Structured Programming

Dr. Abu Nowshed Chy

Department of Computer Science and Engineering

University of Chittagong

March 03, 2025

Faculty Profile

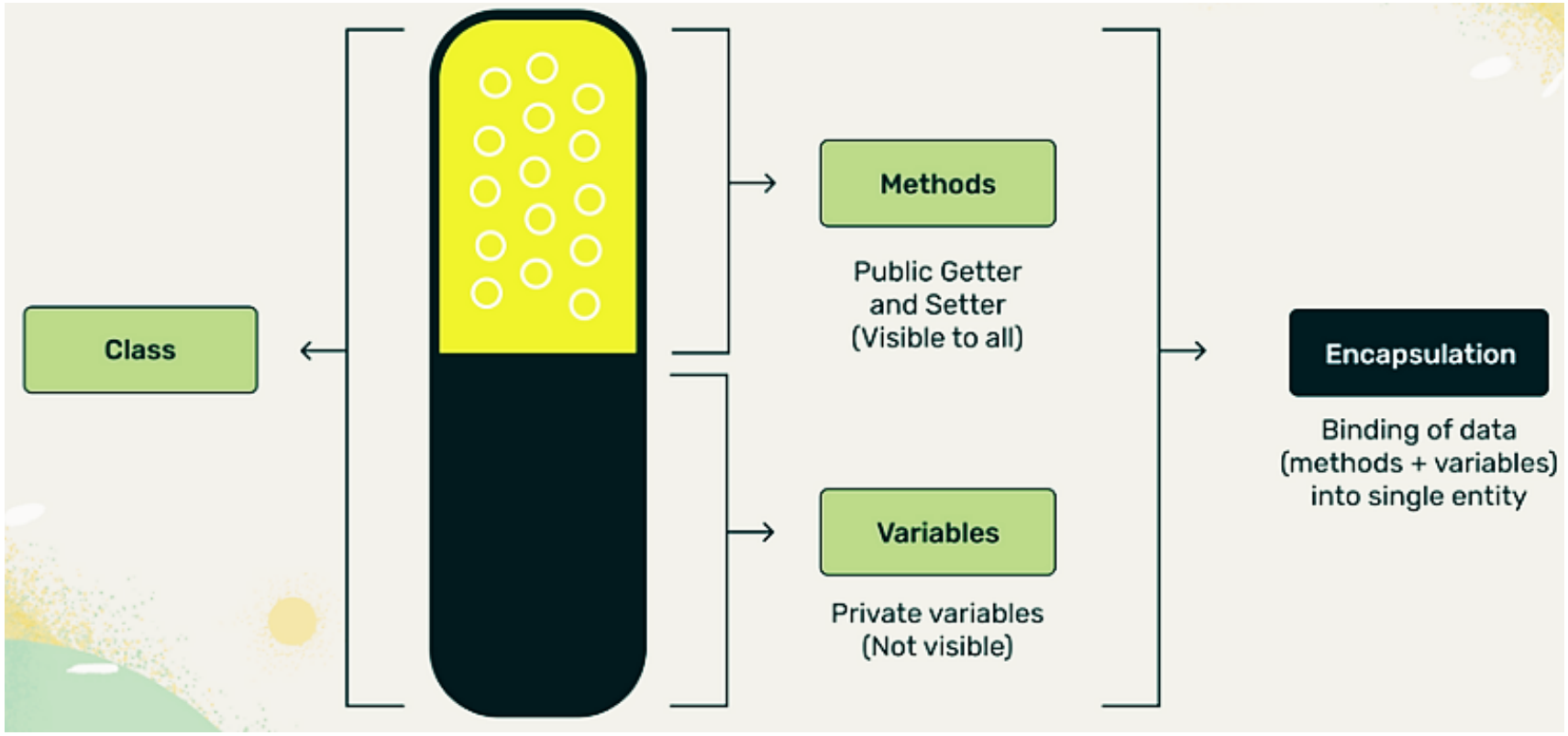
Class And Object

The three important features of OOP are-

- Encapsulation
- Polymorphism
- Inheritance

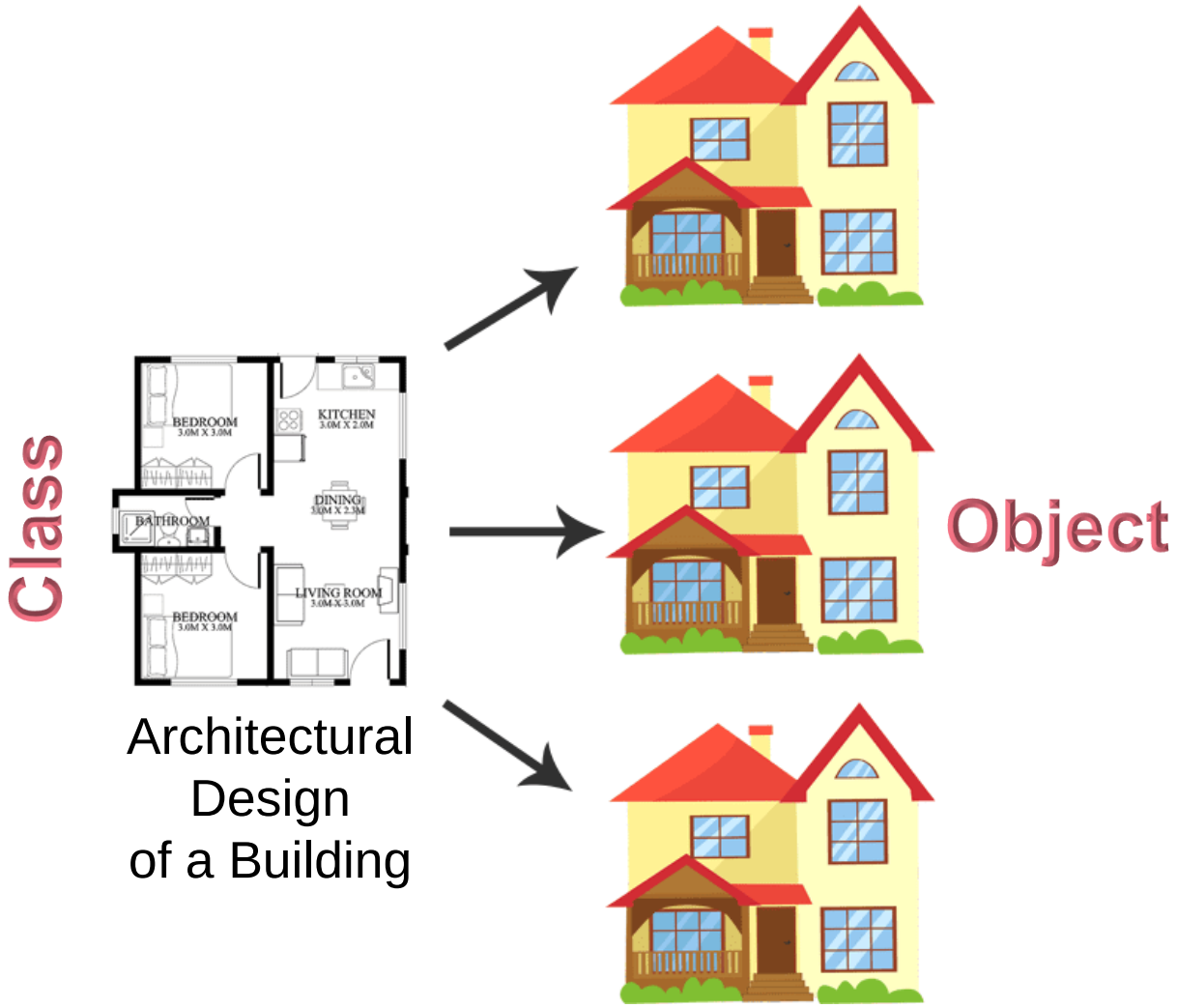
Class And Object

Wrapping up of data and function into a single unit

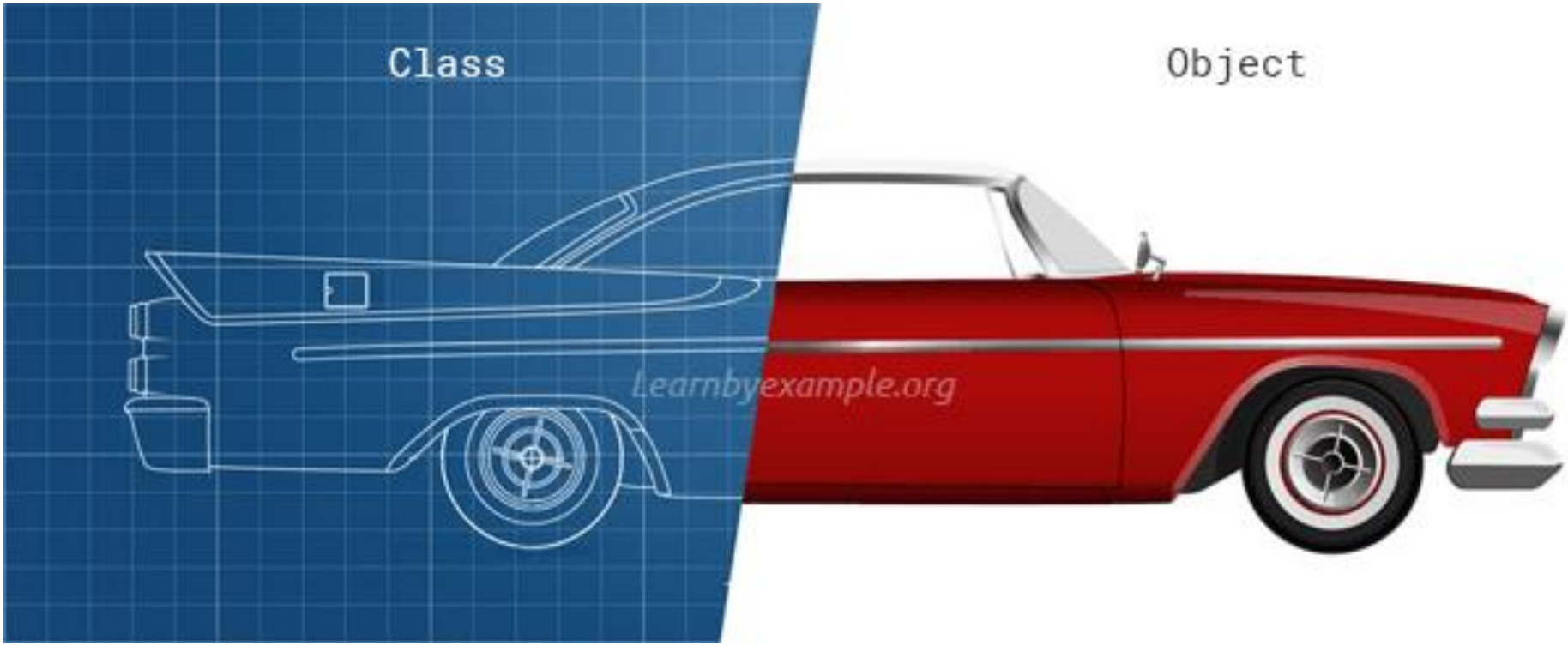


Class And Object

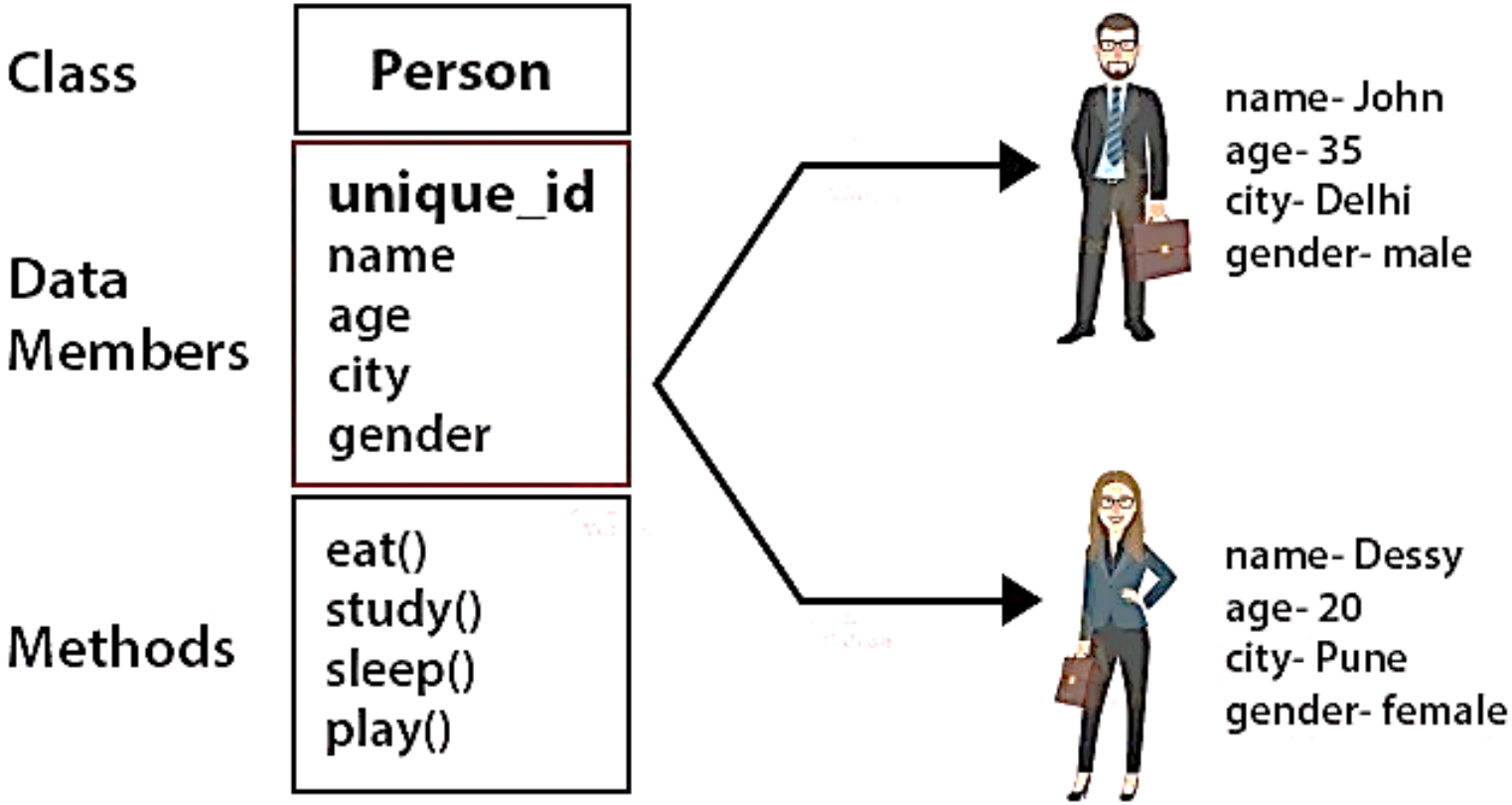
Class And Object



Class And Object



Class And Object



C++ → Hello, World! explained

main.cpp

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       cout << "Hello world!" << endl;
8       return 0;
9   }
10
```

The *main* routine – the start of **every** C++ program! It returns an integer value to the operating system and (in this case) takes no arguments: main()

The **return** statement returns an integer value to the operating system after completion. 0 means “no error”. C++ programs **must** return an integer value.

C++ → Hello, World! explained

loads a *header* file containing function and class definitions

Loads a *namespace* called *std*. Namespaces are used to separate sections of code for programmer convenience. To save typing we'll always use this line.

main.cpp

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       cout << "Hello world!" << endl;
8       return 0;
9   }
10
```

- *cout* is the *object* that writes to the stdout device, i.e. the console window.
- It is part of the C++ standard library.
- Without the "using namespace std;" line this would have been called as *std::cout*. It is defined in the *iostream* header file.
- << is the C++ *insertion operator*. It is used to pass characters from the right to the object on the left. *endl* is the C++ newline character.

C++ → Hello, World! explained

C++ language headers aren't referred to with the .h suffix. <iostream> provides definitions for I/O functions, including the *cout* function.

- C++ (along with C) uses *header files* as to hold definitions for the compiler to use while compiling.
- A source file (file.cpp) contains the code that is compiled into an object file (file.o).
- The header (file.h) is used to tell the compiler what to expect when it assembles the program in the linking stage from the object files.
- Source files and header files can refer to any number of other header files.

```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world ;
    cout <<  msg << endl;
    msg[0] = 'h';
    cout <<  msg << endl;
    return 0;
}
```

C++ → Hello, World! explained

- Let's put the message into some variables of type *string* and print some numbers.
- Things to note:
  - Strings can be concatenated with a + operator.
  - No messing with null terminators or *strcat()* as in C
- Some string notes:
  - Access a string character by brackets or function:
    - msg[0] → "H" or msg.at(0) → "H"
    - C++ strings are *mutable* – they can be changed in place.
- Press F9 to recompile & run.

```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world;
    cout <<  msg << endl;
    msg[0] = 'h';
    cout <<  msg << endl;
    return 0;
}
```

C++ → Hello, World! explained

- *string* is not a basic type (more on those later), it is a class.
- `string hello` creates an *instance* of a string called “hello”.
- `hello` is an object.
- Remember that a class defines some data and a set of functions (methods) that operate on that data.

```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world ;
    cout <<  msg << endl;
    msg[0] = 'h';
    cout <<  msg << endl;
    return 0;
}
```

C++ → Hello, World! explained

- Update the code as you see here.
- After the last character is entered C::B will display some info about the string class.
- If you click or type something else just delete and re-type the last character.
- Ctrl-space will force the list to appear.

```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world ;
    cout <<  msg << endl;
    msg[0] = 'h';
    cout <<  msg << endl;

    msg

    return 0;
}
```

C++ → Hello, World! explained

*main.cpp

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       string hello = "Hello";
8       string world = "world!";
9       string msg = hello + " " + world ;
10      cout <<  msg << endl;
11      msg[0] = 'h';
12      cout <<  msg << endl;
13
14      msg
15      hello: string
16      msg: string
17  }   world: string
18
```

```
(m) (__gthrw_pthread_cond_signal, pthread_cond_signal, pthread_cond_signal)(): (__gthrw_pthread_cond_init, pthread_cond...
(m) (__gthrw_pthread_key_create, pthread_key_create, pthread_key_create)(): (__gthrw_pthread_cond_timedwait, pthread_co...
(m) (__gthrw_pthread_mutex_init, pthread_mutex_init, pthread_mutex_init)(): (__gthrw_pthread_mutex_timedlock, pthread_m...
(m) (__gthrw_pthread_mutex_lock, pthread_mutex_lock, pthread_mutex_lock)(): (__gthrw_pthread_cancel, pthread_cancel, pt...
(m) (__gthrw_pthread_self, pthread_self, pthread_self)(): (__gthrw_pthread_join, pthread_join, pthread_join)(__gthrw_pt...
(m) (__gthrw_pthread_setspecific, pthread_setspecific, pthread_setspecific)(): (__gthrw_pthread_once, pthread_once, pth...
(m) * _pthread_key_dest(): void
(m) abort(): void
(m) address(): const_pointer
```

```
main
string msg
(variable)

Open declaration

Close Top
```

List of other string objects

List of string methods

Shows this function (main) and the type of msg (string)

- Next: let's find the size() method without scrolling for it.

C++ → Hello, World! explained

*main.cpp | basic_string.h

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       string hello = "Hello";
8       string world = "world!";
9       string msg = hello + " " + world
10      cout <<  msg << endl;
11      msg[0] = 'h';
12      cout <<  msg << endl;
13
14      msg.siz
15
16      re
17  }
18
```

```
SIGTERM
SIG_ACK
sig_atomic_t
SIG_BLOCK
SIG_DFL
SIG_ERR
SIG_GET
SIG_IGN
SIG_SETMASK
SIG_SGE
SIG_UNBLOCK
size(): size_type
```

std::__cxx11::basic_string

public size_type **size**() const
(function)

Open declaration
Open implementation

Close Top

- Start typing “msg.size()” until it appears in the list. Once it’s highlighted (or you scroll to it) press the Tab key to auto-enter it.
- On the right you can click “Open declaration” to see how the C++ compiler defines size(). This will open *basic_string.h*, a built-in file.

C++ → Hello, World! explained

- Tweak the code to print the number of characters in the string, build, and run it.
- From the point of view of main(), the *msg* object has hidden away its means of tracking and retrieving the number of characters stored.
- Note: while the string class has a **huge** number of methods your typical C++ class has far fewer!

```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello" ;
    string world = "world!" ;
    string msg = hello + " " + world ;
    cout <<  msg << endl ;
    msg[0] = 'h';
    cout <<  msg << endl ;

    cout << msg.size() << endl ;

    return 0;
}
```

- Note that *cout* prints integers without any modification!

Standard Template Library (STL)

- STL had three basic components:
  - Containers

    Generic class templates for storing collection of data.
  - Algorithms

    Generic function templates for operating on containers.
  - Iterators

    Generalized ‘smart’ pointers that facilitate use of containers. They provide an interface that is needed for STL algorithms to operate on STL containers.

Why we use STL?

- STL offers an assortment of containers
- STL publicizes the time and storage complexity of its containers
- STL containers grow and shrink in size automatically
- STL provides built-in algorithms for processing containers
- STL provides iterators that make the containers and algorithms flexible and efficient.
- STL is extensible which means that users can add new containers and new algorithms such that:
  - STL algorithms can process STL containers as well as user defined containers
  - User defined algorithms can process STL containers as well user defined containers

STL – Standard Template Library

- Collections of useful classes for common data structures
- Ability to store objects of any type (template)
- Study of containers
- Containers form the basis for treatment of data structures
- Container – class that stores a collection of data
- STL consists of 3 container classes:
  - Sequence containers
  - Adapter containers
  - Associative containers

STL Containers

- Sequence Container
  - Stores data by position in linear order:
  - First element, second element, …….
- Associate Container
  - Stores elements by key, such as name, social security number or part number
  - Access an element by its key which may bear no relationship to the location of the element in the container
- Adapter Container
  - Contains another container as its underlying storage structure

STL Containers

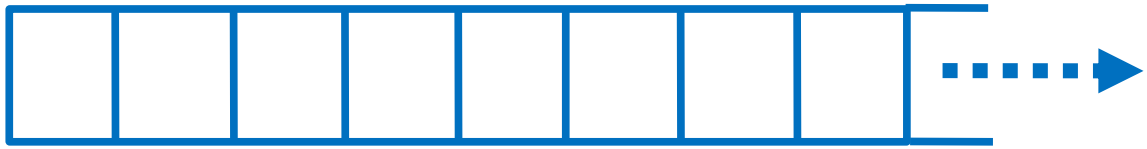
- Sequence Container
  - Vector
  - Deque
  - List
- Adapter Containers
  - Stack
  - Queue
  - Priority queue
- Associative Container
  - Set, multiset
  - Map, multimap

STL Sequence Containers

- Sequence Container
  - Stores data by position in linear order:
  - First element, second element, …….
  - Ordered Collections.
  - Every element has a certain position.
  - Position independent of the value of the element.
  - Position depends on order and place of insertion.
    - Vector
    - Deque
    - List

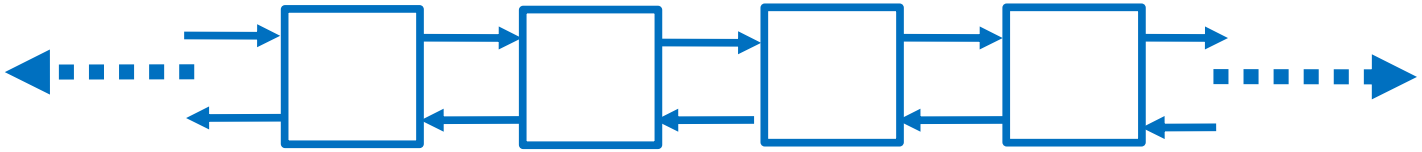
STL Sequence Containers

Vector



Deque

List



Vector Container

- Generalized array that stores a collection of elements of the same data type
- Vector – similar to an array
  - Vectors allow access to its elements by using an index in the range from 0 to n-1 where n is the size of the vector
- Vector vs array
  - Vector has operations that allow the collection to grow and contract dynamically at the rear of the sequence

Vector Container

- The implementation of a vector is based on arrays
- Vectors allow direct access to any element via indexes
- Insertion at the end is normally efficient.
  - The vector simply grows
- Insertion and deletion in the middle is expensive
  - An entire portion of the vector needs to be moved

Vector Container

- When the vector capacity is reached then
  - A larger vector is allocated,
  - The elements of the previous vector are copied and
  - The old vector is deallocated
- To use vectors, we need to include the header <vector>
- Some functions of the class vector include
  - size, capacity, insert…

Sample Program

Example:

```
#include <vector>
.
.
.
vector<int> scores (100);
                    //100 integer scores
vector<Passenger>passengerList(20);
                    //list of 20 passengers
```

Vector

- ✓ Header file <vector>
- ✓ Important member functions
  - begin() and end(): return iterators
  - clear(): delete all elements
  - push_back(): insert at end
  - pop_back(): delete last element
  - size(): number of elements
  - empty(): if vector is empty
  - resize(): change vector size

Vector

```
#include <iostream>
#include <vector>  //vector class-template

using namespace std;

int main()
{
   vector<int> v;

   // add integers at the end of the vector
   v.push_back( 2 );
   v.push_back( 3 );
   v.push_back( 4 );

   cout << "\nThe size of v is: "
        << v.size()
        << "\nThe capacity of v is: "
        << v.capacity()
        << '\n';

    for(int num : v) {
        cout << num << ", ";
    }

    return 0;
}
```

How to access Components - Iterator

- Iterator is an object that can access a collection of like objects one object at a time.
- An iterator can traverse the collection of objects.
- Each container class in STL has a corresponding iterator that functions appropriately for the container
- For example: an iterator in a vector class allows random access
- An iterator in a list class would not allow random access (list requires sequential access)

Common Iterator Operations

*	Return the item that the iterator currently references
++	Move the iterator to the next item in the list
--	Move the iterator to the previous item in the list
==	Compare two iterators for equality
!=	Compare two iterators for inequality

Vector

```
#include <iostream>
#include <vector>  //vector class-template

using namespace std;

int main()
{
   vector<int> v;

   // add integers at the end of the vector
   v.push_back( 2 );
   v.push_back( 3 );
   v.push_back( 4 );

   cout << "\nThe size of v is: "
        << v.size()
        << "\nThe capacity of v is: "
        << v.capacity()
        << '\n';

  // display the content of v
   cout << "Content of v is: \n";

   vector<int>::const_iterator it;

   for (it = v.begin(); it != v.end(); it++)
   {
         cout <<  *it << '\n';
   }
    return 0;
}
```


Thank you!