Classes and ADTs

Sandra Batista, Mark Redekopp, and David Kempe

1.1–1.2

New Paradigm: RAII

Resource Acquisition is Initialization (RAII)

1. Acquire resources in constructor for objects

2. Release the resources in the matching destructor

Key point: Use C++ classes that manage resources for programmers (whenever permissible)

Examples:

1. iostreams for I/IO buffers (e.g. cin, cout, cerr)

2. C++ Strings for character buffers

3. STL vector for variable sized array

4. STL containers such as vector, map, unordered_map, list, stack and queue

5. fstreams for files

```
struct Item {
  int val;  Item* next;
};
class LinkedList {
  public:
//destroys items when list is
//out of scope.
  ~LinkedList();
   // create a new item
   // in the list
   void push_back(int v);
  private:
   Item* head;
};

int main()
{
  doTask();
}
void doTask()
{
  LinkedList y;
  y.push_back(3);
  y.push_back(5);
  /* other stuff */
}
```

RAII for Memory

- The **<memory> library** contains classes for managing pointers: **unique_ptr, shared_ptr, weak_ptr**
- These ptr classes are abstractions for memory management.
- The ptr objects hold raw pointers and can be used syntactically like built-in raw pointers
- Unique_ptr and shared_ptr will destroy memory that it points to when it goes out of scope or no longer used.

Unique_ptr

- **std::unique_ptr<type> is for *exclusive ownership* of memory at address.**
- **Only one std::unique_ptr can own a raw pointer (or physical memory address).**
- As a result, unique_ptrs can be moved or returned from functions transferring ownership of the raw pointers.
- Unique_ptrs *cannot be copied or assigned* because two unique_ptrs cannot own same raw pointer.
- Unique_ptrs automatically destroy memory contained in their raw pointers when destroyed using delete by default

Unique_ptr Declaration

Instantiate a unique_ptr<type> using constructor and new for the type.

Preferably instantiate unique_ptr<type> using make_unique

Use the unique_ptr as you would a raw pointer on the object.

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;

struct Student {
   int id;
   string name;
   Student():id(0), name(""){}
   Student(int i, string n): id(i), name(n){}
};

int main(){
     unique_ptr<Student> sp(new Student(1234, "Jane Doe"));

     unique_ptr<Student> sp2 = make_unique<Student>(2468, "John Clark");

     cout<< "First student ID and name: " << sp->id << " " << sp->name << endl;
     cout<< "Second student ID and name: " << sp2->id << " " << sp2->name << endl;

     return 0;
}
```

Transferring Unique_ptrs

Return a unique_ptr<type> from a function

Use std::move to move the unique pointer

Use the reset function of the unique pointer with a raw address

The release function of a unique pointer returns its raw address and sets it to nullptr

```
// include <iostream>, <string>, <memory> and using namespace std;

unique_ptr<Student> add(int ID, string name);

int main(){
    unique_ptr<Student> sp2 = add(12345, "Jane Doe");   //returned from function


    unique_ptr<Student> sp3 = move(sp2);   // sp2 is set to nullptr and the raw pointer is in sp3

    cout<< "student ID and name: " << sp3->id << " " << sp3->name << endl;

    //sp2 = sp3; /*will not compile cannot assign*/

    sp2.reset(sp3.release());

    cout<< "student ID and name: " << sp2->id << " " << sp2->name << endl;         return 0;}


unique_ptr<Student> add(int ID, string name){
          unique_ptr<Student> s = make_unique<Student>(ID,name);
          // unique_ptr<Student> no_copy = s;   /* will not compile cannot copy*/
          return s;
}
```

Dynamic Arrays using Unique_ptr Declaration

Instantiate a unique_ptr<type[]> using new for the type.

Preferably instantiate unique_ptr<type[]> using make_unique

Use the unique_ptr to the array with the subscript operator, operator **[]**

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;

int main(){
   unique_ptr<int[]> int_array(new int[5]);
   for (size_t i =0; i < 5;i++)  int_array[i] = (i+1)*2;

   unique_ptr<string[]>  s_array = make_unique<string[]>(3);
   s_array[0] = "cat";
   s_array[1] = "bird";
   s_array[2] = "dog";

   for (size_t j =0; j <3 ;j++) cout <<s_array[j] << endl;

   int_array.reset(new int[10]);

    return 0;
}
```

Recommended References for Dynamic Memory

1. Course Lecture Notes Chapter 2 (http://david-kempe.com/teaching/DataStructures.pdf)
2. Lippman,Moo, and Lajoie. C++ Primer. Chapter 12 only sections on unique pointers and dynamic arrays. Available for free from USC library and includes practice exercises. You may skip sections on shared_ptrs and exceptions as we will get back to those in a few weeks. You need only focus on sections 12.1.2, 12.1.5, and 12.2.1 https://uosc.primo.exlibrisgroup.com/permalink/01USC_INST/273cgt/cdi_askewsholts_vlebooks_9780133053036

ABSTRACT DATA TYPES

Abstract Data Types

An abstract data type or ADT is a set of values and collection of operations on those values accessed *only through an **interface.***

A **data structure** is a *representation of the values* and *implementation of the operations on the values* of an ADT.

Public functions of C++ classes, especially abstract classes, permit us to specify interfaces.

ADT interfaces define “a contract” between users or *clients* and implementors of ADTs [RS, pg 138].

Reference: *Robert Sedgewick. 1998. Algorithms in c++, parts 1-4: fundamentals, data structure, sorting, searching, third edition (Third. ed.). Addison-Wesley Professional [RS]*

List

- 2 specialized List ADTs:
- Queues
- Stacks

Dictionary/Map

Set

Ordered collection of items,

- Each item has an index and there is a front and back (start and end)
- Duplicates allowed (i.e. in a list of integers, the value 0 could appear multiple times)
- Accessed based on their position (list[0], list[1], etc.)

What are some operations you perform on a list?

List Operations

Operation	Description	Input(s)	Output(s)
insert	Add a new value at a particular location shifting others back	Index	
remove	Remove value at the given location	Index	Value at location
get	Get value at given location	Index	Value at location
set	Changes the value at a given location	Index & Value	
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		Size_t
push_back	Add a new value to the end of the list	Value	
find	Return the location of a given value	Value	Index

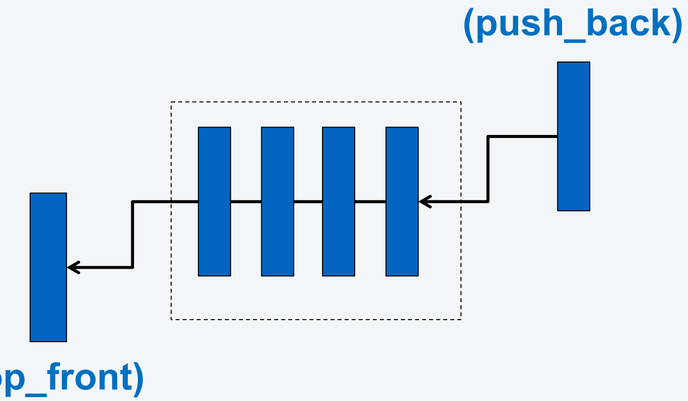
Queues and Stacks

Two specialized List ADTs

Queue (FIFO)

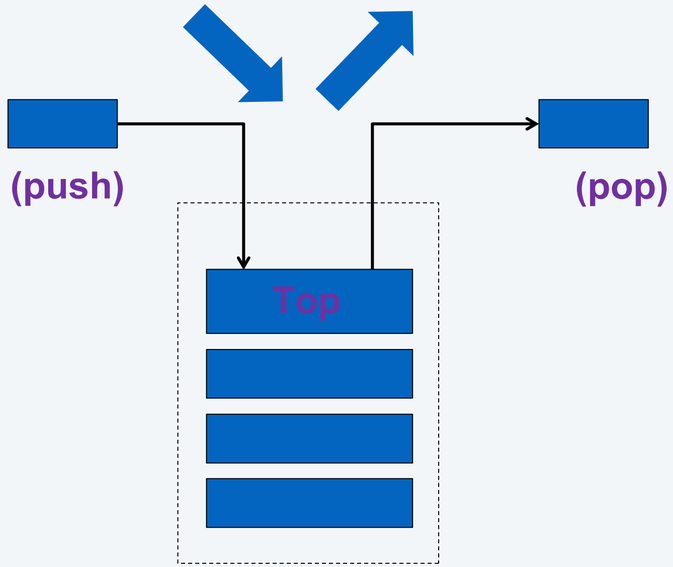
Items leave from the front (pop_front)

Items enter at the back (push_back)



Stack (LIFO)

Items enter and leave from the same side (i.e. the top)



Queue & Stack Operations

Queues

Operations Relative to Lists	Notes
~~insert~~	
~~remove~~	
front	Can only get front item
set	
empty	
size	
push_back	Add to one side
pop_front	Remove from the other

Stacks

Operations Relative to Lists	Notes
~~insert~~	
~~remove~~	
top	Can only get top item
set	
empty	
size	
push	Add to one size
pop	Remove from the same

STL Container Classes

Container classes are classes used simply for storing other items such as a linked list of integers.

C++ Standard Template Library provides implementations of all these containers such as

- DynamicArrayList => C++: std::vector<T>
- LinkedList => C++: std::list<T>
- Deques => C++: std::deque<T>
- Sets => C++: std::set<T>
- Maps => C++: std::map<K,V>

OVERVIEW AND CONCEPTS

Structs vs. Classes

The struct has changed in C++!

- In C

  Only data members

- In C++

  Like a class (data + member functions)

*Default access is **public** access whereas class default to **private** access*

```
struct Person{
  string name;
  int age;
};

int main()
{
  // Anyone can modify
  // b/c members are public
  Person p1;

  p1.name = “Jake Doe”;
  p1.age = 25;


  return 0;
}
```

Classes & Object Oriented Design

Encapsulation

- Place data and operations on data into one logical unit
- Protect who can access data via private members

Abstraction

- Depend only on an **interface** regardless of implementation to create low degree of *coupling* between different components

Unit of composition

- Combined to create larger applications
- Delegation of responsibility

Polymorphism & Inheritance

Protect yourself from users & protect your users from themselves

deck.h

```
class Deck {
 public:
   Deck();   // Constructor
   ~Deck();  // Destructor
   void shuffle();
   void cut();
   int get_top_card();
 private:
   int cards[52];
   int top_index;
};
```

cardgame.cpp

```
#include<iostream>
#include “deck.h”

int main(int argc, char *argv[]) {
  Deck d;
  int hand[5];

  d.shuffle();
  d.cut();

  d.cards[0] = ACE; //won’t compile
  d.top_index = 5;  //won’t compile
}
```

Coupling

Coupling refers to how much components depend on knowledge of each other's implementation details

OO Design seeks to reduce coupling as much as possible by

- Creating well-defined interfaces to update (write) or access (read) the state of an object
- Allow alternate implementations that do NOT require interface changes
- Goal: Carefully designed interfaces that hide implementation details, so underlying implementations can be changed **without needing to change code** that use the interfaces
- Abstract classes are often used to specify such interfaces

PARTS OF A CLASS

Parts of a C++ Class

What are the main parts of a class?

- Data members

  What data is needed to represent the object?

- Constructor(s)

  How do you build an instance?

- Member functions

  How does the user need to interact with the stored data?

- Destructor

  How do you clean up an after an instance?

```
class GroceryList {
   public:
   GroceryList();
   GroceryList(size_t n);
   ~GroceryList();
   void addItem(string item);
   void removeItem(string item);
   void merge(const GroceryList& other);
   const string& getItem(size_t position) const;
   void printList() const;
   bool empty() const;
   size_t size()const;

   private:
   size_t num_items;
   size_t list_size;
   unique_ptr<string[]> list;
   void doubleList();
};
```

Notes About Classes

Member data can be public or private (and later protected)

- Default is private (only class functions can access)
- Must explicitly declare something public

Most common C++ operators will not work by default (e.g. ==, +, <<, >>, etc.)

Classes may be used just like any other data type (e.g. int)

- Get pointers or references to them
- Pass them to functions
- Dynamically allocate them
- Return them from functions

C++ Classes: Constructors

Called when an object of a class is instantiated

No return value

Default Constructor

- Has the name ClassName()
- The default constructor is empty
- For arrays, a default constructor is called for each array element

Overloading Constructors

- Additional constructors can take arguments for initialization
- **Appropriate version is called based on how many and what type of arguments are passed when a particular object is created**
- Good practice tip: include a default constructor in your classes when you include constructors that take arguments

```
GroceryList::GroceryList(){
    num_items = 0;
    list_size = 5;
    list = make_unique<string []>(5);
}
GroceryList::GroceryList(size_t n){
    num_items = 0;
    list_size = n;
    list = make_unique<string []>(n);
}
```

Identify that Constructor

Prototype these constructors:

s1

- `string::_____`

s2

- `string::_____`

dat

- `vector<int>::_____`

```
#include <string>
#include <vector>
using namespace std;

int main()
{
  string s1;
  string s2("abc");

  vector<int> dat(30);

  return 0;
}
```

Identify that Constructor

s1

- `string::string()`
  `// default constructor`

s2

- `string::string(const char* )`

dat

- `vector<int>::vector<int>( int );`

```
#include <string>
#include <vector>
using namespace std;

int main()
{
  string s1;
  string s2("abc");

  vector<int> dat(30);

  return 0;
}
```

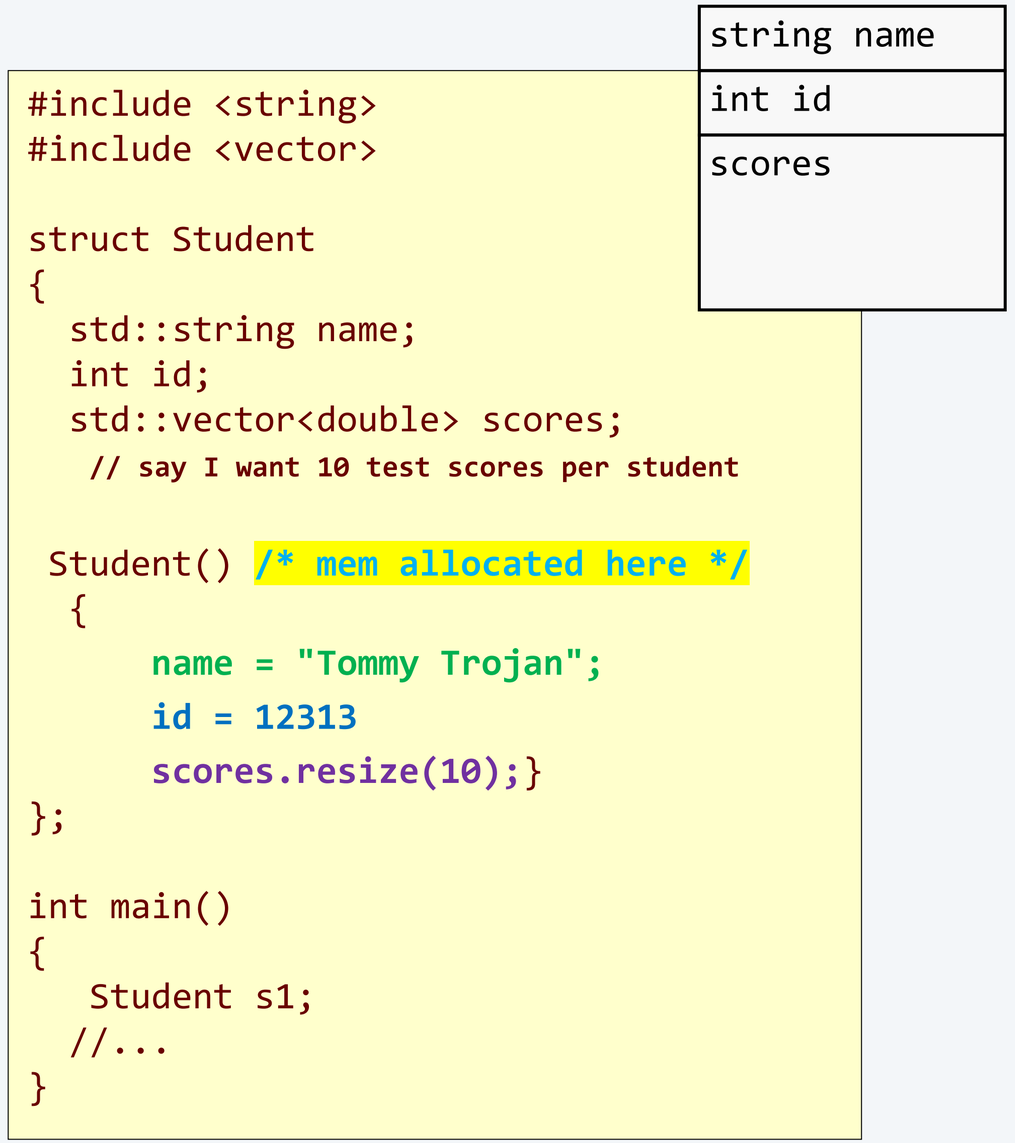
CONSTRUCTOR INITIALIZATION LISTS

Object Construction

Memory for an object is allocated before '{' of the constructor code

Constructors for objects *ONLY EVER* get called at the time memory is allocated

By default for each data member, its default constructor is called to allocate memory for it.



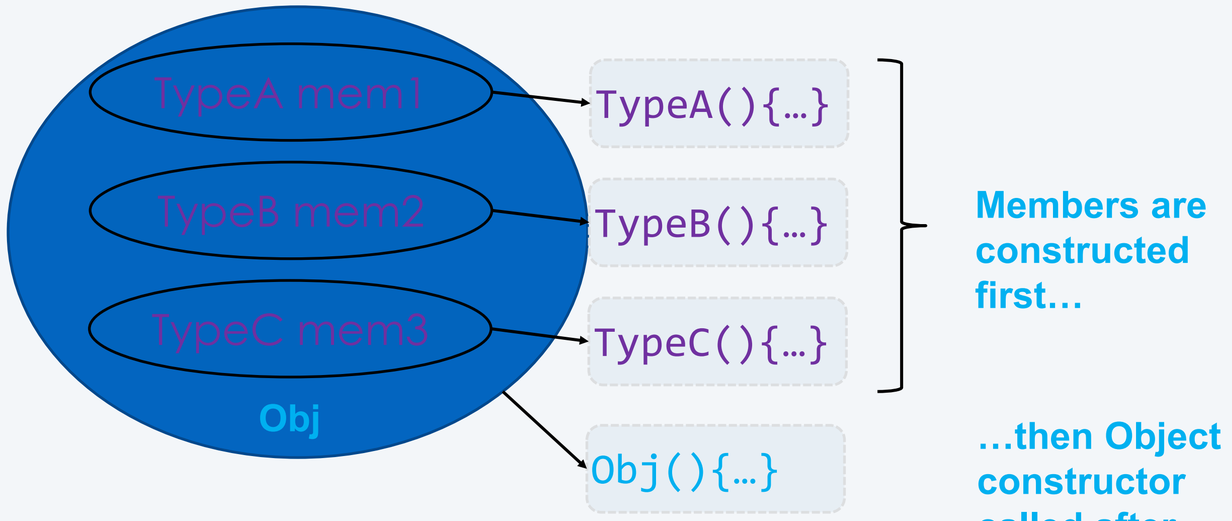

```
#include <string>
#include <vector>

struct Student
{
  std::string name;
  int id;
  std::vector<double> scores;
   // say I want 10 test scores per student

 Student() /* mem allocated here */
  {
      name = "Tommy Trojan";
      id = 12313
      scores.resize(10);}
};

int main()
{
   Student s1;
  //...
}
```

Initializing Members

To recap: When an object is constructed the individual members are constructed first

- Member constructors are called BEFORE object's constructor

```
Class Obj
{ public:
   Obj();
   // public members
  private:
   TypeA mem1;
   TypeB mem2;
   TypeC mem3;
};
```



Calling default constructors

```
Student::Student()
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

If you write this…

```
Student::Student() :
    name(), id(), scores()
    // calls to default constructors
{
  name = "Tommy Trojan"; // now modify
  id = 12313
  scores.resize(10);
}
```

The compiler will still generate this.

The default constructors are called for each data member before entering the {…} to create memory for the data member

Then the data is initialized within the constructor

This is a **2-step** process:

1. Call default constructor for each data member
2. Initialize the value of each data member

Good Practice: Use Constructor Initialization List

```
Student::Student() :
    name("Tommy"), id(12313), scores(10)
{
}
```

A C++ **constructor initialization list has the following format:**

- Constructor(param_list) **:** member1(param/val), …, memberN(param/val) { … }

- If initial values are known and appropriate constructors exist, the initialization list will call the constructors with arguments
- For the data member name: the string constructor with the argument "Tommy"
- For the data member id, an int is created with value 12313
- For the vector scores, the vector constructor with argument 10.

Constructor Initialization List Example

```
GroceryList::GroceryList(){
     num_items = 0;
     list_size = 5;
     list = make_unique<string []>(5);
}
```



```
GroceryList::GroceryList(): num_items(0), list_size(5), list(make_unique<string []>(5)){

}
```

```
GroceryList::GroceryList(size_t n){
     num_items = 0;
     list_size = n;
     list = make_unique<string []>(n);
}
```



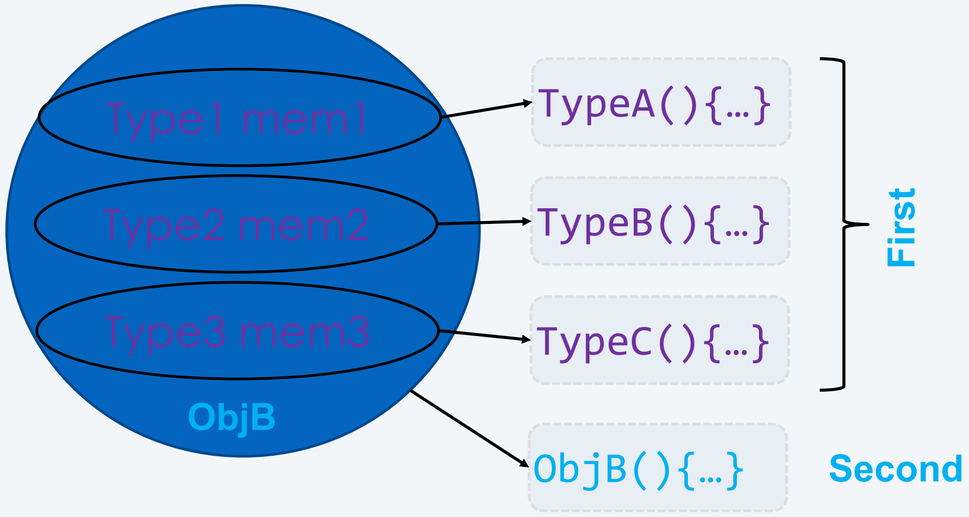
```
GroceryList::GroceryList(): num_items(0), list_size(n), list(make_unique<string []>(n)){

}
```

Deallocating Members

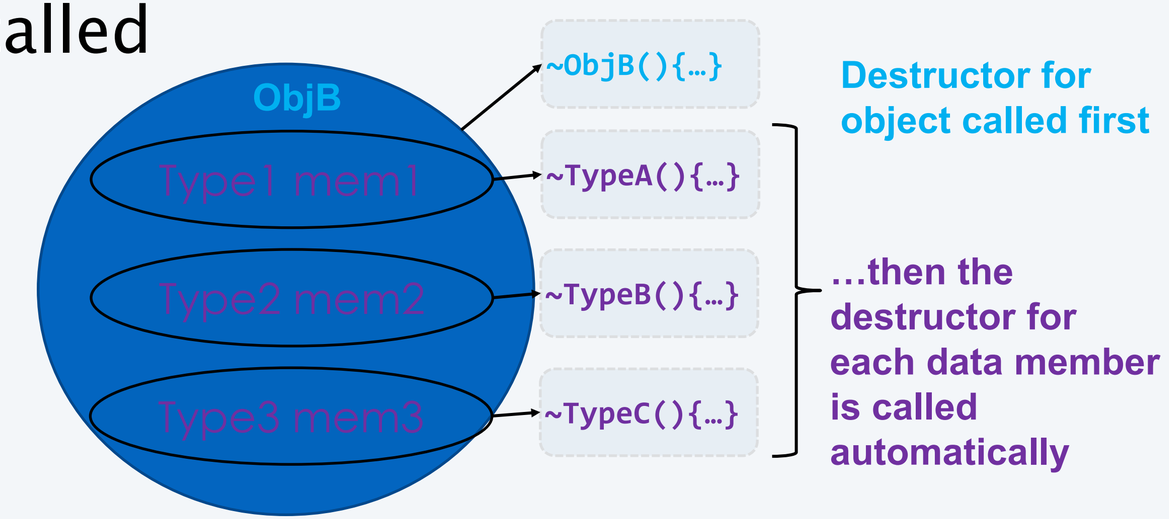
When an object is created, the constructors for data members are called BEFORE the constructor of the object

When an object is destructed, the destructors of the data members are called AFTER the destructor of the object

Destructors are never explicitly called



Construction



Destruction

C++ Classes: Destructors

Destructors are called when an object goes out of scope or is freed from the heap

Destructors

- **The default destructor is empty**
- Have no return value
- Have the name ~ClassName()
- The destructors of data members are called automatically upon completion of the destructor.

When to write destructor

- To clean up resources that won't go away automatically
- Destructors may be need to release resources such as files and dynamically allocated memory

```
GroceryList::~GroceryList(){
  //To learn when destructors are called
  // try printing from them:
  // cout <<"destroying list" <<endl;
}
/* After GroceryList destructor is called
   1) memory for size_t data members
num_items and list_size is destroyed
    2) the destructor for the
unique_ptr<string []> list is called.
This destroys the memory on the heap
*/

/* Old school warning: if we had managed
memory manually, i.e. in the constructor
    string *list = new string[10];
Then in the destructor it needs to be
deleted, i.e.
      delete [] list;
*/
```

Member Functions

Member functions have access to all data members of a class

Use the dot (.) operator on object or C++ reference to access data members

Use the arrow (->) operator on pointers to objects

Use the [] operator on arrays

```
// private member function
void GroceryList::doubleList(){
   unique_ptr<string[]> old_list(list.release());
   list = make_unique<string []>(list_size*2);
   for (size_t i = 0; i < num_items; i++){
        list[i] = old_list[i];
   }
    list_size *= 2;
}
// public member function
void GroceryList::addItem( string item){
   if( num_items == list_size) doubleList();
       list[num_items] = item;
       num_items++;
}
int main()
{
  GroceryList l1;
  l1.addItem(“bananas”);
  return 0;
}
```

'const' Keyword for member functions

`const` keyword can be used with member functions to protect the object

After a member function ensures data members aren't modified by the function

The GroceryList object on which these member functions are called is not changed.

```
/* In GroceryList class*/
const string& getItem(size_t position) const;
 void printList() const;
 bool empty() const;
size_t size() const;
/* In GroceryList class */


size_t GroceryList::size() const {
      return num_items;
}


int main()
{
  GroceryList l1;
  size_t s = l1.size();
  return 0;
}
```

'const' Keyword for return values

const keyword can be used with return value

- Before a return value indicates the value returned cannot be modified
- The GroceryList object is not changed by this const member functions.
- The string returned is not changed by this function.

```
/* In GroceryList class*/
const string& getItem(size_t position) const;
/* In GroceryList class */

const string& GroceryList::getItem(size_t position) const{
   // Exercise: Add error checking that position is in list
   return list[position];
}

int main()
{
   GroceryList l1;
   l1.addItem("bananas");
   string s = l1.getItem(0);
   const string& s = l1.getItem(0);
   //string &s = l1.getItem(0); Will not compile

   return 0;
}
```

'const' Keyword for input parameters

const keyword can be used with input parameters

Before an input argument indicates the input object cannot be modified by the function

The GroceryList object passed as an input parameter, list2, is not changed by this function.

this pointer is a pointer to the calling object. The calling object in this example is list 1.

.

```
void GroceryList::merge(const GroceryList& other){
          size_t limit = other.size();
          for (size_t i = 0; i  < limit; i++){
            this->addItem(other.getItem(i));
        }
}


int main()
{
       GroceryList list1;
        list1.addItem("apples");
        list1.addItem("bananas");
        list1.addItem("peaches");
        GroceryList list2;
        list2.addItem("onions");
        list2.addItem("peppers");
        list2.addItem("broccoli");
         list1.merge(list2);
}
```