An Introduction to C++

Classes

Classes

- Introduction to C++
- C++ classes
- C++ class details

Introduction to C++

Complex Types in C

- To create a complex type in C
- In the .h file
  - Define *structs* to store data
  - Declare function prototypes
- The .h file serves as the interface
- In the .c file
  - Define function implementations
- Implementation is kept separate from the interface

But …

- Data and operations (functions) are still separate to some extent
- It is still open to misuse by errant programmers
  - As direct access to *struct* data is still possible
    - `LL_t* ll = LLcreate();`
    - …
    - `ll->head->next->next->next = ll->head->next;`
- One solution: classes
  - Which do not exist in C

C++

- C++ evolved from C
  - Created by Bjarne Stroustrup in 1978
  - Motivated by interface issues
- Provides constructs to support
  - Information hiding
  - Encapsulation of data and functions (methods)
  - Common situations for code re-use

C++ Classes

- Classes encapsulate both data and operations
  - Functions that belong to a class are referred to as *methods*
- Class data and methods can explicitly be made public or private
  - Which prevents programmers using a class to access its implementation details
    - Syntactically enforcing information hiding
- Classes can be *inherited*
  - Which we will not discuss in CMPT 125

C++ and C

- There are many differences between C and C++
  - C++ has many libraries that incorporate classes
    - Such as a string class
  - The *bool* data type
  - Template classes and functions
  - Exception handling
  - Different pointer types
    - A feature of modern C++ (C++11)
  - ...
- We do not have time to look at all these features
  - But we will briefly discuss memory management

Allocating Dynamic Memory in C++

- We can use *malloc* and *calloc* to allocate dynamic memory in our C++ programs
  - Don't
- C++ has its own syntax for allocating and deallocating dynamic memory
  - To allocate dynamic memory use *new*
    - `int* arr = new int[10];`
    - `Node* nd = new Node;`

The compiler figures out how much space is needed

  - To deallocate dynamic memory use *delete*
    - `delete[] arr;`
    - `delete nd;`

The []s are needed to delete an array

C++ Class Syntax

Talking About Classes

- A class provides the definition of a complex datatype and its operations
    - A class is a type definition
        - And is used in much the same way as base types (*int*, *char*, etc.)
- Creating a class does not create class variables
    - Class variables are called *objects*
    - Creating a new object of a class is referred to as *instantiating* an object
        - The process is referred as *instantiation*

Class Structure and Files

- C++ classes are typically broken down into two files
- The class definition is in a *.h* file
  - Contains class variables (properties)
  - And method prototypes

```
class LinkedList {
    //…
};
```

- The method implementations are in a *.cpp* file
  - Which `#includes` the *.h* file
  - Method implementations are preceded by the class name and the scope resolution operator, `::`
    - their *fully qualified* names

```
void LinkedList::append(int val) {
    //…
}
```

Constructors

- Classes have special methods that are used to instantiate objects
  - Called *constructors*
  - Constructors give class properties appropriate values
    - That respect class invariants
  - Constructors have the same name as the class and no return type

```
class LinkedList {
    //constructor
    LinkedList();
```

- A class can have multiple constructors
  - With different parameter lists
    - An example of *function overloading*

If no constructor is defined for a class the compiler creates a default constructor

Stack or Heap

- In C++ the programmer decides whether objects are created on the stack or the heap
  - `LinkedList ll;`

    Creates a linked list on the stack
    - Not the lack of brackets in the default constructor call
  - `LinkedList* ll2 = new LinkedList(ll);`

    On heap
    - Creates a copy of a linked list using the copy constructor
- Note that different constructors have the same name but different parameter lists
  - Because the parameter lists are different there is no ambiguity

Copy Constructors

- A copy constructor allows us to create a copy of an existing object
  - Its sole parameter is the object to be copied
  - Passed as a constant reference
- C++ helpfully auto-generates a copy constructor if a class doesn't have one
  - However it is often necessary to create your own copy constructor
  - We will discuss this later

Public and Private

- A class definition is divided into *public* and *private* sections
  - And some times *protected* – relating to inheritance
- Public attributes and methods can be accessed by non-class objects and functions
  - i.e. from outside the class
- Private attributes and methods can only be accessed inside the class
  - That is, within the implementation of class methods

Private Section

- The private section of a class relates to its implementation and data
  - Class data is generally made private
- Making class data private has two useful effects
  - It allows the implementation to be changed without also changing the interface
  - It protects class data from being given inappropriate values
- In addition to data, helper methods should also be made private

Public Section

- The public section of a class makes up its interface
  - A set of methods that define the class operations
  - Only methods that are required to be accessed from outside the class should be made public
- Since class data is private it can only be accessed through methods

  but can be directly accessed from within the class

  - Setter methods change data

    also known as mutators

  - Getter methods access data

    also known as accessors

Good Design: Public and Private

- The interface should be public and the implementation private
- This allows the implementation to be protected from inappropriate changes
  - Typically, class attributes should be made private
  - And only changed through public methods
- Making the implementation private allows it to be changed
  - Without affecting programs using the class

Setter Methods

- A setter method sets the value of a class attribute
- Setters should *respect class invariants*
  - That is they should not allow class attributes to be given inappropriate values
    - Such as radius never being negative
- If an invalid value is passed to a setter it should do something about it – what?
  - Change the method to return a boolean?
  - Assign a default value?
  - Throw an error?

Good Design?

```
class Point
{
public:
    Point();
    Point(int x1, int y1);
    void setPoint(int x1, int y1);
    int getx();
    int gety();

private:
    int x;
    int y;
};
```

```
void Point::setPoint(int x1, int y1)
{
    x = x1;
    y = y1;
}
```

What's the point of the setter?

C++ Class Issues

More About new and delete

- C++ provides *new* and *delete* instead of *malloc* and *free* to allocate and deallocate dynamic memory
  - It is particularly important to use *new* and *delete* when creating or destroying objects
  - In addition to allocating space on the heap and returning its address *new* also calls the appropriate constructor
  - In addition to deallocating space *delete* also calls the class *destructor*
- What's a destructor?
  - A special function responsible for cleaning up memory associated with an object

Classes and Dynamic Memory

- Class programmers have responsibilities relating to the use of dynamic memory
- If a class allocates space in dynamic memory to the class properties the programmer must
  - Write a destructor
  - Write a copy constructor
  - Write an overloaded assignment operator
- Failure to do may result in undesired results

Destructors

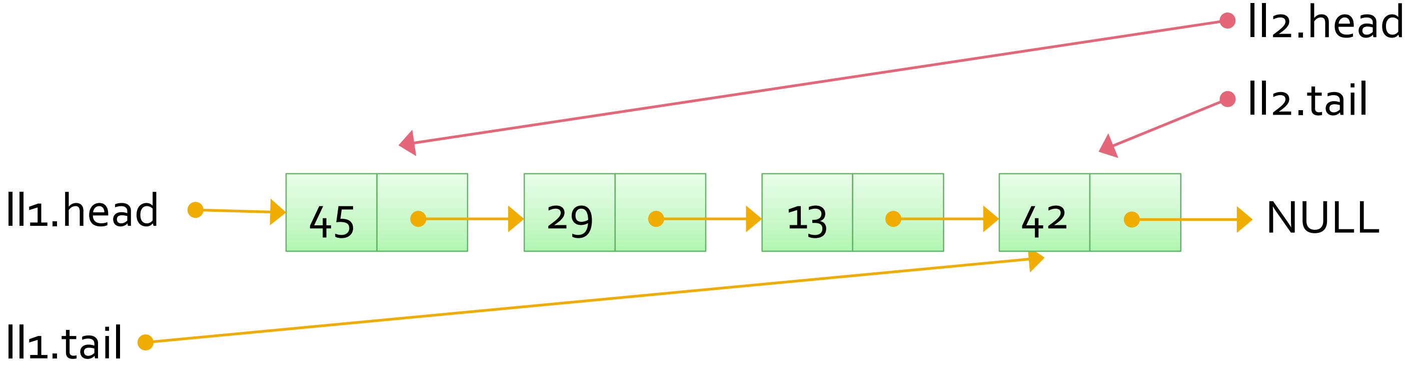
- A destructor is responsible for cleaning up dynamic memory allocated to an object
  - The destructor should call *delete* on any variable allocated space in dynamic memory
    - For a linked list this would entail traversing the list and calling *delete* on each node
- Destructors have a particular prototype
  - The name of the class preceded by a tilde
    - `~LinkedList();`
- Destructors are never explicitly called
  - But are invoked when *delete* is called on an object

Copy Constructors

- A copy constructor creates an object that is a copy of an existing object
  - If a copy constructor is not created one is automatically created
    - But it only make a *shallow copy*

```
LinkedList ll2(ll1);
```

Copies the addresses not the nodes!



ll2.head

ll2.tail

ll1.head

45

29

13

42

NULL

ll1.tail

Deep Copy

- A copy constructor for a class that allocated dynamic memory should make a *deep copy* of an object
  - A deep copy creates a copy of data stored on the heap
    - Instead of making copies of addresses to the data
- A copy constructor for a linked list would traverse through the original list
  - Calling *new* to create a copy of each node to build a separate and complete list
  - And setting the head and tail of the new linked list object to the addresses of the start and end of the new list

Assignment

- The copy constructor and destructor address most of the issues with classes and dynamic memory
- The destructor deallocates dynamic memory allocated to an object
- The copy constructor deals with
  - Explicitly creating a copy of an existing object
    - By calling the constructor
  - Passing an object by value to a function parameter
    - Which calls the copy constructor to create the new object
- What happens when one object is assigned another?

Consider This

```
LinkedList ll1;
LinkedList ll2;

// Populate lists and work with ll1 and ll2

ll1 = ll2;
```

What happens?

- Presumably the intent is to make *ll1* a copy of *ll2*
  - Destroying the original contents of *ll1* in the process
- Looks very similar to what the copy constructor does
  - But *ll1*'s copy constructor is not called in this situation
    - Why not?
- Solution: overload the assignment operator

Overloaded Operators

- C++ allows its operators to be overloaded
  - Have their operations defined for use with non base-type variables
- Much like the copy constructor the assignment operator creates a shallow copy
  - Unless the class programmer explicitly overloads the assignment operator to make a deep copy
  - The overloaded assignment operator should also clean up memory associated with the original object