UPE Tutoring:

CS 32 Midterm 1 Review

Sign-in https://tinyurl.com/cs32mt1review

Slides link available upon sign-in

Table of Contents

Abstract Data Types (ADTs)

- “A coordinated group of **data structures, algorithms** and **interface functions** that is used to solve a particular problem.” —Carey Nachenberg
- **Data structures** and **algorithms**
  - Kept hidden from the user
- **Interface Functions**
  - Allow the user to interact with the data structures, use the algorithms

C++ Classes

In C++, the tool used to create an ADT is called a class:

```
class Car
{
  public:
    // Interface functions go here
    void printMake(...);
    double getMPG(...);
  private:
    // Algorithms and Data Structures go here
    double m_numGallons;
    void innerWorkingsOfCar(...);
};
```

C++ Classes

In C++, the tool used to create an ADT is called a class

```
class Car
{
  public:
    // Interface functions go here
    void printMake(...);
    double getMPG(...);
  private:
    // Algorithms and Data Structures go here
    double m_numGallons;
    void innerWorkingsOfCar(...);
};
```

These are in the public section, so they can be accessed by anyone

These are in the private section, so only functions in the Car class can access them

Constructors

- Special function used to initialize the variables in a class
- Has the same name as the class
- Has no return type
- Default constructor provided if and only if no user defined constructor exists

Constructors

```
class Car
{
  public:
    Car(double gallons, string make, string model)
    {
      m_numGallons = gallons;
      m_make = make;
      m_model = model;
    }
  private:
    double m_numGallons;
    string m_make;
    string m_model;
};
```

Constructors - Initializer Lists

Another way to initialize member variables in the constructor

```
class Car
{
  public:
    Car(double gallons, string make, string model)
      : m_numGallons(gallons), m_make(make), m_model(model)
    {}
  private:
    double m_numGallons;
    string m_make;
    string m_model;
};
```

Practice Question: Construction

What is the output of the following code snippet?

```
class Cat {
public:
  Cat(string name) {
    cout << "I am a cat: " << name << endl;
    m_name = name;
  }
private:
  string m_name;
};
```

```
class Person {
  public:
    Person(int age) {
      cout << "I am " << age << " years old. ";

    m_cat = Cat("Alfred");
    m_age = age;
  }
private:
  int m_age;
  Cat m_cat;
};

int main() {
  Person p(21);
}
```

Solution: Construction

This code won't compile! The `Cat` class does not have a default constructor, meaning that its arguments need to be passed in as part of the initializer list.

```
class Person {
public:
  Person(int age) {
    cout << "I am " << age << " years old. ";
    m_cat = Cat("Alfred");
```

To fix this issue, we need to pull out the initialization of `m_cat` like so:

```
Person(int age) : m_cat("Alfred") { ... }
```

If we apply this fix, we would find that the output is as follows:

```
I am a cat: Alfred
I am 21 years old.
```

This ordering is a consequence of the order of construction, where member variables are constructed before the constructor is called.

Destructors

- Special functions called whenever the object is destroyed
- Has the name `~className`
- Has no return type and takes no parameters
- If not defined by user, a default one will be used that simply calls the destructors of all ADT data members
- Often used to free dynamically allocated variables

Run-through

```
class Engine
{
  public:
    Engine(int numCylinders) : m_numCylinders(numCylinders)
    {
      cout << “I am an engine with ”;
      cout << m_numCylinders + “ cylinders” << endl;
    }

    ~Engine() { cout << “Kaboom!” << endl; }
  private:
    int m_numCylinders;
};
```

Run-through

```
class Car
{
  public:
    Car(double gallons, string make, string model, int cylinders)
      : m_engine(cylinders), m_numGallons(gallons), m_make(make), m_model(model)
    {
      cout << “A car has been created” << endl;
    }
    ~Car() { cout << “The car has been destroyed” << endl; }
  private:
    Engine m_engine;
    double m_numGallons;
    string m_make;
    string m_model;
};
```

Error or no error?

Remember our constructor definitions:

```
Engine(int cylinders);
Car(double gallons, string make, string model, int cylinders);
```

```
int main()
{
  Engine e;
  Engine(“GMC”, 8);
  Engine(4);

  Car(10, 4, “Chevy”, “Equinox”);
  Car(25, “Hummer”, “H2”, 10);
}
```

Error or no error?

Remember our constructor definitions:

```
    Engine(int cylinders);
    Car(double gallons, string make, string model, int cylinders);
```

```
int main()
{
  Engine e;                                   // Parameters required, but there are none
  Engine(“GMC”, 8);                           // Wrong parameters
  Engine(4);

  Car(10, 4, “Chevy”, “Equinox”);             // Wrong order of parameters
  Car(25, “Hummer”, “H2”, 10);
}
```

Run-through

```
int main()
{
  Car c(25, “Hummer”, “H2”, 10);
  // What will be printed out?
}
```

Run-through

```
int main()
{
  Car c(25, “Hummer”, “H2”, 10);
  // What will be printed out?
}
```

Back to our Car constructor:

```
  Car(double gallons, string make, string model, int cylinders)
    : m_engine(cylinders), m_numGallons(gallons), m_make(make), m_model(model)
  {
    cout << “A car has been created” << endl;
  }
```

Run-through

```
int main()
{
  Car c(25, “Hummer”, “H2”, 10);
  // What will be printed out?
}
```

Back to our Car constructor:

```
  Car(double gallons, string make, string model, int cylinders)
    : m_engine(cylinders), m_numGallons(gallons), m_make(make), m_model(model)
  {
    cout << “A car has been created” << endl;
  }
```

Run-through

The Car constructor creates an Engine!

Remember, we passed in `10` for `cylinders`.

Run-through

The Car constructor creates an Engine! Remember, we passed in 10 for cylinders when we used the Car constructor.

Back to our Engine constructor:

```
Engine(int numCylinders) :
  m_numCylinders(numCylinders)
{
  cout << "I am an engine with ";
  cout << m_numCylinders + " cylinders";
  cout << endl;
}
```

```
> I am an engine with 10 cylinders
```

Run-through

Now that the Engine in the Car's initializer list has been constructed, we go back to the Car constructor.

```
Car(double gallons, string make,
    string model, int cylinders)
  : m_engine(cylinders), m_numGallons(gallons),
    m_make(make), m_model(model)
{
  cout << "A car has been created" << endl;
}
```

```
> I am an engine with 10 cylinders
> A car has been created
```

Run-through

Now that the Car is created, we go back to the main function.

```
int main()
{
  Car c(25, “Hummer”, “H2”, 10);
}
```

We created the Car c.

Now we’re at the end of the main function, when destructors are called!

First, the container class: Car

Then the contained class: Engine

Destructors

```
class Car
{
  public:
    Car(...) {...}

    ~Car()
    {
      cout << “The car has been destroyed” << endl;
    }
  private:
    double m_numGallons;
    string m_make;
    string m_model;
};
```

Run-through

To the Car destructor.

```
~Car()
{
  cout << “The car has been destroyed” << endl;
}
```

```
> I am an engine with 10 cylinders
> A car has been created
> The car has been destroyed
```

Run-through

To the Car destructor.

```
~Car()
{
  cout << “The car has been destroyed” << endl;
}
```

The Car object’s destructor is finished. Now, the destructors for all of Car’s ADT data members are called.

```
~Engine()
{
  cout << “Kaboom!” << endl;
}
```

```
> I am an engine with 10 cylinders
> A car has been created
> The car has been destroyed
> Kaboom!
```

Practice Question: Destruction

What is the output of the following code snippet?

```
class Cat {
public:
  Cat(string name) {
    cout << "I am a cat: " << name << endl;
    m_name = name;
  }
  ~Cat() { cout << "Farewell, meow." << endl; }
private:
  string m_name;
};
```

```
class Person {
public:
  Person(int age) {
    cout << "I am " << age << " years old. ";
    m_cat = new Cat("Alfred");
    m_age = age;
  }
  ~Person() { cout << "Goodbye!" << endl; }
private:
  int m_age;
  Cat *m_cat;
};

int main() {
  Person p(21);
}
```

Solution: Destruction

We would expect the following output:

```
I am 21 years old. I am a cat: Alfred
Goodbye!
```

Notice that the destructor for `m_cat` is never called: this is a memory leak, which should be addressed by adding `delete` in the `Person` destructor.

Copy Constructors

- Functions used when the following style of code is executed

```
Engine e(8);
Engine f = e;   // Copy constructor used (not assignment operator!)
Engine g(e);    // Copy constructor used
```

- Default version just copies the member variables from the existing object to the new object
- We can write our own copy constructor
  - Why would it be necessary?

Copy Constructors - When are they Needed?

```
class Dictionary
{
  public:
    Dictionary(int words) : m_numWords(words)
    {
      contents = new string[m_numWords];
    }
    void addWord(string w); // Adds a word to the dictionary
    ~Dictionary()
    {
      delete [] contents;
    }
  private:
    int m_numWords;
    string* contents;
};
```

Copy Constructors - When are they Needed?

```
class Dictionary
{
  public:
    Dictionary(int words) : m_numWords(words)
    {
      contents = new string[m_numWords];
    }
    void addWord(string w); // Adds a word to the dictionary
    ~Dictionary()
    {
      delete [] contents;
    }
  private:
    int m_numWords;
    string* contents;
};
```

This class needs a non-default copy constructor because it has dynamically allocated data.

Copy Constructors Example

```
Dictionary(const Dictionary &other) // Takes a constant reference to
                                    // another object of the same type
```

Copy Constructors Example

```
Dictionary(const Dictionary &other)
{
  // Next, determine how much memory is needed for the new object
  // and allocate it
  contents = new string[other.m_numWords];
```

Copy Constructors Example

```
Dictionary(const Dictionary &other)
{
  contents = new string[other.m_numWords];

  m_numWords = other.m_numWords;          // Finally, copy over all the
  for (int i = 0; i < m_numWords; ++i)    // contents of the other
  {                                       // Dictionary to our new
    contents[i] = other.contents[i];      // Dictionary object
  }
}
```

Practice Question: The Ricks Must Be Crazy

The Council of Ricks is trying to make duplicates of each Rick and each of the Mortys he controls. A Rick is defined by the class below. Each Rick contains a string to hold his dimension, and a Morty pointer to the head of a Linked List of Mortys. A Morty is defined by the struct below. Implement the copy constructor for the Rick class that does a deep copy.

```
class Rick {
public:
   Rick(const string& d) :
      dimension(d), head(nullptr) { }
   ...
   // Implement Me!!!
   Rick(const Rick &other);
private:
   struct Morty {
      string name;
      Morty* next;
   };
   string dimension;
   Morty* head;
};
```

Solution: The Ricks Must Be Crazy

```
// Copy Constructor
Rick::Rick(const Rick &other)
   : Rick(other.dimension)
{
   // Copy the Mortys from other Rick.
   if (other.head != nullptr) {
      head = new Morty(*other.head);      // copies everything from other.head into head
      Morty* p = head;
      while (p->next != nullptr) {
         p->next = new Morty(*p->next);
         p = p->next;
      }
   }
}
```

Assignment Operators

- Functions used when the following type of code is executed

```
Dictionary d(8);
Dictionary e(4);
d = e; // Assignment operator called
```

- Default version just copies the member variables from the existing object to the new object
- We need to write our own assignment operator when a class has dynamically allocated data, just like a Copy Constructor

Assignment Operators - Syntax

- Assignment operators have the name **operator=** and return a **reference** to an object of the class for which they are defined
  - This allows us to chain assignments:

```
Dictionary d(4), e(2), f(3);
e = f = d;
```

- The parameter is the Dictionary on the right hand side of the assignment

```
    Dictionary& operator=(const Dictionary& src);
or: Dictionary& operator=(Dictionary src);
```

Assignment Operators Example

```
// Prerequisite: We need to be able to swap with another Dictionary.
void swap(Dictionary& src) {
  std::swap(contents, src.contents);
  std::swap(m_numWords, src.m_numWords);
}
```

Assignment Operators Example

```
void swap(Dictionary& src) {
  std::swap(contents, src.contents);
  std::swap(m_numWords, src.m_numWords);
}

Dictionary& operator=(const Dictionary& src) {
  Dictionary temp = src;  // First, make a copy of the other Dictionary
  // ...

}
```

Assignment Operators Example

```
void swap(Dictionary& src) {
  std::swap(contents, src.contents);
  std::swap(m_numWords, src.m_numWords);
}

Dictionary& operator=(const Dictionary& src) {
  Dictionary temp = src;
  swap(temp);  // Second, swap this dictionary with the copy.
  // ...
}
```

Assignment Operators Example

```
void swap(Dictionary& src) {
  std::swap(contents, src.contents);
  std::swap(m_numWords, src.m_numWords);
}

Dictionary& operator=(const Dictionary& src) {
  Dictionary temp = src;
  swap(temp);
  return *this;  // Finally, return *this (by convention).
}
```

Assignment Operators Example

```
void swap(Dictionary& src) {
  std::swap(contents, src.contents);
  std::swap(m_numWords, src.m_numWords);
}

Dictionary& operator=(const Dictionary& src) {
  Dictionary temp = src;
  swap(temp);
  return *this;
}  // temp destructed along with old data
```

Assignment Operators Example

```
void swap(Dictionary& src) {
  std::swap(contents, src.contents);
  std::swap(m_numWords, src.m_numWords);
}

Dictionary& operator=(Dictionary src) {  // This is equivalent (and faster in some cases).
  Dictionary temp = src;
  swap(src);
  return *this;
}
```

Assignment Operators Example

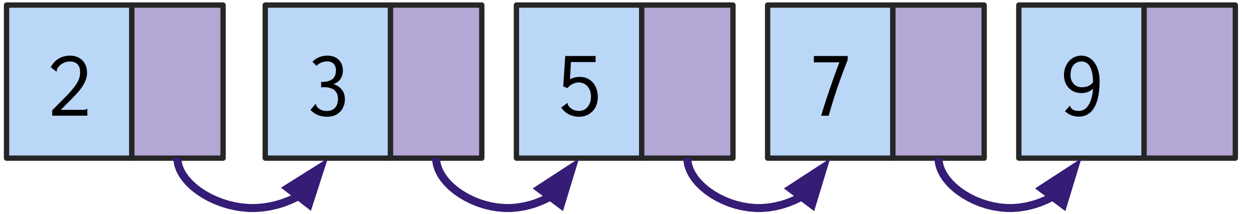
```
void swap(Dictionary& src) {
  std::swap(contents, src.contents);
  std::swap(m_numWords, src.m_numWords);
}

Dictionary& operator=(const Dictionary& src) {
  if (this != &src) {  // optionally check for aliasing
    Dictionary temp = src;
    swap(temp);
  }
  return *this;
}
```

Linked Lists

- A linked list is a data structure composed of **nodes**, each of which holds a **value** and a **pointer** to the **next** node in the list
- The list starts with a **head** pointer to the first node and ends with a node that has a next pointer set to nullptr. Occasionally, a **tail** pointer points to the last node in the linked list.
- In C++, nodes are usually declared as structs.

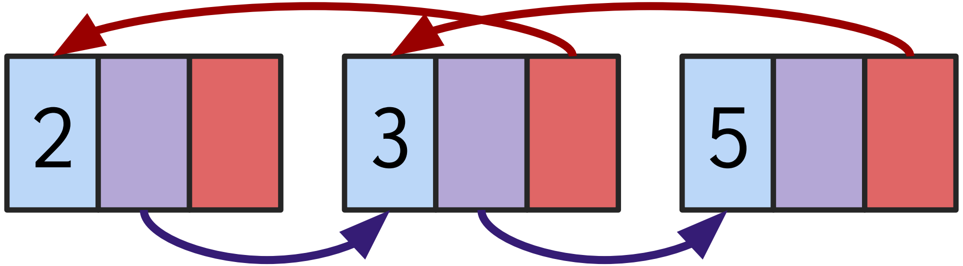
```
struct Node {
  int value;
  Node* next;
};
```



Doubly Linked Lists

Doubly linked lists are simply linked lists where each node also has a **prev** pointer in addition to the next pointer and the value.

```
struct Node {
    int value;
    Node* next;
    Node* prev;
};
```



Linked List Example

Write a function called deleteNodeWithValue that deletes all nodes with a specified value from a doubly linked list. You can assume that you have a global Node pointer called m_head that points to the beginning of your doubly linked list.

```
Node* m_head;
struct Node {
    int value;
     Node* next;
     Node* prev;
  };
```

Function Declaration:

```
Void deleteNodeWithValue(int value)
```

As we solve this problem together, think about how each step would interact if we have a list consisting of 2 → 3 → 2 and passed in 2 to this function. We can talk it through as we go as well.

Linked List Example

Step 1: Deal with an empty linked list:

Linked List Example

Step 1: Deal with an empty linked list:

If we have an empty linked list, we can return without doing anything.

```
void deleteNodeWithValue (int value)
{
    if(m_head == nullptr)
        return;
    ...
}
```

Linked List Example

Step 2: The start of the list contains the value to delete:

Linked List Example

Step 2: The start of the list contains the value to delete:

If this happens, we delete and shift the head until it no longer does or empties.

```
void deleteNodeWithValue (int value)
{
      //Step 1 code
      while(m_head != nullptr && m_head->value == value)
      {
           // the value is found in the first element of the list
           Node* temp = m_head;
           m_head = m_head->next;
           if(m_head != nullptr)
                m_head->prev = nullptr;
           delete temp;
      }
      ...
}
```

Linked List Example

Step 3: Check to see if we have finished going through the list:

Linked List Example

Step 3: Check to see if we have finished going through the list:

If this happens, we can stop execution because the list is empty.

```
void deleteNodeWithValue (int value)
{
    //Step 1 and Step 2 code

    if (m_head == nullptr)
        return;
    ...
}
```

Linked List Example

Step 4: Scan the remaining part of the list for nodes to delete:

Linked List Example

Step 4: Scan the remaining part of the list for nodes to delete:

We take advantage of the fact that m_head is safe to delete from the rest of it.

```
void deleteNodeWithValue (int value)
{
      //Step 1 , Step 2 and Step 3 code
      Node* cur = m_head;

      while (cur != nullptr)
      {
            if (cur->next != nullptr && cur->next->value == value)
            {

            }
            else
                  cur = cur->next;
      }
}
```

```
Node* target = cur->next;
if (target->next != nullptr)
      target->next->prev = target->prev;
target->prev->next = target->next;
delete target;
```

Practice Question: Sorted Linked List Insertion

Given a sorted Linked List, write a function where you can insert a new Node into the Linked List while keeping it sorted.

```
void sortedInsert(Node*& list, Node* newNode)
```

Nodes look like this:

```
struct Node {
  int data;
  Node* next;
}
```

(Problem contributed by Rishi Bhargava)

Solution: Sorted Linked List Insertion

```
void sortedInsert(Node*& head, Node* newNode) {
  // Special case for the head node pointer (located at head)
  if (head == nullptr || newNode->data <= head->data) {
    newNode->next = head;
    head = newNode;
  } else {
    // Locate the node before the point of insertion
    Node* current = head;
    while (current->next != NULL && current->next->data < newNode->data) {
      current = current->next;
    }
    // Insert the node in between current and current->next
    newNode->next = current->next;
    current->next = newNode;
  }
}
```

Good luck!

Sign-in https://tinyurl.com/cs32mt1review
Slides https://tinyurl.com/cs32mt1reviewslides
Practice https://github.com/uclaupe-tutoring/practice-problems/wiki

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
  - Location: ACM/UPE Clubhouse (Boelter 2763)
  - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.