Week 10 Lecture 2

NWEN 241
Systems Programming

Alvin Valera

Alvin.valera@ecs.vuw.ac.nz

Content

- Structures in C++
- Containers
- File Handling

Structures in C++

Structure in C vs Structure in C++

- C++ structures adds **extra features** to C structures

- Same declaration syntax

- C++ structures can –
  - have functions as members
  - treated like a *built-in* data type
  - be extended (supports inheritance)
  - define access specifiers (public, private, protected)

Structure in C vs Structure in C++

- C++ structures adds **extra features** to C structures

- Same declaration syntax

- C++ structures can –
  - have functions as members
  - treated like a *built-in* data type
  - be extended (supports inheritance)
  - define access specifiers (public, private, protected)

```
struct S {
  int a;
  int b;
  void set() {
    a = 10;
    b = 20;
  }
};
```

Structure in C vs Structure in C++

- C++ structures adds **extra features** to C structures
- Same declaration syntax
- C++ structures can –
  - have functions as members
  - treated like a *built-in* data type
  - be extended (supports inheritance)
  - define access specifiers (public, private, protected)

```
struct S {
  int a;
  int b;
  void set() {
    a = 10;
    b = 20;
  }
};

~~struct S s1;~~
S s1;
```

Structure in C vs Structure in C++

- C++ structures adds **extra features** to C structures
- Same declaration syntax
- C++ structures can –
  - have functions as members
  - treated like a *built-in* data type
  - be extended (supports inheritance)
  - define access specifiers (public, private, protected)

```
struct Base
{
   :
   :
};

struct Derived: Base {
   :
};
```

Structure in C vs Structure in C++

- C++ structures adds **extra features** to C structures

- Same declaration syntax

- C++ structures can –
  - have functions as members
  - treated like a *built-in* data type
  - be extended (supports inheritance)
  - define access specifiers (public, private, protected)

```
struct Base {
public:
   int a;
private:
   int b;
protected:
   int c;
};

struct Derived: Base {

}
```

Members are public by default

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 `vector<int>`
- 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 write different functions for different data types: 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

C++ Standard Library

- The C++ standard library provides a wide range of facilities that are usable in standard C++
- A large part of the C++ library is based on the **Standard Template Library (STL)**
- STL has four major components:
  - **Containers**
  - **Algorithms**
  - **Iterators**
  - **Function Objects**

STL Components

Containers

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

Algorithms

- Algorithms act on containers
- Independent of containers

Iterators

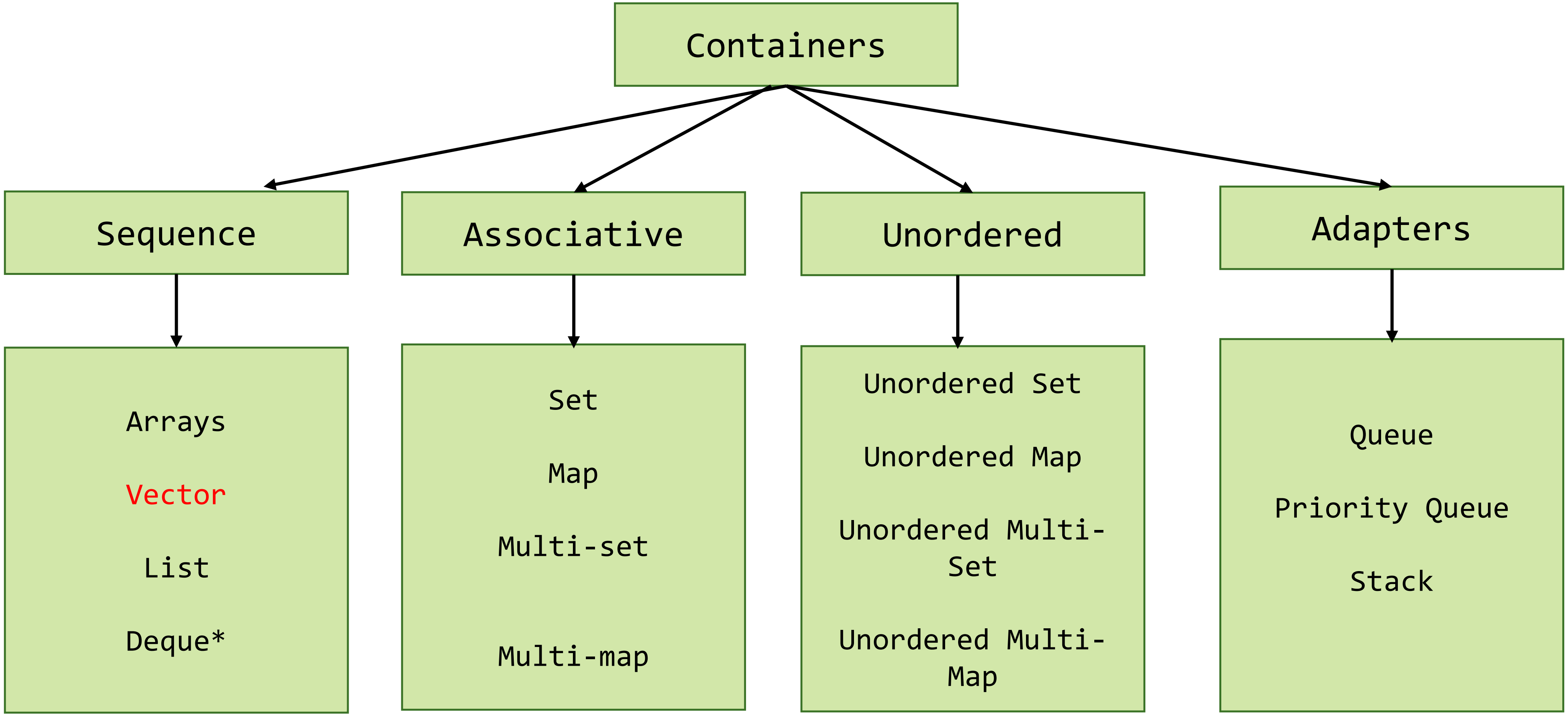
- Generalized pointers that facilitate use of containers
- Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers

Function Objects

- Allows an object to be invoked or called as if it were an ordinary function

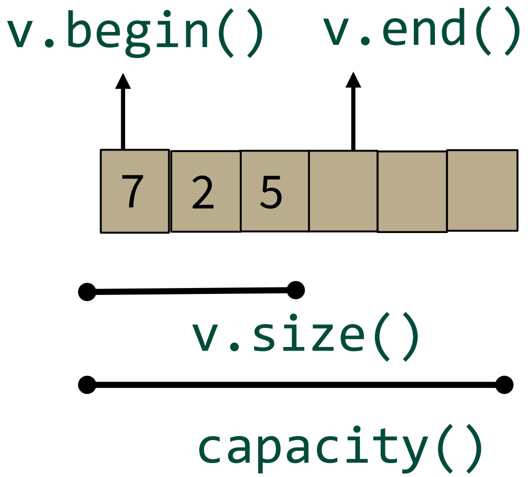
Containers

- 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 (implementation dependent). 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<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

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
```

can also use auto – the auto keyword dynamically determines the data type of the assigned value

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; };
//book is a vector of Entries
```

Range-based for loop

```
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

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 ***in place*** *(without requiring creation of a temporary object)*

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;
    }
};
```

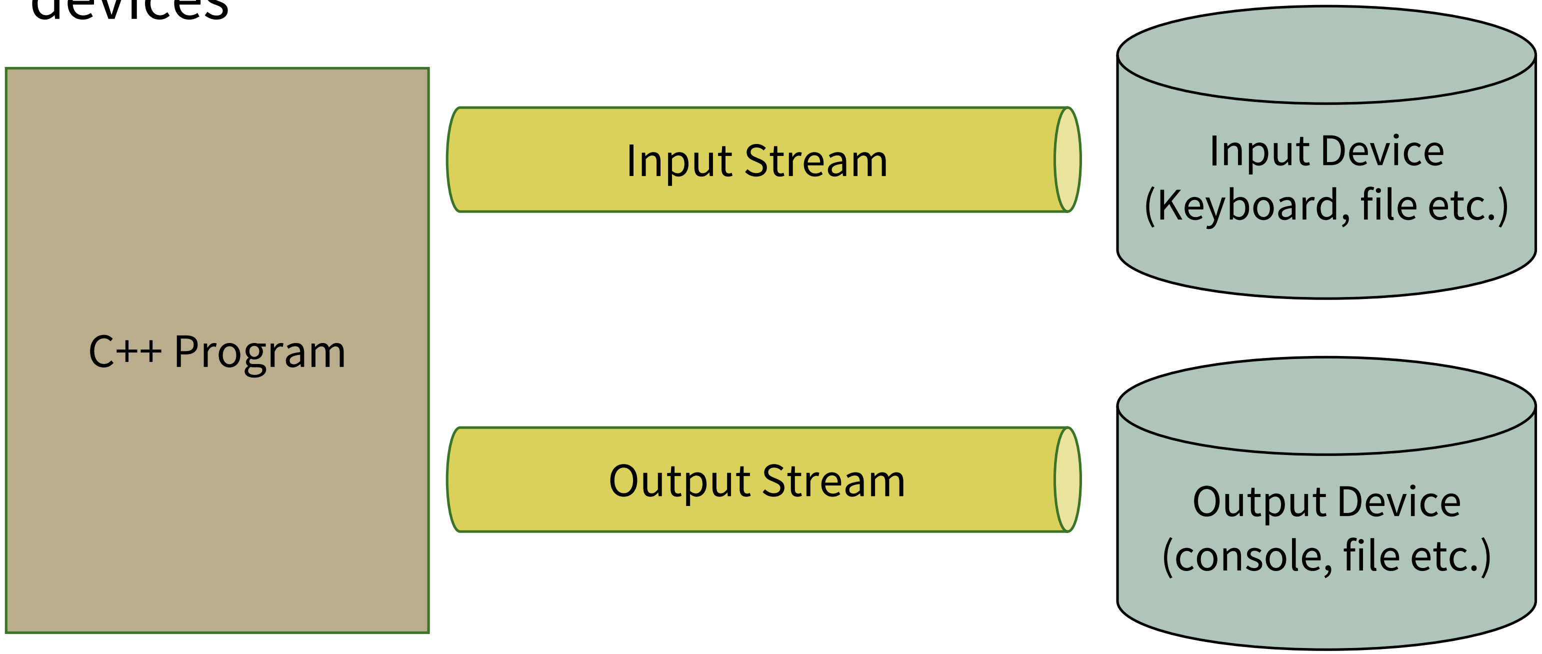
```
int main(void) {
  vector<A> vecA;
  for (int i = 0; i <= 4; i++) {
    vecA.push_back(A(i,i));
  }
  cout<<"Size: "<<vecA.size()<<endl;
  cout<<"Capacity: "<<vecA.capacity()<<endl;

  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;
}
```

```
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
```

File Handling in C++

Recap: I/O Basics

- C/C++ I / O are based on *streams*, which are sequence of bytes flowing in and out of the programs
- Streams acts as an intermediaries between the programs and the actual IO devices

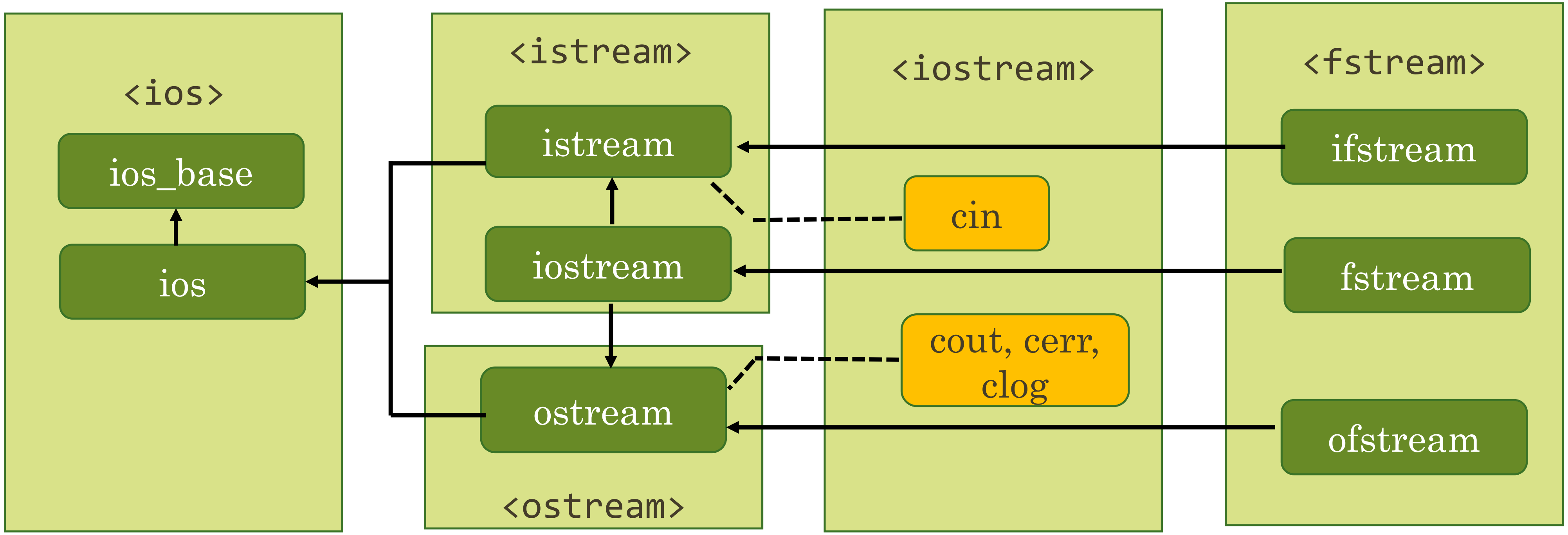


With **input** streams, the **extraction operator (>>)** is used to remove values from the stream

With **output** streams, the **insertion operator (<<)** is used to put values in the stream

C++ IO operations are *device independent*. The same set of operations can be applied to different types of IO devices.

Stream Hierarchy



A stream is represented by an object of a particular class.

File Streams

- `ifstream` – stream class to **read** from files
- `ofstream` – stream class to **write** to files.
- `fstream` - stream class to both **read (from) and write (to)** files.
- The stream objects `cin`, `cout`, `cerr` and `clog` are declared in `iostream` header file and are automatically added to our program, when `iostream` header file is included in our program.
- In contrast, we are responsible for creating and setting up our own file streams.

Steps for File IO:

1. Create file stream objects

```
ifstream  fsIn;   //input
ofstream fsOut;   // output
fstream fsBoth;   //input & output
```

2. Open the file

```
fsIn.open("data.txt",fileopenmode);
```

File open mode

OR Combine the two steps

```
ifstream fsIn("data.txt", fileopenmode);
```

File Open Modes

Name	Description
`ios::in`	Open file to read (default for ifstream)
`ios::out`	Open file to write (default for ofstream)
`ios::app`	Output operations happen at the end of the file, appending to its existing contents.
`ios::ate`	The stream's position indicator is set to the end of the file.
`ios::trunc`	Deletes all previous content in the file (empties the file)
`ios::nocreate`	If the file does not exists, new file is not created
`ios::noreplace`	If the file exists, trying to open it with the open() function, returns an error.
`ios::binary`	Opens the file in binary mode.

`ios::in` is default for ifstream

`ios::out` is default for ofstream

`ios::in |ios::out` is the default for fstream

Next Lecture

- File Handling (continuation)

- Dynamic Memory Allocation