C++ Data Structures
Prerequisites

CHAPTER 4: C++ STL ALGORITHMS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

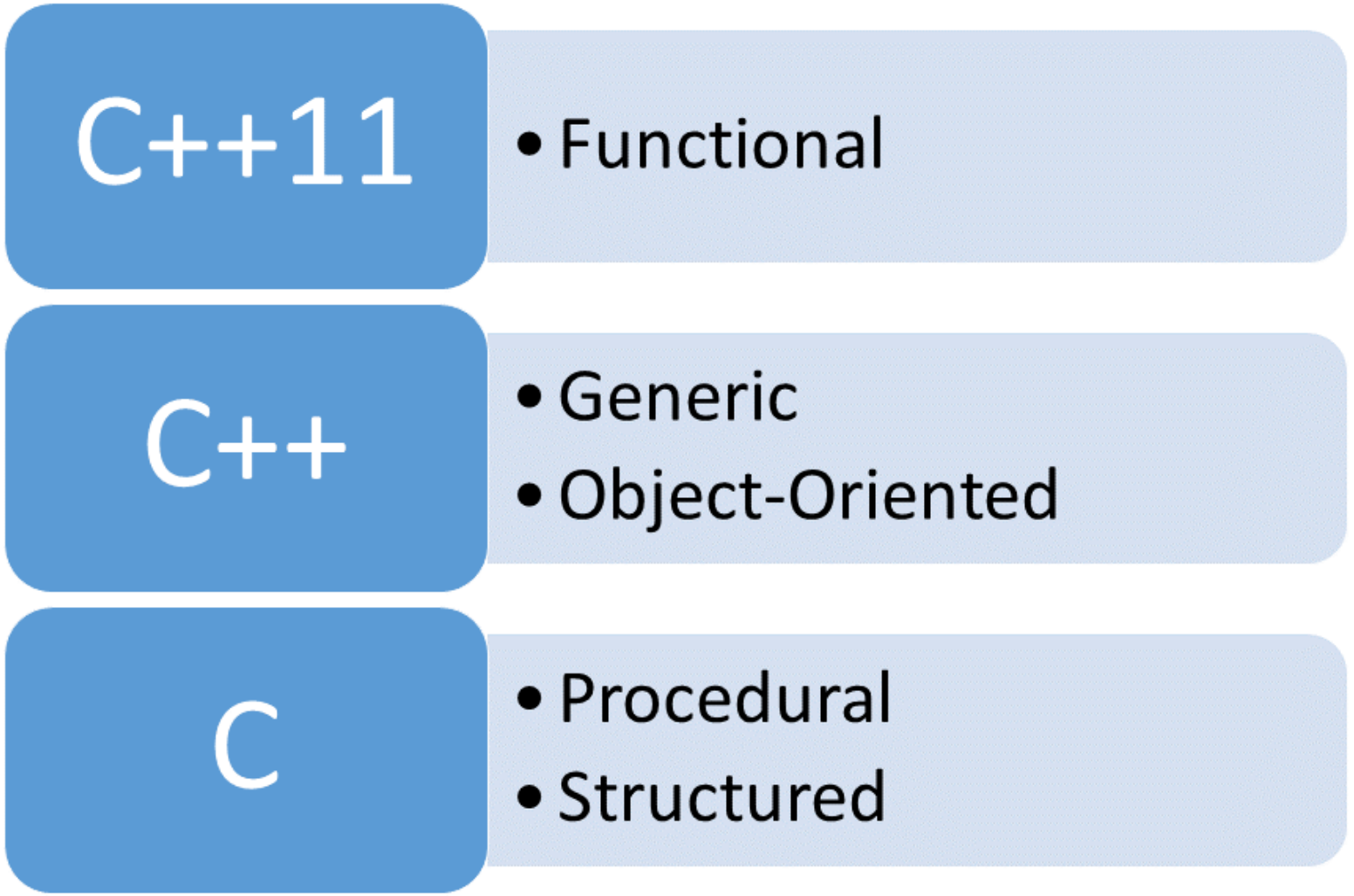
- C++ Programming Paradigms
- Generic Programming and Standard Template Library (STL)
- C++ STL Algorithms
- Constructors (Default, Overloading, Copy, Move, List)
- Delete
- Deep Copy/Shallow Copy

LECTURE 1

Programming Paradigms

Abstraction of C++ Languages

- C++ is not a functional programming language. C++ has its roots in procedural and object-oriented programming.
- So, it's quite surprising that programming in a functional style becomes more and more important in C++. That is not only true for C++. That holds also for python, which has a lot of functional features and even for Java. Now Java has lambda functions.


C++11
• Functional
C++
• Generic
• Object-Oriented
C
• Procedural
• Structured

Object-oriented programming

•Object-oriented programming is based on the three concepts encapsulation, inheritance, and polymorphism.

Encapsulation

- An object encapsulates its attributes and methods and provides them via an interface to the outside world. This property that an object hides its implementation is often called *data hiding.*

Inheritance

- A derived class get all characteristics from its base class. You can use an instance of a derived class as an instance of its base class. We often speak about *code reuse* because the derived class automatically gets all characteristics of the base class.

Polymorphism

- Polymorphism is the ability to present the same interface for differing underlying data types. The term is from Greek and stands for many forms.

Object-Oriented Programming

```
1   class HumanBeing{
2   public:
3     HumanBeing(std::stringn):name(n){}
4     virtual std::string getName() const{
5       return name;
6     }
7   private:
8     std::string name;
9   };
10
11  class Man: public HumanBeing{};
12
13  class Woman: public HumanBeing{};
```

- In the example, you only get the name of HumanBeing by using the method getName in line 4 (encapsulation).
- In addition, getName is declared as virtual. Therefore, derived classes can change the behaviour of their methods and therefore change the behaviour of their objects (polymorphism).
- Man and Woman are derived from HumanBeing.

Generic Programming

```
1   template <typename T> void xchg(T& x, T& y){
2     T t= x;
3     x= y;
4     y= t;
5   };
6   int i= 10;
7   int j= 20;
8   Man huber;
9   Man maier;
10
11  xchg(i,j);
12  xchg(huber,maier);
```

Template Function (Generic Function)

- The key idea of **generic programm**ing or programming with templates is to define families of functions or classes. By providing the concrete type you get automatically a function or a class for this type. Generic programming provides a similar abstraction to object-oriented programming. A big difference is that polymorphism of object-oriented programming will happen at runtime; that polymorphism of generic programming will happen in C++ at compile time. That the reason why polymorphism at runtime is often called dynamic polymorphism but polymorphism at compile is often called static polymorphism.
- By using the function template, I can exchange arbitrary objects.

Generic Programming

```
1  template <typename T, int N>
2  class Array{
3  public:
4    int getSize() const{
5      return N;
6    }
7  private:
8    T elem[N];
9  };
10
11 Array<double,10> doubleArray;
12 std::cout << doubleArray.getSize() << std::endl;
13
14 Array<Man,5> manArray;
15 std::cout << manArray.getSize() << std::endl;
```

Template Class (Generic Container)

- It doesn't matter for the function template if I exchange numbers or men (line 11 and 12). In addition, I have not to specify the type parameter (line) because the compiler can derive it from the function arguments (line 11 and 12).
- The automatic type deduction of function templates will not hold for class templates. In the concrete case, I have to specify the type parameter T and the non-type parameter N (line 1).
- Accordingly, the application of the class template Array is independent of the fact, whether I use doubles or men.

Functional programming

```
1  std::vector<int> vec{1,2,3,4,5,6,7,8,9};
2  std::vector<std::string> str{"Programming","in","a","functional","style."};
3
4  std::transform(vec.begin(),vec.end(),vec.begin(),
5                 [](int i){ return i*i; }); // {1,4,9,16,25,36,49,64,81}
6
7  auto it= std::remove_if(vec.begin(),vec.end(),
8                          [](int i){ return ((i < 3) or (i > 8)) }); // {3,4,5,6,7,8}
9  auto it2= std::remove_if(str.begin(),str.end(),
10                          [](string s){ return (std::lower(s[0])); }); // "Programming"
11
12
13 std::accumulate(vec.begin(),vec.end(),[](int a,int b){return a*b;}); // 362880
14 std::accumulate(str.begin(),str.end(),
15                 [](std::string a,std::string b){return a + ":"+ b;});
16                 // "Programming:in:a:functional:style."
```

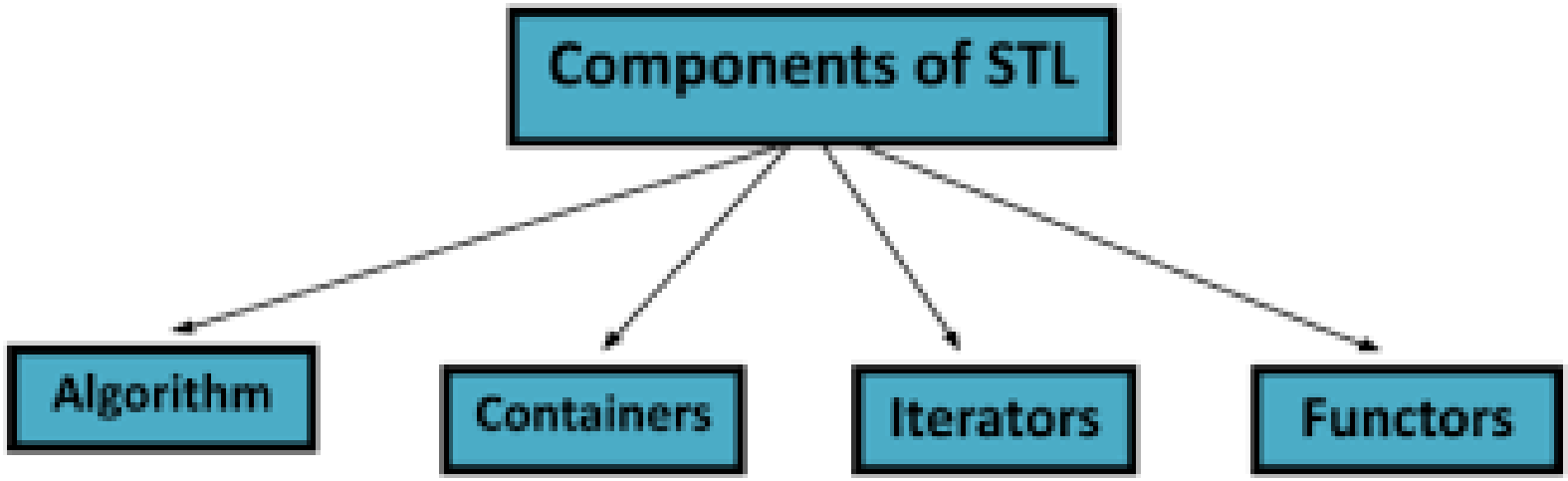
Functional Closure / Lambda Expression

Functional programming

- A will only say a few words about functional programming because I will and can not explain the concept of functional programming in a short remark. Only that much. I use the code snippet of the pendants in C++ to the typical functions in functional programming: map, filter, and reduce. These are the functions **std::transform**, **std::remove_if**, and **std::accumulate**.
- I apply in the code snippet two powerful features of functional programming. Both are now mainstream in modern **C++:** automatic type deduction with auto and lambda functions.

LECTURE 2

Generic Programming


Components of STL
Algorithm
Containers
Iterators
Functors

Standard Template Library in C++

Container

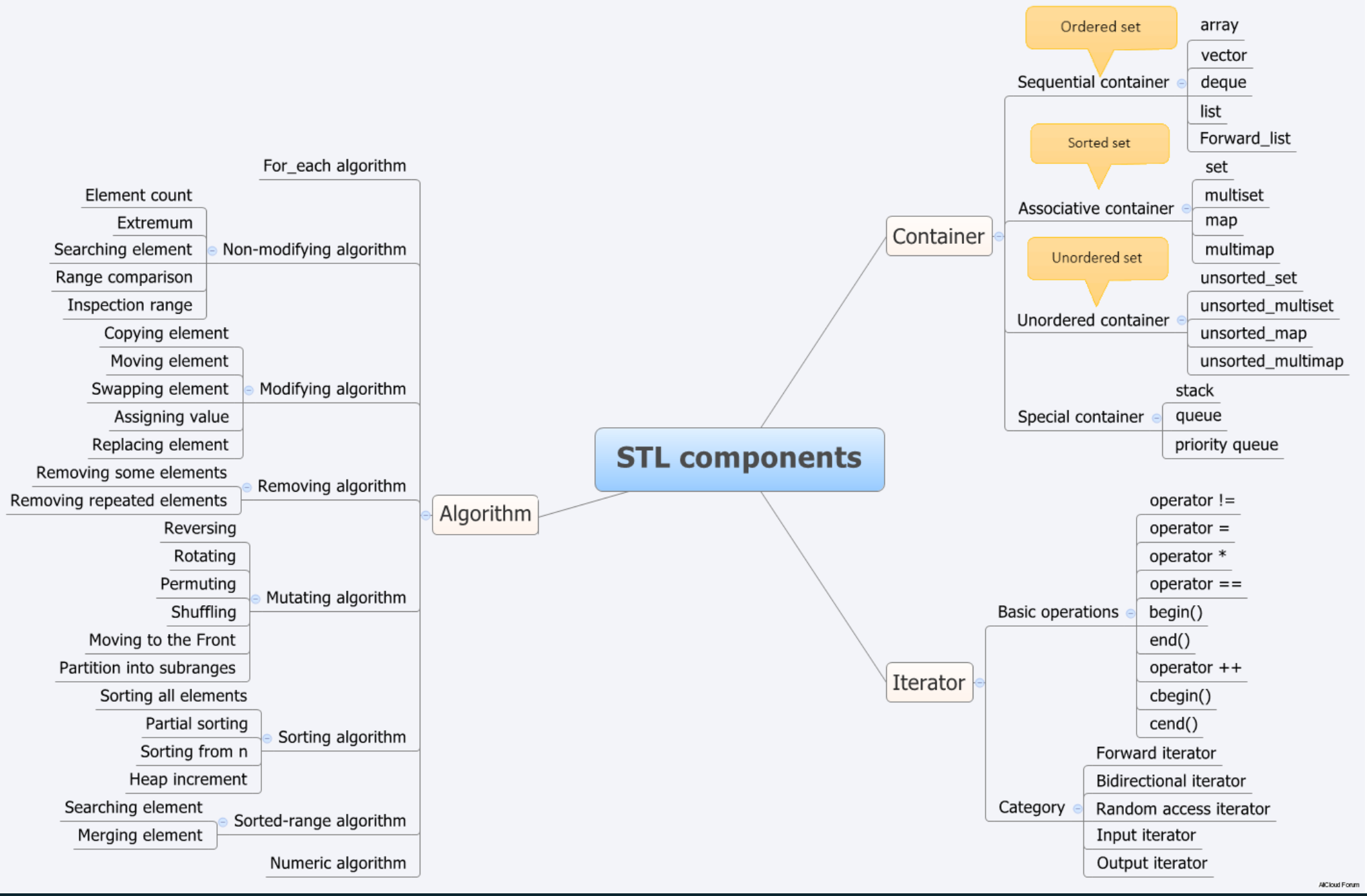
- Sequence Containers
- Associative Containers
- Container Adapters
- Unordered Associative Containers

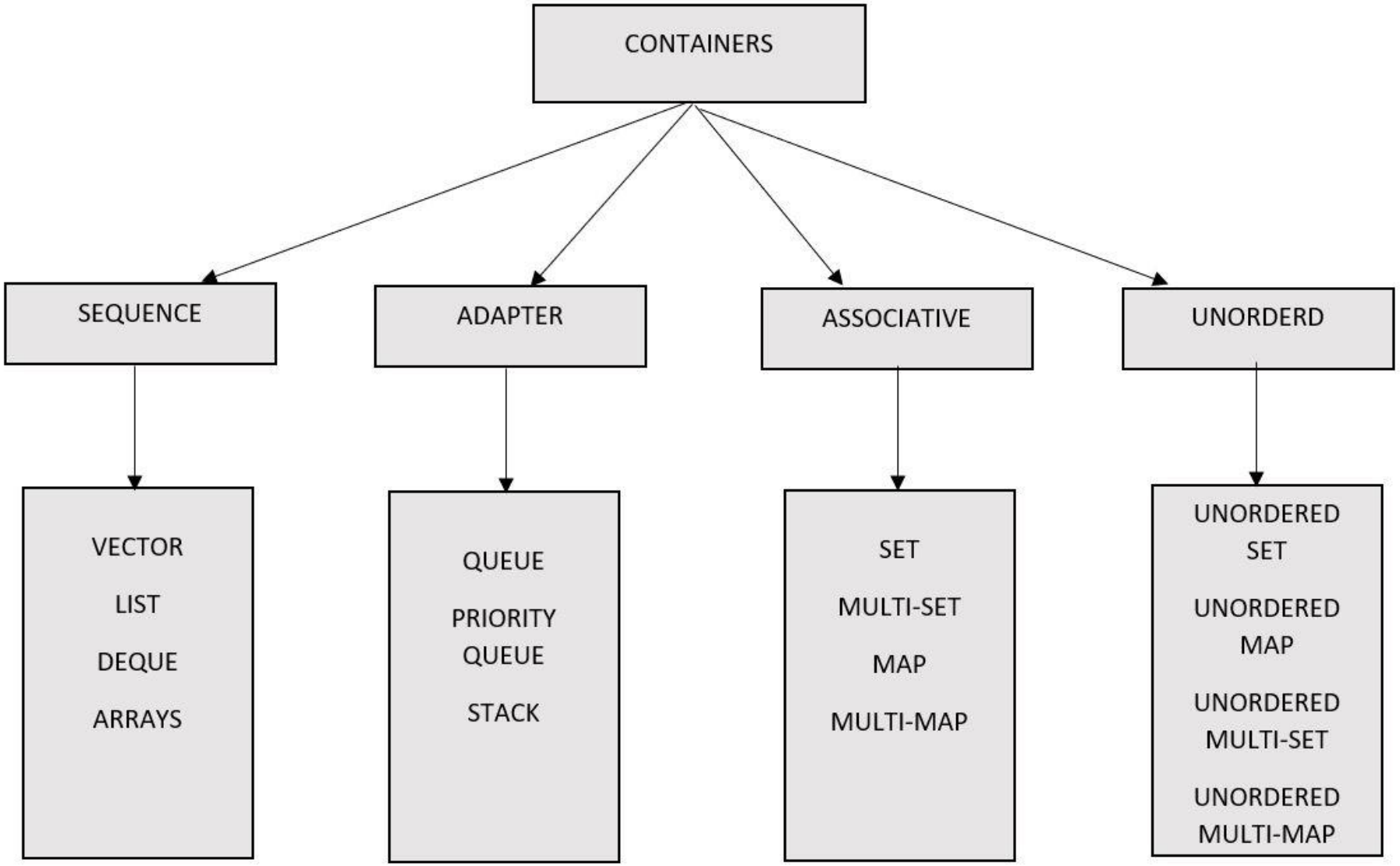
Iterator

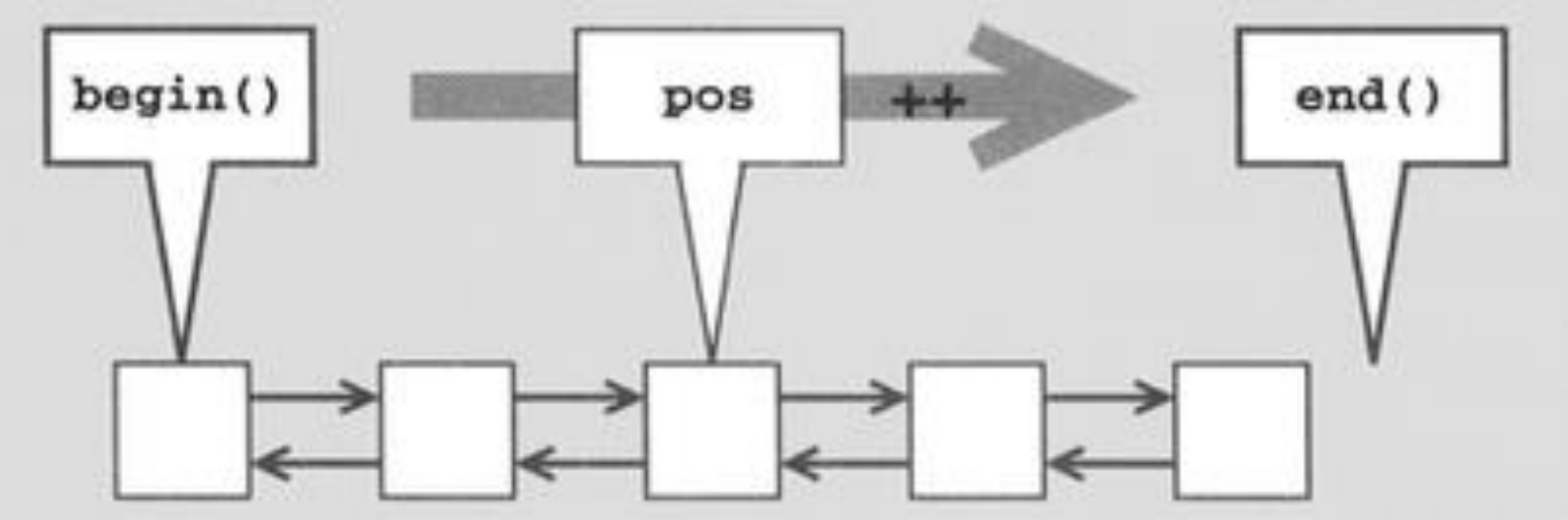
- begin()
- next()
- prev()
- advance()
- end()

Algorithm

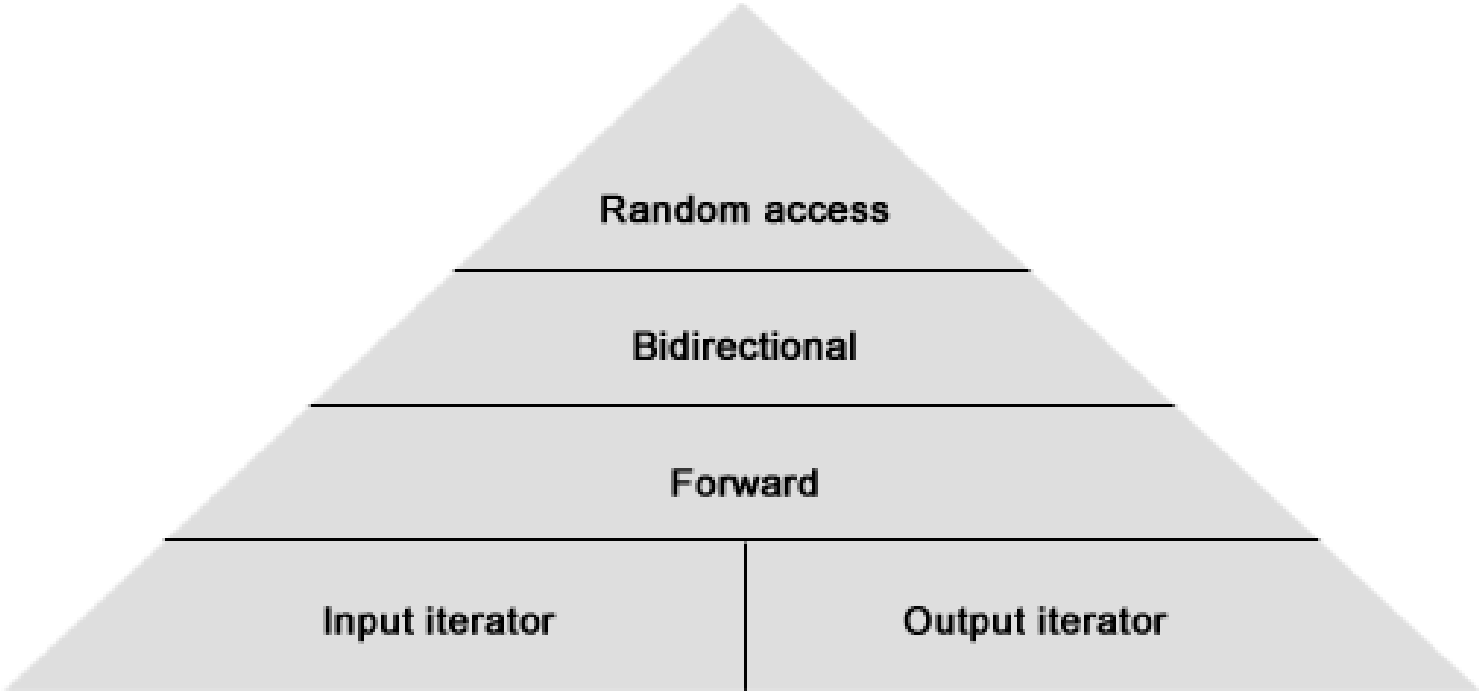
- Sorting Algorithms
- Search algorithms
- Non modifying algorithms
- Modifying algorithms
- Numeric algorithms
- Minimum and Maximum operations




CONTAINERS
SEQUENCE
ADAPTER
ASSOCIATIVE
UNORDERD
VECTOR
LIST
DEQUE
ARRAYS
QUEUE
PRIORITY QUEUE
STACK
SET
MULTI-SET
MAP
MULTI-MAP
UNORDERED SET
UNORDERED MAP
UNORDERED MULTI-SET
UNORDERED MULTI-MAP


begin()
pos
++
end()

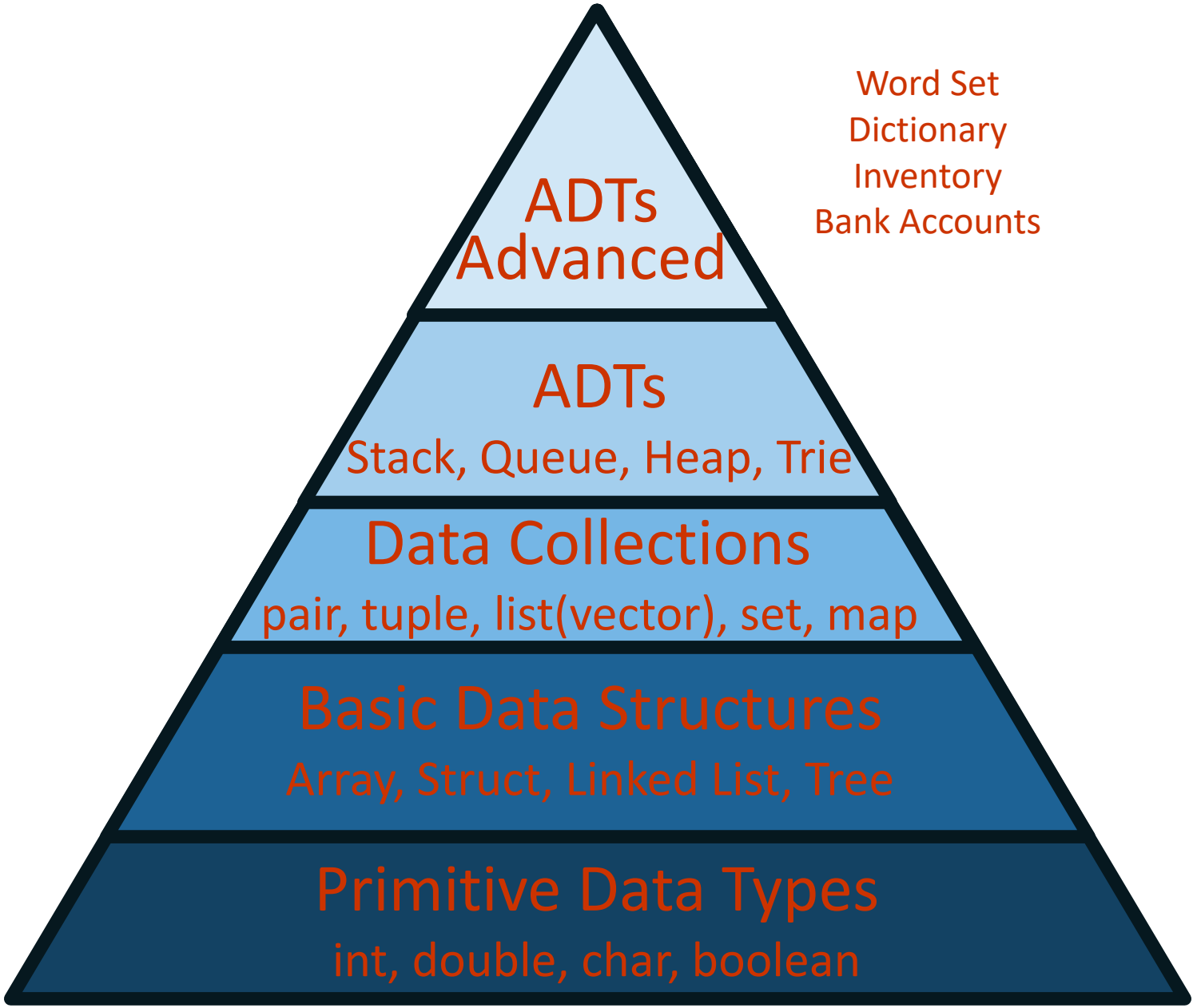
Iterator categories



Iterator categories	Provider
Input iterator	istream
Output iterator	ostream
Forward iterator	
Bidirectional iterator	list, set, multiset, map, multimap
Random access iterator	vector, deque, array

LECTURE 3

Data Structures, Data Collections, and Abstract Data Types



Word Set
Dictionary
Inventory
Bank Accounts

LECTURE 4

Constructors

Constructors

- As you might guess, constructors are a pretty important part of object-oriented programs. Constructors are important enough that five different kinds are independently named to make them easier to talk about.
  - Default Constructor (no-arg)
  - Overloaded Constructor (w-arg)
  - Copy Constructor
  - Move Constructor
  - List Constructor

Overloaded constructors

A a(3);

- Constructors are called whenever an object is instantiated, and, like any overloaded function, the specific constructor that gets called is determined by the arguments appearing in the call. That is, the constructor function whose parameter list matches the arguments in the function call, is selected to run.

Constructor Prototypes

```
class Foo
{
        public:
                Foo();

                Foo(int x);

                Foo(int x, int y);
};
```

Constructor Calls

```
Foo f1;                         // (a)
Foo* f2 = new Foo;
//Foo f1();                     // not in VS
//Foo* f2 = new Foo();          // okay in VS

Foo f3(5);                      // (b)
Foo* f4 = new Foo(5);

Foo f5(5, 10);                  // (c)
Foo* f6 = new Foo(5, 10);
```

```
class Time
{
    private:
        int     hours;
        int     minutes;
        int     seconds;
};
```

(a)

```
class Time
{
    private:
        int     hours = 0;
        int     minutes = 0;
        int     seconds = 0;
};
```

(b)

```
Time() : hours(0), minutes(0), seconds(0) {}
```

Or

```
Time()
{
        hours = 0;
        minutes = 0;
        seconds = 0;
}
```

(c)

```
Time t1;
Time* t2 = new Time;

Time t3();
Time* t4 = new Time();
```

(d)

Figure: Default constructor examples. The example illustrates the evolution of the C++ syntax relating to object construction or initialization, which presents contemporary programmers with some options.

Copy Constructor

A copy constructor creates a new object by copying an object that already exists. Two very important and fundamental programming operations are based on this constructor:

1. Pass by value
2. Return by value

Copy Constructor

- This means that whenever objects are passed to or returned from a function, so long as it is done by value, that the objects are actually copied from one part of the program to another. That is, each operation creates a new object, which must be constructed, and constructing new objects is always the task of a constructor function.
- These operations are so important and so fundamental to programming that the compiler automatically generates a copy constructor for every class. In the situations that we've seen so far, the task of copying an existing object is very easy, easy enough that the compiler can easily create the constructor. Later, we will see more complex situations where the compiler-generated copy constructor is insufficient and we will be forced to override it with our own constructor.

Copy Constructor

- Since the copy constructor is used to implement pass by value, the argument to the copy constructor cannot be passed by value (which would cause infinite recursion). It is for this reason that copy constructors have a very specific signature that makes them easy to identify and which programmers must follow when they need to override the compiler-generated copy constructor. Copy constructors always take a single argument that is the same class type as the class in which the constructor appears, and the argument is always a reference variable:

LECTURE 5

Move Constructor

Move Constructor

- Move a data container from one instance to another.
- A move constructor allows the resources owned by an **rvalue** object to be moved into an **lvalue** without creating its copy.
- An **rvalue** is an expression that does not have any memory address, and an **lvalue** is an expression with a memory address.

What is an Rvalue reference?

RHS value, LHS variable

Before jumping on to the rvalue reference, let's see one of the limitations in C++:

```
int &j = 20;
```

The above code snippet will give an error because, in C++, a variable cannot be referenced to a temporary object. The correct way to do it is to create an rvalue reference using && :

```
int &&j = 20;
```

Temporary objects are often created during the execution of a C++ program.

Copy Constructor

- Make duplicated copy.
- Now, when the code is executed, the default constructor is called at the time that the temporary object A is created. The copy constructor is called as the temporary object of A is pushed back in the vector.
- In the above code, there is a serious performance overhead as the temporary object A is first created in the default constructor and then in the copy constructor.

```cpp
#include <iostream>
#include <vector>
using namespace std;
class A{
  int *ptr;
public:
  A(){
    // Default constructor
    cout << "Calling Default constructor\n";
    ptr = new int ;
  }
  A( const A & obj){
    // Copy Constructor: copy of object is created
    this->ptr = new int;
    // Deep copying
    cout << "Calling Copy constructor\n";
  }
  ~A(){
    // Destructor
    cout << "Calling Destructor\n";
    delete ptr;
  }
};

int main() {
  vector <A> vec;
  vec.push_back(A()); // make copy here. Deep Copy
  return 0;
}
```

copy2.cpp

```
Calling Default constructor
Calling Copy constructor
Calling Destructor
Calling Destructor
```

Move Constructor

- Do not copy. Only copy reference. Shallow Copy
- When the code is executed, the move constructor is called instead of the copy constructor. With the move constructor, the copy of the temporary object of A is avoided. For a large number of push_back statements, using the move constructor is an efficient choice.
- **Note:** a move constructor can be explicitly called, with the move() function, for **already created** objects.

```
class A{
  int *ptr;
public:
  A(){
    // Default constructor
    cout << "Calling Default constructor\n";
    ptr = new int ;
  }
  A( const A & obj){
    // Copy Constructor: copy of object is created
    this->ptr = new int;
    // Deep copying
    cout << "Calling Copy constructor\n";
  }
  A ( A && obj){
    // Move constructor: It will simply shift the resources, without creating a copy.
     cout << "Calling Move constructor\n";
    this->ptr = obj.ptr;
    obj.ptr = NULL;
  }

  ~A(){
    // Destructor
    cout << "Calling Destructor\n";
    delete ptr;
  }
};
```

move2.cpp

```
int main() {
  vector <A> vec;
  vec.push_back(A());
  return 0;
}
```

Calling Default constructor
Calling Move constructor
Calling Destructor
Calling Destructor

LECTURE 6

List Constructor using initializer_list

Initializer lists

- When we want to initialize a vector, we had to do the following:

```
// C++03
vector v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

- Now, with C++ 11, we can do this:

```
// C++11
vector v = { 1, 2, 3, 4 };
```

List Constructor

- C++11 binds the concept to a template, called **`initializer_list`**. This allows constructors and other functions to take initializer-lists as parameters:

List_Constructor.cpp

```
#include <iostream>
#include <vector>
using namespace std;
class MyNumber{
    private:
        vector<int> mVec;
    public:
        MyNumber(const initializer_list<int> &v) {
            for (auto itm : v) {
                mVec.push_back(itm);
            }
        }
        void print() {
            for (auto itm : mVec) {
                cout << itm << " ";
            }
        }
};
int main(){
    MyNumber m = { 1, 2, 3, 4 };
    m.print();  // 1 2 3 4
    return 0;
}
```

```
1 2 3 4
```

Project: algo1.cpp

- MyIntArray Class with List constructor

algo1.cpp

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

class SimpleTimer{
    std::chrono::time_point<std::chrono::steady_clock> start_time;
    public:
    void start(){
        start_time = std::chrono::steady_clock::now();
    }
    double elapsed_seconds() const{
        std::chrono::duration<double> diff(std::chrono::steady_clock::now() -
 start_time);
        return diff.count();
    }
};
```

```
11
22
33
```

algo1.cpp

```
class MyIntArray{
     int *array;
     int size_;
   public:
     MyIntArray(int size): size_(size){
         array = new int[size_];
         for (int i=0; i<size_; i++){
             array[i] = 0;
         }
     }
     MyIntArray(initializer_list<int> ilist){
         size_ = ilist.size();
         array = new int[size_];
         int i=0;
         for (int x: ilist){
            array[i++] = x;
         }
     }
     ~MyIntArray(){
         delete[] array;
     }
     int size(){ return size_; }
     int& operator[](int index){ return array[index]; }
};

int main(){
     MyIntArray a(3);
     a[0] = 11;
     a[1] = 22;
     a[2] = 33;
     for (int i=0; i<a.size(); i++){
        cout << a[i] << endl;
     }
     return 0;
}
```

```
11
22
33
```

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD**[]**, array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy		f(const X&) + f(X&&) & move **	
In & move from	f(X&&) **		

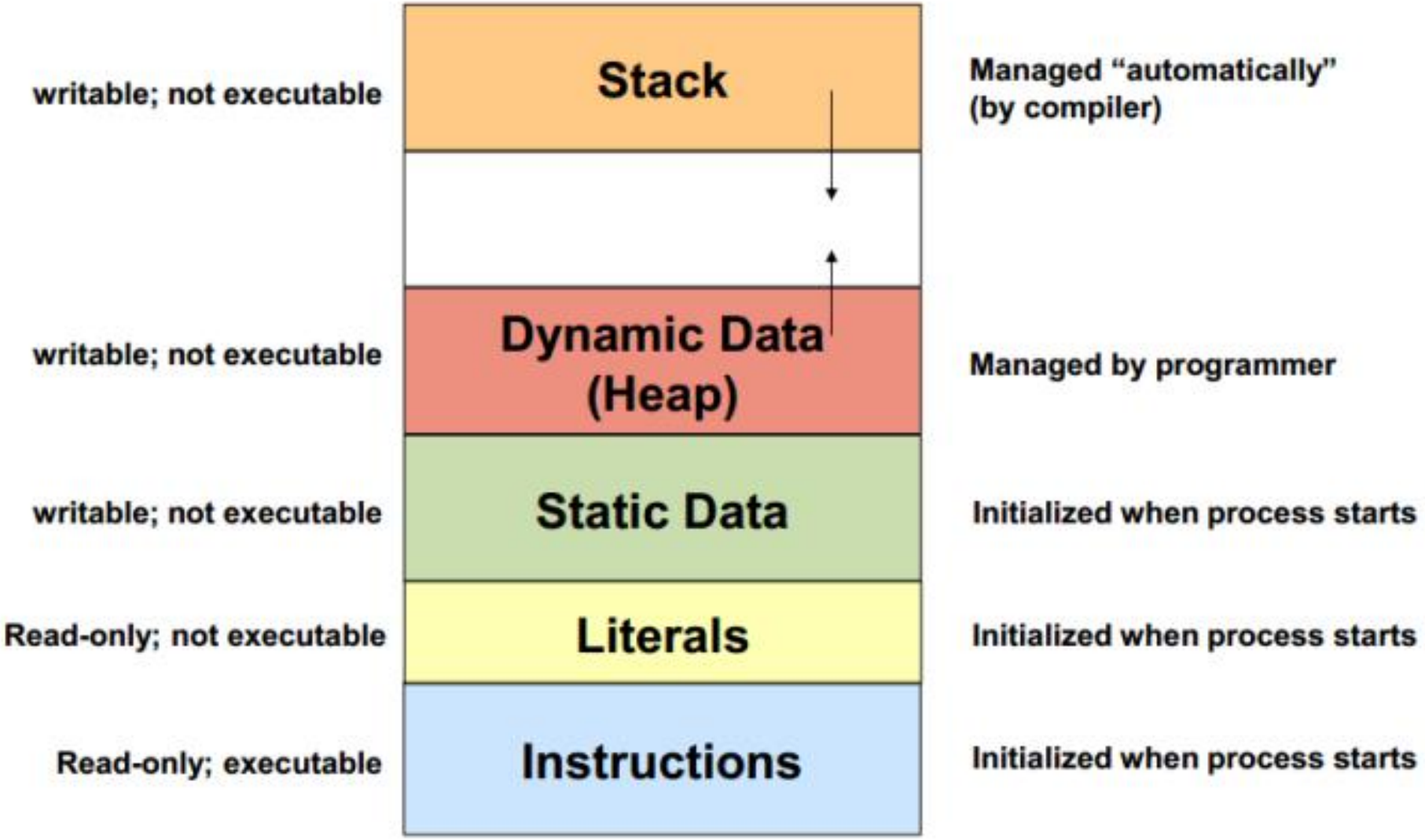
** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

*** special cases can also use perfect forwarding (e.g., multiple in+copy params, conversions)*

LECTURE 7

delete keyword in C++

C++ Memory Model



delete keyword in C++

Delete is an ***operator*** that is used to **destroy array** and **non-array**(pointer) **objects** which are created by new expression.

- Delete can be used by either using ***Delete operator*** or ***Delete [ ] operator***
- New operator is used for dynamic memory allocation which puts variables on heap memory.
- Which means Delete operator deallocates memory from heap.
- Pointer to object is not destroyed, value or memory block pointed by pointer is destroyed.
- The delete operator has **void** return type does not return a value.

[A] Deleting Array Objects

- We delete an array using [] brackets.

dela.cpp

```
// Program to illustrate deletion of array
#include <bits/stdc++.h>
using namespace std;

int main(){
    // Allocate Heap memory
    int* array = new int[10];

    // Deallocate Heap memory
    delete[] array;
    return 0;
}
```

[B] Deleting NULL pointer

- Deleting a NULL does not cause any change and no error.

delb.cpp

```cpp
// C++ program to deleting NULLL pointer
#include <bits/stdc++.h>
using namespace std;

int main(){
    // ptr is NULL pointer
    int* ptr = NULL;
    // deleting ptr
    delete ptr;
    return 0;
}
```

[C] Deleting pointer with or without value

- Deleting dangling pointer or a primitive data.

delc.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;
int main(){
    // Creating int pointer
    int* ptr1 = new int;
    // Initializing pointer with value 20
    int* ptr2 = new int(20);
    cout << "Value of ptr1 = " << *ptr1 << "\n";
    cout << "Value of ptr2 = " << *ptr2 << "\n";
    delete ptr1; // Destroying ptr1
    delete ptr2; // Detroying ptr2
    return 0;
}
```

[D] Deleting a void pointer

- **void pointer** is a generic pointer. **auto pointer** is a local pointer.

deld.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    void* ptr; // Creating void pointer
    delete ptr; // Destroying void pointer
    cout << "ptr deleted successfully";
    return 0;
}
```

[E] deleting memory dynamically allocated by malloc

- Typically **struct**

dele.cpp

```
#include <bits/stdc++.h>
using namespace std;
struct data {
  int a;
  int b;
};
int main(){
    // Dynamic memory allocated by using malloc
    data* ptr2 = (data*)malloc(sizeof(data));
    delete ptr2;
    cout << "ptr2 deleted successfully";
    return 0;
}
```

[E] deleting memory dynamically allocated by malloc

- Typically **struct**

dele.cpp

```
#include <bits/stdc++.h>
using namespace std;
struct data {
  int a;
  int b;
};
int main(){
    // Dynamic memory allocated by using malloc
    data* ptr2 = (data*)malloc(sizeof(data));
    delete ptr2;
    cout << "ptr2 deleted successfully";
    return 0;
}
```

[F] Deleting variables of User Defined data types

- Delete array of **struct**
- Overloading of delete and delete[] operators

delf.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

struct P {
    static void operator delete(void* ptr, std::size_t sz){
        cout << "custom delete for size " << sz <<endl;
        delete (ptr); // ::operator delete(ptr) can also be used
    }
    static void operator delete[](void* ptr, std::size_t sz){
        cout << "custom delete for size " << sz <<endl;
        delete (ptr); // ::operator delete(ptr) can also be used
    }
};

int main(){
    P* var1 = new P;
    delete var1;
    P* var2 = new P[10];
    delete[] var2;
}
```

```
custom delete for size 1
custom delete for size 18
```

Exceptions [A] Trying to delete Non-pointer object

- Only dynamic allocated memory (new or malloc) can be deleted.
- All dynamically allocated memory must be returned to avoid memory leak.

except1.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int x;

    // Delete operator always
    // requires pointer as input
    delete x;
    return 0;
}
```

```
except1.cpp: In function 'int main()':
except1.cpp:9:9: error: type 'int' argument given to 'delete', expected pointer
    9 |  delete x;
      |         ^
```

Exceptions [B] Trying to delete pointer to a local stack allocated variable

- Only dynamic allocated memory (new or malloc) can be deleted.

except2.cpp

Runtime error

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int x;
    int* ptr1 = &x;
    // x is present on stack frame as local variable, only dynamically
    // allocated variables can be destroyed using delete operator
    delete ptr1;
    return 0;
}
```

```
C:\Eric_Chou\Cpp Course\C++ Data Structures and Algorithms\CppDev\Practice\Algorithms>g++ except
2.cpp -o test

C:\Eric_Chou\Cpp Course\C++ Data Structures and Algorithms\CppDev\Practice\Algorithms>test

C:\Eric_Chou\Cpp Course\C++ Data Structures and Algorithms\CppDev\Practice\Algorithms>
```

LECTURE 8

Delete objects when out of scope – Using Unique Pointers

unique_ptr

- unique_ptr is a simple ‘smart’ pointer: **it holds an instance of an object and deletes it when it goes out of scope.** In terms of lifetime behavior it’s much like any regular C++ object with a constructor and destructor, only with dynamic memory allocation. No reference counting, no fancy tricks.
- And that’s the beauty. Simple, elegant & efficient, yet extremely powerful. With **unique_ptr** you no longer have to worry about new and delete. Simply call **make_unique()** (instead of new), and the destructor will call delete automatically. It’s as simple as that, and covers 90% of use-cases that require dynamic memory.

make_unique()

- Above I mentioned make_unique(). This is a new standard function that came in C++14. You may think that it's supposed to save us the need to typing the type we're interested in, similar to make_pair. Well, not quite. Let's look at make_unique()'s signature:

```
// inside namespace std
template <typename T, typename ... Args>
std::unique_ptr<T> make_unique(Args&&... args);
// '&&' here means forwarding references, not necessarily rvalues. I hope to
// explain what these are in a future post.
```

make_unique()

- Note that T is not an argument to the function, thus it can't possible be deduced by the compiler.
- So to use make_unique() one must *always* provide T explicitly, like so:

```
auto u_int = std::make_unique<int>(123);
cout << (*u_int == 123) << endl;
auto u_string =
std::make_unique<std::string>(3, '#');
cout << (*u_string == "###") << endl;
```

make_unique()

```cpp
#include <iostream>
#include <memory>
using namespace std;
int main(){
    unique_ptr<int> xptr = make_unique<int>(123);
    // int *xp = new int(3);
    auto yptr = make_unique<int>(456);
    auto ustr = make_unique<string>(3, '#');
    cout << "*xptr=" << *xptr << endl;
    cout << "*yptr=" << *yptr << endl;
    cout << "*ustr=" << *ustr << endl;
    return 0;
}
```

unique1.cpp

```
*xptr=123
*yptr=456
*ustr=###
```

Project: algo2.cpp

- MyIntArray Class with List constructor
- Use unique_ptr<int[]> as array
- Remove the ~MyIntArray() destructor
- Add copy constructor

algo2.cpp

a[]=11 22 33
b[]=11 2 33

```cpp
#include <iostream>
#include <memory>
#include <chrono>
#include <utility>
using namespace std;

class SimpleTimer{
    std::chrono::time_point<std::chrono::steady_clock> start_time;
    public:
    void start(){
        start_time = std::chrono::steady_clock::now();
    }
    double elapsed_seconds() const{
      std::chrono::duration<double> diff(std::chrono::steady_clock::now()-start_time);
      return diff.count();
    }
};
```

```
class MyIntArray{
    unique_ptr<int[]> array;
    int size_;
  public:
    MyIntArray(int size): size_(size){
        array = make_unique<int[]>(size_);
        for (int i=0; i<size_; i++){
            array[i] = 0;
        }
    }
    MyIntArray(initializer_list<int> ilist){ // list constructor
        size_ = ilist.size();
        array = make_unique<int[]>(size_);
        int i=0;
        for (int x: ilist){
           array[i++] = x;
        }
    }
    MyIntArray(const MyIntArray& other): MyIntArray(other.size_){ // deep copy, create new array and size_
        array = make_unique<int[]>(other.size_);
        size_ = other.size_;
        for (int i=0; i<size_; i++){
            array[i] = other.array[i];
        }
    }
    /*
    ~MyIntArray(){
        delete[] array;
    }
    */
    int size(){ return size_; }
    int& operator[](int index){ return array[index]; }
};
```

```
int main(){
    MyIntArray a(3);
    a[0] = 11;
    a[1] = 22;
    a[2] = 33;
    cout << "a[]=" ;
    for (int i=0; i<a.size(); i++){
       cout << a[i] << " ";
    }
    cout << endl;

    MyIntArray b = a; // deep copy
    b[1] = 2;
    cout << "b[]=" ;
    for (int i=0; i<b.size(); i++){
       cout << b[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Project: algo3.cpp

- MyIntArray Class with List constructor
- Use unique_ptr<int[]> as array
- Remove the ~MyIntArray() destructor
- Add copy constructor
- Use copy constructor and measuring copy time

algo3.cpp

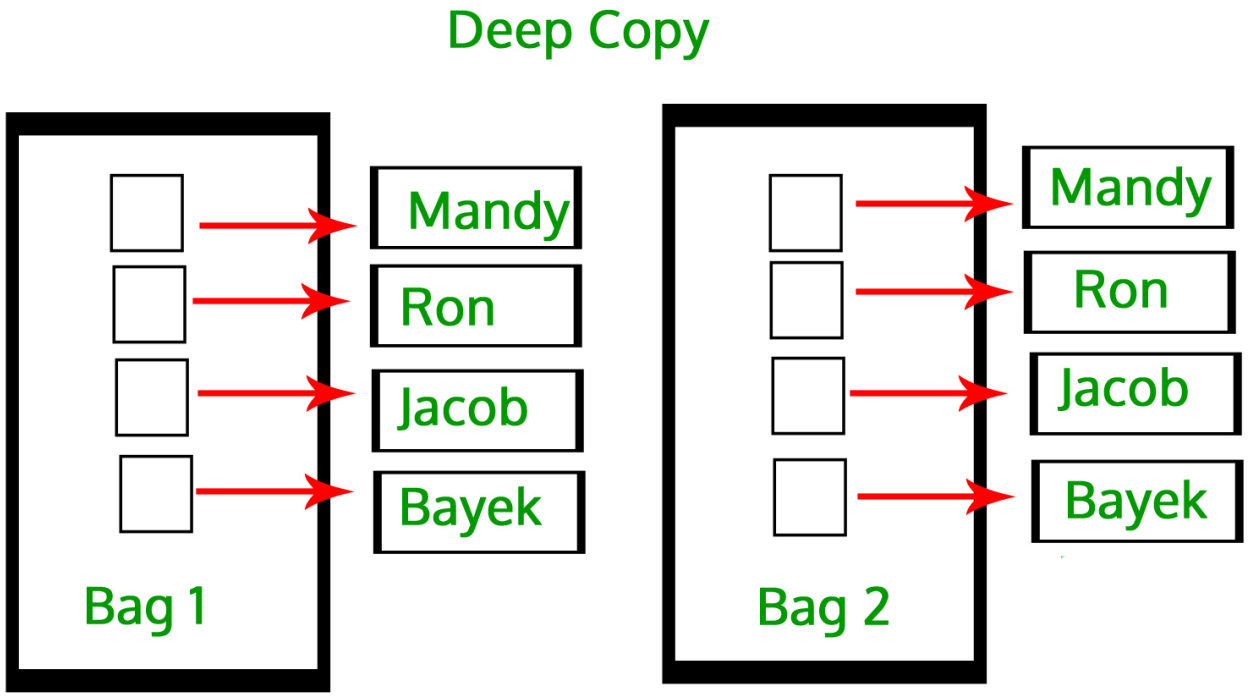
Copy Seconds: 3.49102 secs

```cpp
int main(){
    int n = 1000000;
    SimpleTimer t;
    double total_time = 0;
    for (int i=0; i<100; i++){
        MyIntArray a(n);
        t.start();
        MyIntArray b(a); // deep copy of a
        total_time += t.elapsed_seconds();
    }
    cout << "Copy Seconds: " << total_time << " secs"<< endl;
    return 0;
}
```

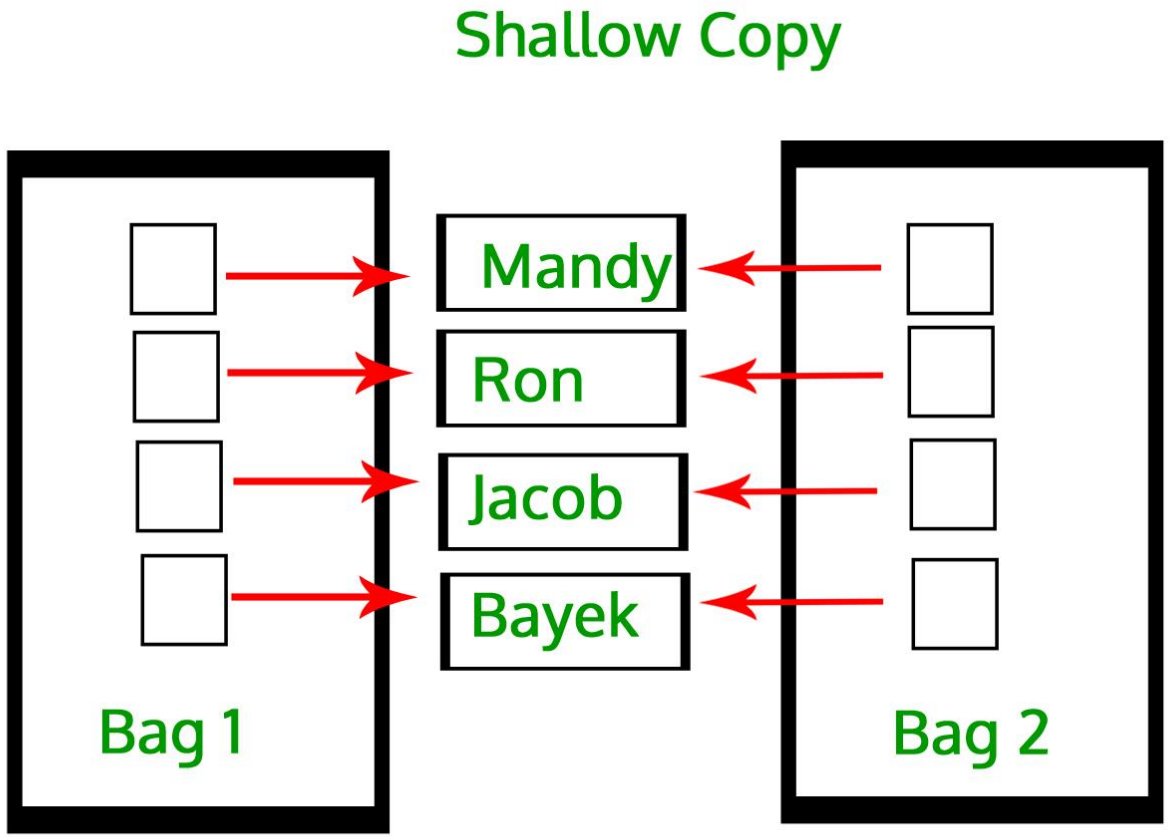
LECTURE 9

Copy Constructor Design: Shallow Copy/Deep Copy

Deep Copy versus Shallow Copy



Deep Copy versus Shallow Copy



Project: algo4.cpp

- MyIntArray Class with List constructor
- Use unique_ptr<int[]> as array
- Remove the ~MyIntArray() destructor
- Add copy constructor
- Use copy constructor and measuring copy time
- Vector version
- Move Constructor

```cpp
int main(){
    int n = 1000000;
    SimpleTimer t;
    double total_time = 0;
    for (int i=0; i<100; i++){
        vector<int> a(n, 0);
        t.start();
        vector<int> b(a);
        total_time += t.elapsed_seconds();
    }
    cout << "Copy Seconds: " << total_time << " secs"<< endl;

    total_time = 0;
    for (int i=0; i<100; i++){
        vector<int> a(n, 0);
        t.start();
        vector<int> b(move(a));
        total_time += t.elapsed_seconds();
    }
    cout << "Move Seconds: " << total_time << " secs"<< endl;
    return 0;
}
```

algo4.cpp

Copy Seconds: 0.0917421 secs

Move Seconds: 1.02e-05 secs

Project: algo5.cpp

- MyIntArray Class with List constructor
- Use unique_ptr<int[]> as array
- Remove the ~MyIntArray() destructor
- Add copy constructor
- Use copy constructor and measuring copy time
- Move Constructor (other will be destroyed)

Adding Move Constructor

```
MyIntArray(MyIntArray&& other): array(move(other.array)),
                                size_(other.size_){
        other.array.release();
        other.size_ = 0;
    }
```

```cpp
int main(){
    int n = 1000000;
    SimpleTimer t;
    double total_time = 0;
    for (int i=0; i<100; i++){
        MyIntArray a(n);
        t.start();
        MyIntArray b(a); // deep copy of a
        total_time += t.elapsed_seconds();
    }
    cout << "Copy Seconds: " << total_time << " secs"<< endl;
    total_time = 0;
    for (int i=0; i<100; i++){
        MyIntArray a(n);
        t.start();
        MyIntArray b(move(a)); // shallow copy of a
        total_time += t.elapsed_seconds();
    }
    cout << "Move Seconds: " << total_time << " secs"<< endl;
    return 0;
}
```

algo5.cpp

Copy Seconds: 3.72958 secs

Move Seconds: 5.19e-05 secs

SHALLOW COPY
VERSUS
DEEP COPY

SHALLOW COPY	DEEP COPY
Process of constructing a new collection object and then populating it with references to the child objects found in the original	Process of constructing a new collection object and then recursively populating it with copies of the child object found in the original
Does not recurse and it does not create copies of the child objects themselves	Creates a complete independent clone of the original object and all of its children
Shallow copy is not recursive	Deep copy is recursive

Visit www.PEDIAA.com

LECTURE 10

Segmentation Faults

Core Dump (Segmentation fault) in C/C++

Core Dump/Segmentation fault is a specific kind of error caused by accessing memory that “does not belong to you.”

- When a piece of code tries to do read and write operation in a read only location in memory or freed block of memory, it is known as core dump.
- It is an error indicating memory corruption.

Common segmentation fault scenarios

- Modifying a string literal
- Accessing an address that is freed
- Accessing out of array index bounds
- Improper use of scanf()
- Stack Overflow
- Dereferencing uninitialized pointer

[A] Modifying a string literal

- The below program may crash (gives segmentation fault error) because the line *(str+1) = ‘n’ tries to write a read only memory.

fault1.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    char *str;

    /* Stored in read only part of data segment */
    str = "GfG";

    /* Problem: trying to modify read only memory */
    *(str+1) = 'n';
    return 0;
}
```

[B] Accessing an address that is freed

- Here in the below code, the pointer p is dereferenced after freeing the memory block, which is not allowed by the compiler. So, it produces the error segment fault or abnormal program termination at runtime.

fault2.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;
#include <cstdlib>
int main(void){
    // allocating memory to p
    int* p = (int *) malloc(8);
    *p = 100;
    // deallocated the space allocated to p
    free(p);
    // core dump/segmentation fault
    // as now this statement is illegal
    *p = 110;
    return 0;
}
```

[C] Accessing out of array index bounds

- Accessing out of array index bounds

fault3.cpp

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

int main(){
    int arr[2];
    arr[3] = 10; // Accessing out of bound
    return 0;
}
```

[D] Improper use of scanf()

- scanf() function expects address of a variable as an input.Here in this program n takes
- value of 2 and assume it’s address as 1000. If we pass n to scanf(), input fetched from STDIN is placed in invalid memory 2 which should be 1000 instead.It’s a memory corruption leading to Seg fault.

fault4.cpp

```cpp
#include <cstdio>

int main(){
    int n = 2;
    scanf("%d",n);
    return 0;
}
```

[E] Stack Overflow

- It’s not a pointer related problem even code may not have single pointer. It’s because of recursive function gets called repeatedly which eats up all the stack memory resulting in stack overflow. Running out of memory on the stack is also a type of memory corruption. It can be resolved by having a base condition to return from the recursive function.

[F] Dereferencing uninitialized pointer

- A pointer must point to valid memory before accessing it.

fault6.cpp

```cpp
#include <cstdio>

int main(){
    int *p;
    printf("%d",*p);
    return 0;
}
```