LECTURE 4

ARRAY IMPLEMENTATIONS AND MEMORY MANAGEMENT

TODAY'S CLASS:

- Conclude Recurrsion
- Array Implementation of ADT Bag and Memory Management
- Introduce ICE 3 and P2.

Do we always want to use recursion?

- There are limitations here – every recursive call requires memory for the function call and its variables on the stack...
- For very deep recursion, this will cause a stack overflow. For less powerful hardware, this happens much sooner...
- The language matters – Haskell and Lisp are designed to use recursion for nearly everything! They use lazy evaluation to keep things fast. Some languages handle recursion more efficiently than others...
- So we have a tradeoff here: if recursion offers a sufficiently better solution, we should try to use it if we have the memory to do so... can the compiler help?

What if we are given a recursive function?

- Many C++ compilers have an optimization called tail-call optimization
- A tail-call-recursive function has the following properties:
  - There is exactly one recursive call
  - The recursive call is the last statement in the function
- In this case, the compiler (or you!) can remove recursion by using a loop:
  - The tail recursive call makes some change and calls itself
  - Instead, make the change locally, then jump back to the top of the function
  - Only one stack frame is needed, and the recursive function can perform as in iterative function

Example: Tail-Call Optimization

- There is a compiler flag you can pass to tell the compiler to look for and remove tail recursion
- Here we have a rough approximation of what the compiler does at the assembly level
- If you find yourself writing tail-call-recursive functions, think carefully about why recursion is justified…

```
int factorial(int n){
    int t = 1; // set up a return value
    mylabel:   // set up a goto target
    if(n == 1){
        return t;
    }
    t *= n;
    n--;
    goto mylabel;

}
```

Example... (see section 2.4.2 in Carrano)

- Consider how we might search a sorted list:
  - If the value we are looking for is greater than the middle value, do we need to search the lower half of the array?

```
int find(int a[], int lwr, int upr, int val){

    /*
     * Given a sorted integer array
     * of size n, return the index
     * of the first position which
     * matches val, or -1 if it is
     * not in the array
     */
    return -1;
}
```

Review of the ADT Concept

- We already saw the benefits of describing our collection of data, and ways to operate on it:
  - The user does not need to know or mind the implementation details
  - The user cannot inadvertently cause errors or data loss
- Implementation is up to us as programmers, but we need to be careful:
  - What functionality will we expose to make our ADT useful?
  - What core operations are needed?
  - What state do we need to maintain to support those operations?

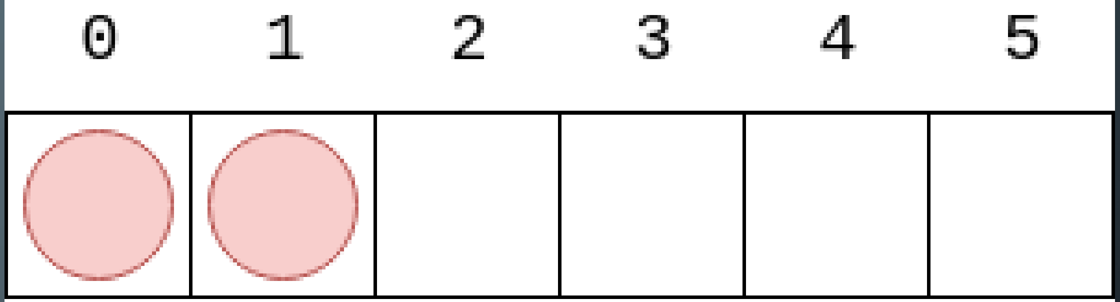
Case Study: An array implementation for ADT bags

- A common task requires maintaining a collection of data storage items
  - Our ADT bag fits the bill, but what changes are required?
  - How many items can we hold?
  - How do we organize them?
- These are design questions, and to answer them, we need to consider the core functionality we want to provide

ADT Bag
+ getCurrentSize: integer
+ isEmtpy: boolean
+ add(newEntry: T): bool
+ remove(anEntry: T): bool
+ clear:void
+ getFrequencyOf(anEntry: T): integer
+ contains(anEntry: T): bool
+ toVector: vector

Case Study: An array implementation for ADT bags

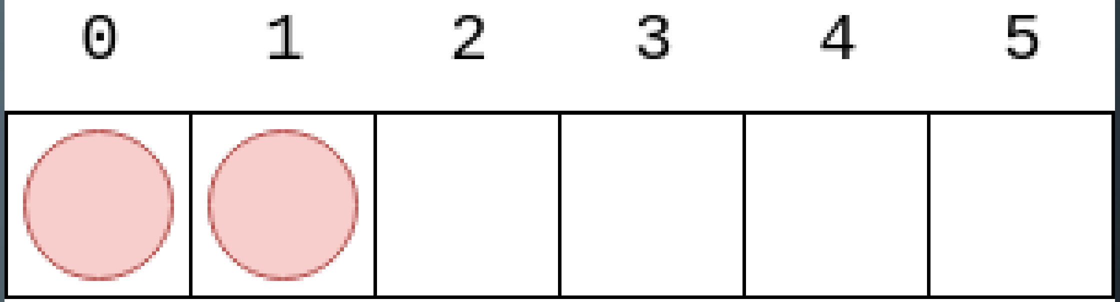
- We want to store data,
  - process of adding and accessing data is fundamental
- To organize our data
  - enumerate each element
  - we can jump directly to value we care about
- We need to track positions for data
- Track how many positions occupied




```
static const int CAPACITY = 6;
// Internal Array Data Struct.
T items[CAPACITY];
// Current count of items
int itemCount;
// Maximum capacity for items
int maxItems;
```

Case Study: An array implementation for ADT bags

- Assumption: items always added to first empty position
- What makes information sufficient for adding?
  - We know the count of items
  - Next item → `itemCount+1`
  - We know when reached capacity
    - Can't accept new elements




```
static const int CAPACITY = 6;
// Internal Array Data Struct.
T items[CAPACITY];
// Current count of items
int itemCount;
// Maximum capacity for items
int maxItems;
```

Anatomy of the Abstract Bag Interface

- Template keywords tell the compiler type T is generic
  - Classes which implement this interface preserve this
  - Instances must define which type T is
- virtual keyword prevents the abstract interface itself from being instantiated.
  - A concrete class implements the interface!

```
template<typename T>
class AbstractBag
{
public:

  // add an item to the bag
  virtual bool add(const T & item) = 0;

  // remove an item
  virtual bool remove(const T & item) = 0;

  // check is the bag is empty
  virtual bool isEmpty() const = 0;

  // get number of items in the bag
  virtual std::size_t getCurrentSize() const = 0;

  // clear the bag contents
  virtual void clear() = 0;

  // count how many time item occurs in bag
  virtual std::size_t getFrequencyOf(const T & item) const = 0;

  // check if item is in the bag
  virtual bool contains(const T& item) const = 0;
};

#endif
```

A Class Implements an Interface…

- Our concrete implementation limited_size_bag.hpp has a function to match each function in the interface
- Again, we have carried the generic T through to our implementation

```
#ifndef LIMITED_SIZE_BAG_HPP
#define LIMITED_SIZE_BAG_HPP

Template<typename T>
class Limited_Size_Bag : public AbstractBag<T>{
public:
        Limited_Size_Bag();
        ~Limited_Size_Bag();
        bool add(const T &item);
        bool remove(const T &item);
        bool isEmpty() const;
        std::size_t getCurrentSize() const;
        void clear();
        std::size_t getFrequencyOf(const T &item) const;
        bool contains(const T &item) const;

};


#endif
```

Question 1: Testing Revisited

- When should we write tests for our code?
    A. When we have finished writing each method
    B. In the morning on the day the assignment is due
    C. When we are defining our methods
    D. When we are defining our ADT
    E. Both C and D

Testing Revisited

- Unit tests are critical to developing useful and robust code
  - Test early, test often, test automatically
- Interfaces provide a perfect template for unit testing
  - Interface is a contract – guarantees certain behavior
  - Tests prove that guarantee
- Think about method tests before coding the method.
  - What inputs are valid
  - What outputs are valid
  - What assumptions are made *and how might they fail*?

QUESTION 2:

Suppose I have a class Foo. What C++ code will dynamically allocate an object my_foo of type Foo using the default constructor?

1. `Foo my_foo = new Foo;`
2. `Foo* my_foo = new Foo;`
3. `Foo* my_foo = new Foo();`
4. `Foo my_foo;`

QUESTION 3:

What C++ code will free the memory allocated in question 2?

1. `delete Foo;`
2. `delete my_foo;`
3. `delete [] my_foo;`
4. `my_foo.delete();`

QUESTION 4:

Given the same Foo class, what C++ code will dynamically allocate an array of 10 Foo objects my_foo_array?

1. `Foo * my_foo_array = new Foo[10];`
2. `Foo my_foo_array[10];`
3. `my_foo_array = new Foo(10);`
4. `Foo my_foo_array[] = new Foo[10];`

QUESTION 5:

What C++ code will free the memory allocated in question 5?

1. `delete my_foo_array;`
2. `delete [] my_foo_array;`
3. `delete(my_foo_array,10);`
4. `delete [] Foo;`

Review of C++ Memory

- There are three types of memory in a general C++ program:
  - Static Memory: The compiler configures space set aside for constants, global variables, and anything with the static qualifier for your program. The space is predefined and immutable, set up at runtime by the OS
  - Automatic Memory: Space for function calls, their parameters, local variables, and pointers is automatically set aside on the stack as you go
  - Dynamic Memory: Space is set aside for objects, variables, when you specify. *You are responsible for allocating and freeing the memory*

Review of C++ Memory

- Automatic memory becomes invalid as soon as the scope is exited
- We use the heap as a free store for data we wish to share across scopes
- In the example method, we return a copy of a pointer to data on the heap
- Data with the new specifier is allocated on the heap, and we use delete or delete[] to deallocate it

```
<main>

int myInt;
int *myIntPtr;
myIntPtr = giveIntPlz();
```

```
<giveIntPlz>

int localInt = 5;
// Bad!
//return &localInt;
int *heapInt = new int[1];
*heapInt = 5;
return heapInt;
```

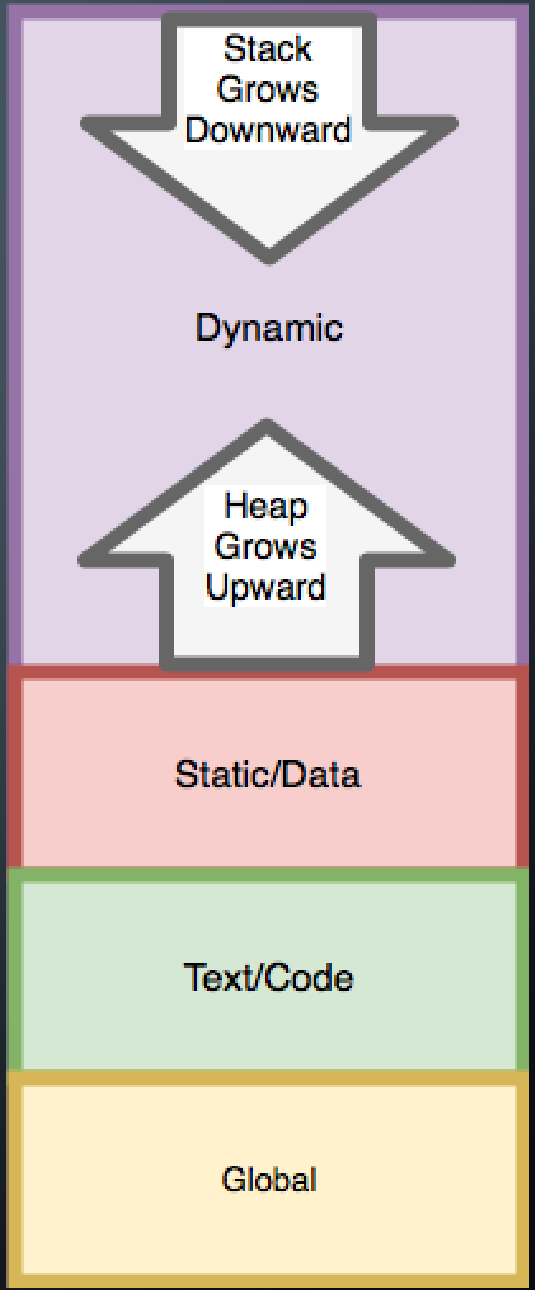
```
<main>

// All is well
cout << *myIntPtr << endl;

delete myIntPtr;
myIntPtr = nullptr;
```

Memory Management : Size Limitations

- In your programs, you will need to manage your memory carefully
- The OS lets you pretend there is 4GB of memory at your disposal
    - In reality, all programs share memory
    - The OS will limit the amount of stack you get for automatic memory
    - You can request heap memory, but you have to do so explicitly



Memory Management : Scope

- The OS will keep an eye on your address space, and if you try to access something outside your stack frame, you get a `segfault`
- Using dynamic memory assigns your local pointer to point to other stack frames or address spaces – now, you can pass around the pointer and access memory local to another scope
- With great power comes great responsibility:
  - You are responsible for the memory you allocate – memory leaks are bad!
  - Passing a pointer to dynamic memory gives unfettered access!
  - Const functions and pointers to const data tell the compiler to make the data read-only

Memory Management : Syntax

- Allocate any class or built-in type with the new keyword:
  - `float *dynamicPi = new float(3.14159);`
  - `int *myList = new int[5];`
  - `Bag<int> *myArray = new Bag<int>;`
- Deallocate individually, or with the array operator:
  - `delete myArray;`
  - `delete [] myList;`
  - `delete dynamicPi;`

Memory Management : Interfaces

- So far, our interfaces have not included a destructor
  - By default, all C++ objects (class instances) have a default constructor
  - If we do not explicitly define a destructor, the default is used

- Why is this a problem for an implementation of an interface?

Memory Management : Interfaces

- Case study: Bag Implementation
  - A HeapBag100 is an implementation of the AbstractBag interface
  - The interface does not require an implementation of a destructor!
    - The default destructor is called when using an AbstractBag pointer!

```
template<typename T>
class AbstractBag {
public:
     virtual bool add(const T & item) = 0;
     virtual bool remove(const T & item) = 0;
     virtual bool isEmpty() const = 0;
};

template<typename T> class HeapBag100 : public
AbstractBag<T> {
public:
  HeapBag100(){
     position = 0;
     data = new T[100]; }
  ~HeapBag100() {
     delete [] data;
  }

. . .
private:
  T *data;
  int position;
};

int main(){
  AbstractBag<int> * myBag = new HeapBag100<int>();
  myBag->add(1);
  delete myBag; // ahh…man!
}
```

Memory Management : Interfaces

- Case study: Bag Implementation
  - A HeapBag100 is an implementation of the AbstractBag interface
  - The interface does not require an implementation of a destructor!
    - The default destructor is called when using an AbstractBag pointer!
  - Include a purely virtual destructor
    - Force using of the implementing class destructor!

```
template<typename T>
class AbstractBag {
public:
      virtual ~AbstractBag()=0; // must have a body
      virtual bool add(const T & item) = 0;
      virtual bool remove(const T & item) = 0;
      virtual bool isEmpty() const = 0;
};

template<typename T> class HeapBag100 : public
AbstractBag<T> {
public:
  HeapBag100(){
     position = 0;
     data = new T[100]; }
  ~HeapBag100() {
     delete [] data;
  }
. . .
private:
  T *data;
  int position;
};
// called after ~HeapBag100()
AbstractBag::~AbstractBag() {}

int main(){
  AbstractBag<int> * myBag = new HeapBag100();
  myBag.add(1);
  delete myBag; // now I'm good!
}
```

ONE OF THE ADVANTAGES OF C++ OVER C IS THE INCREASED EASE IN CONTROLLING OBJECT STATE

- Variables have a lifetime.
  - The lifetime of a (automatic) stack allocated variable is determined by it's scope.
  - For dynamically allocated variables the lifetime is bounded by birth (Construction) and death (Destruction).
- C++ Rule of three: If you write a (non-trivial) destructor you should also write a
  - copy constructor and
  - copy assignment operator

Assignment/Homework

- Reading: Carrano pp. 133 -155
- P1 is due Today.
- ICE3 Released, due Thursday
- P2 will be released later today, due June 10th.

In-Class Exercise 4

- "Egyptian Powers" is an algorithm that computes powers by a recurrence relation
- Use the relationship shown to derive a recursive version

```
int fancyPower(int n, int x){
    // Your code here...
}
```

For even values of n:

$$x^n = (x^2)^{\frac{n}{2}}$$

For odd values of n:

$$x^n = x \cdot (x^2)^{\frac{n-1}{2}}$$