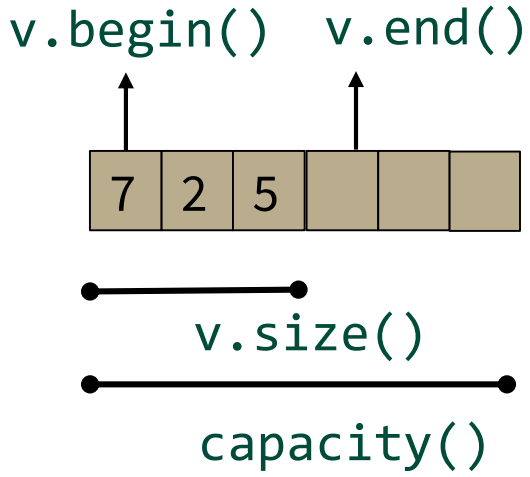
Containers

- Containers or container classes store objects and data.



Vector

- One of the containers is **Vector.**
- Defined in `<vector>`
- Same as **dynamic arrays** with the ability to resize itself **automatically** when an element is inserted or deleted, with their storage being handled automatically by the container.



- Vector elements are placed in contiguous storage.
- The **capacity** of the vector is decided by the compiler. It is generally bigger than the **size** of elements in it.
- This gives the ability to quickly insert an element to the end or remove the last one, just by keeping track of the number of elements.
- Vectors also have safety features that make them easier to use than arrays, automated bounds checking and memory management

Vector

- From time to time the size of the array may not be enough, so a new bigger one is allocated, the older elements are copied to the new one, and the old one will be destroyed.

- Inserting at the end takes differential time, as sometimes there may be a need of extending the array.

- Removing the last element takes only constant time because no resizing happens.

- Inserting and erasing at the beginning or in the middle is linear in time.

- Compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Initializing a vector

```
vector<int> v1 = {1, 2, 3, 4 }; // size is 4

vector<string> v2; // size is 0;

vector<Shape*> v3(23); // size is 23; initial element value: nullptr

vector<double> v4(32,9.9); // size is 32; initial element value: 9.9

vector<double> v5(v4);  // a copy of v4
```

When we define a vector, we can give it an initial size (initial number of elements):

An explicit size is enclosed in ordinary parentheses, e.g., (23)

By default the elements are initialized to the element type's default

If we don't want the default value, we can specify one as a second argument

Accessing members of a vector

```
for (vector<Entry>::iterator it = book.begin() ; it != book.end(); ++it)
{
      cout<< it->name <<" "<< it->number << endl;
}
```

for (*range_declaration* **:** *range_expression* **)***loop_statement*

a declaration of a named variable, whose type is the type of the element of the sequence represented by range_expression, or a reference to that type. Often uses the auto specifier for automatic type deduction.

any expression that represents a suitable sequence or a braced-init-list.

```
struct Entry { string name; int number; };
```

```
void print_book(vector<Entry> & book)
{
    for (const auto& x : book)
         cout << x.name <<" "<<x.number << endl;
}
```

Vectors

Iterators	`begin()`	Returns an iterator pointing to the first element in the vector
	`end()`	Returns an iterator pointing to the theoretical element that follows the last element in the vector
Reverse Iterators	`rbegin()`	Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
	`rend()`	Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

https://en.cppreference.com/w/cpp/container/vector

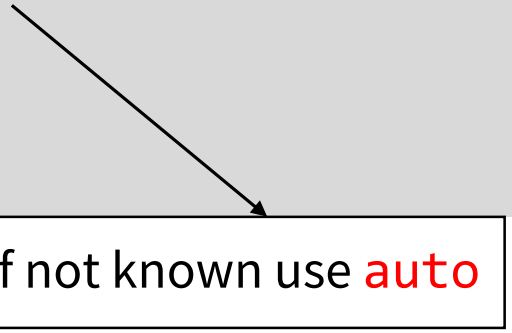
Example

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


int main(){
   vector <int> v ={1,2,3,4,5};

   cout<<"Incrementing the vector by 1:"<<endl;

   for (std::vector<int>::iterator it = v.begin() ; it != v.end(); ++it){
     *it = *it + 1;
      std::cout << ' ' << *it;
    }
}
```




```
Output:
Incrementing the vector by 1:
2 3 4 5 6
```

Basic Vector Operations

capacity	`empty()`	checks whether the container is empty
	`size()`	returns the number of elements
	`resize()`	resizes the container so that it contains *n* elements
	`capacity()`	returns the number of elements that can be held in currently allocated storage
	`shrink_to_fit()`	reduces memory usage by freeing unused memory

https://en.cppreference.com/w/cpp/container/vector

Basic Vector Operations

Element access	`at()`	access specified element with bounds checking (throws exception when a non-existent member is accessed)
	`operator[]`	access specified element (does not do range checking)
	`front()`	access the first element
	`back()`	access the last element

https://en.cppreference.com/w/cpp/container/vector

Basic Vector Operations

Modifiers	`assign()`	assigns new content
	`insert()`	inserts elements
	`erase()`	erases elements
	`clear()`	removes all elements from the vector
	`push_back()`	Adds a new element at the end of the vector, after its current last element
	`pop_back()`	removes the last element in the vector, effectively reducing the container size by one.
	`emplace()`	Adds a new element at a given position *inplace*

https://en.cppreference.com/w/cpp/container/vector

```
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;
    }
};
```

```
Size: 5
Capacity: 8
Element at Loc 0
a = 0 b = 0
Element at last Loc
a = 4 b = 4
Inserting a new element in the beginning
Element at Loc 0 is:
a = -1 b = -1
```

```
int main(){
vector<A> vecA;
 for (int i = 0; i <= 4; i++)
{
      vecA.push_back(A(i,i));
}
cout<<"Size: "<<vecA.size()<<endl;
cout<<"Capacity: "<<vecA.capacity()<<endl;

cout<<"Element at Loc 0"<<endl;
vecA[0].show();

cout<<"Element at last Loc"<<endl;
vecA.back().show();

cout<<"Inserting a new element in the beg"<<endl;
A a1(-1,-1);
vecA.insert(vecA.begin(),a1);

cout<<"Element at Loc 0 is: "<<endl;
vecA.at(0).show();
return 0;   }
```

Next Lecture

- File Handling in C++
- Dynamic Memory Allocation