CIS 014 – C++ Programming

Lecturer: Joseph Su

PASADENA CITY COLLEGE ®

REFERENCES

Optional Textbook:

Programming: Principles and Practice Using C++, 2nd ed, B. Stroustrup, Addison-Wesley, 2014

PDF:

http://www.cplusplus.com/files/tutorial.pdf

Online:

http://www.cplusplus.com/doc/tutorial/

The C++ Programming Language, 4th ed.

B. Stroustrup, Addison-Wesley, 2013

C++ How to Program, 10th ed

Deitel & Deitel, Pearson Hall, 2016

C++ Primer, 5th ed

S. Lippman, J. Lajoie, and B. Moo, Addison-Wesley, 2012

READING ASSIGNMENTS

ONLINE

- Containers (STL – Standard Libraries)
- **C++ Class Constructor and Destructor**
- Linked List Data Structure

TEXTBOOK

- 17.7 Pointers to class objects, 17.9 Pointers and references, 17.9.1, 17.9.3 An example: lists, 17.9.4 List operations
- 20.4 Linked lists

READING ASSIGNMENTS

REFERENCES

ASCII http://www.cplusplus.com/doc/ascii/
BOOLEAN http://www.cplusplus.com/doc/boolean/
RAND(): http://www.cplusplus.com/reference/cstdlib/rand/

http://www.cplusplus.com/files/tutorial.pdf (pages 1-94)
http://www.cplusplus.com/doc/tutorial/

- ✓ Program Structure
  - Complete all chapters
- ✓ Compound Data Types
- ✓ Classes
  - Classes (I)

TODAY

- Reviews: Pointers
- STL Libraries
  - Linked List (ADT)
    - Introduction
    - Creation
- Program Layout
  - .c, .cpp
  - .h, .hpp
  - Review: Header File Usage
- Reviews: Classes
- Class:
  - Constructor
  - Destructor

REVIEW: POINTERS

- Binky’s Pointer Fun Video http://cslibrary.stanford.edu/104/

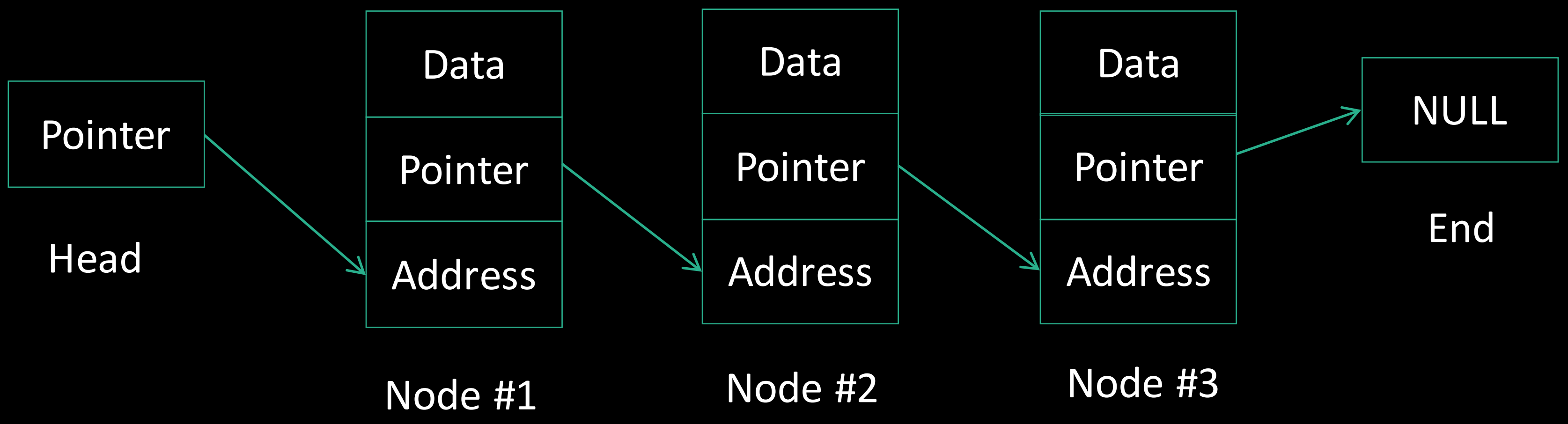
Standard Template Library (STL): Linked List

- http://www.cplusplus.com/reference/stl/
- STL is a library of many containers
- A container is a collection of objects
- In C/C++ you have <array>, <vector>, etc., which are provided to you as classes of objects for you to work with in programs
- Linked list is one of those provided container in STL
- Linked list is also called *Abstract Data Type* (ADT) in computer science
- Linked list is implemented using pointers

  - Singly linked list (#include<forward_list>)
  - Doubly linked list (#include<list>)
  - http://www.cplusplus.com/reference/

LINKED LIST

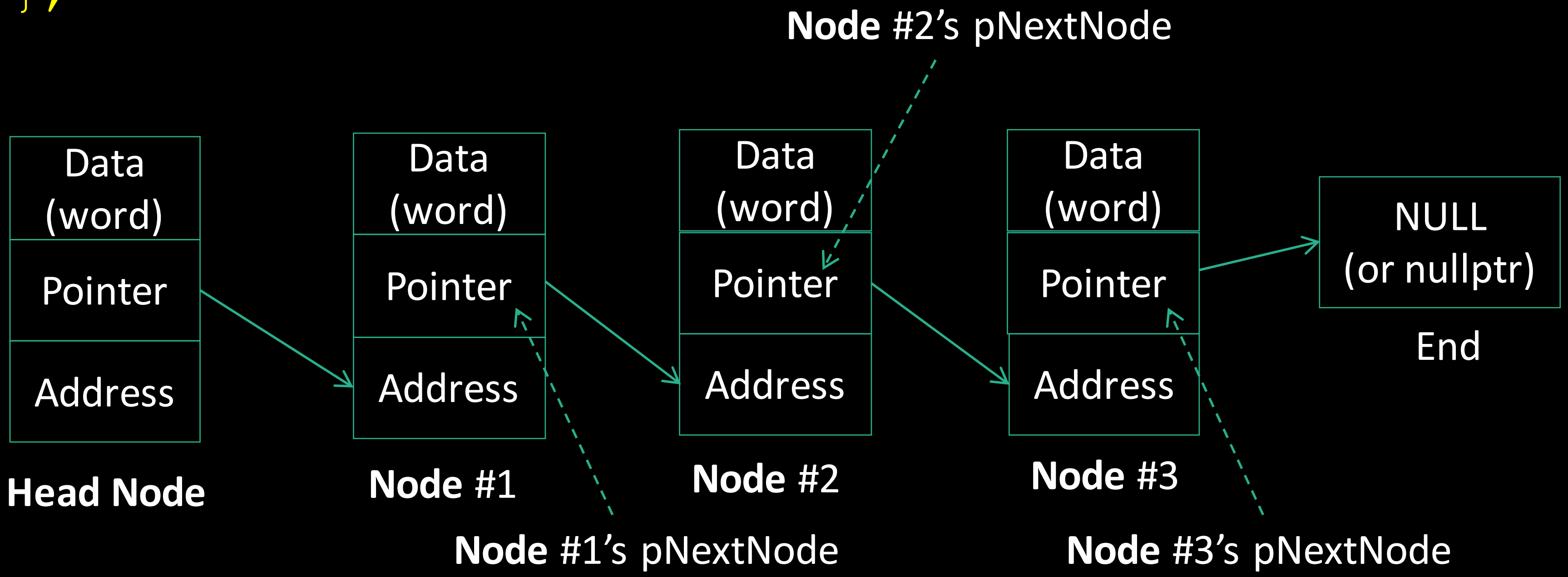
- Linked List is a collection of nodes containing data
- Each node is “linked” to the next by a pointer
- Individual nodes can be added or removed dynamically at the beginning, at the end, in the middle, or anywhere in this list
- The beginning of the list starts with a pointer
- The end of the list points to a NULL (‘\0’)
- An example is shown as follows:



LINKED LIST: INTRODUCTION

- Let’s say each Data in each node is a string
- The first task is to define a node using a `class` in C++, or in C, with a `struct`:

```
struct Node {
   char word[MAX_WORD_LENGTH];
   struct Node* pNextNode;
};
```



LINKED LIST: CREATION

- We may create the list:

```
struct Node {
    char word[MAX_WORD_LENGTH];
    struct Node* pNextNode;
};
Node* list = new Node;  //creating new master list
```

- Then we may create the first Node and attach it to the list:

```
Node* newNode = new Node;
cout << “Enter new word” << endl;
cin >> newNode->word;

list->pNextNode = newNode; //assigning newNode to list
```

- Remember to delete dynamic links one by one after done:

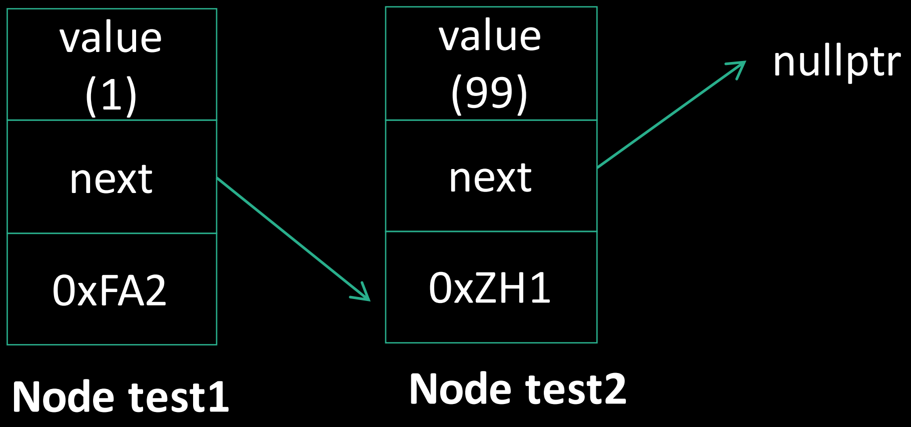
```
delete newNode; // avoid memory leak
```

LINKED LIST: Creation on Stack, C++

- In C++:

```
class Node {
   public:
      int value;
      Node* next;
      Node(int v) : value(v), next(NULL) {}
};
Node test1(1);
Node test2(99);
cout << test2.value; // prints 99
```

- Then we create a small linked list: test1 -> test2 -> nullptr:



LINKED LIST: Creation on Stack, C++ (continued)

- Implementation:

```
Node test1(1);
Node test2(99);
test1.next=&test2;
test2.next=nullptr;
```

PROGRAM LAYOUT: SINGLE CPP FILE

- Declaring classes all in one file:
  - #include’s
  - #define’s
  - Global variables
  - Constants
  - Class declarations
  - Class member functions
  - Functions
  - main()

- Motivations:
  - Want to make cpp file shorter and more readable
  - Want to make classes reusable by other programs

PROGRAM LAYOUT: CPP + HEADER

- Put basic program in source file (.cpp, .c)
  - Global variables
  - Non-member functions
  - Main()

- Put class information in a separate header file (.h, .hpp), which contains definitions of the classes
  - Definition of constants
  - Declaration of classes
  - Definition of member functions
  - Declaration of non-member functions
  - Declaration of *more global* variables

Header files can be used by other programs. For example, #include <iostream> in your console programs

REVIEW: HEADER FILE USAGE

The source file (.c, .cpp) includes the header file (.h, .hpp) at the top:

```
   #include “filename.h”
```

While your header file looks like:

```
   #ifndef FILENAME_H
   #define FILENAME_H
   #include <libraries>
      **YOUR CLASSES HERE**
   #endif
```

The `#ifndef-#define` statements prevent multiple inclusion of the same header file in your program. This is to ensure that this exact header file is compiled only once.

PUTTING EVERYTHING TOGETHER

```
main.cpp   // main source or entry point of console program

class1.cpp // class1 members – all definitions
class1.h   // class1 declarations

class2.cpp // class2 members – all definitions
class2.h   // class2 declarations
```

In your class1.h:

```
#ifndef CLASS1_H
#define CLASS1_H
    class Class1 {
        void func1();
    };
#endif
```

PUTTING EVERYTHING TOGETHER

In your main.cpp (including “class1.h”):

```
#include “class1.h”
void func1() {…}          // different than the func1() in class1.cpp
int main() {…}
```

In your class1.cpp (including “class1.h”):

```
#include “class1.h”
Class1::Class1() {…}
Class1::func1() {…}// different than the func1() in main.cpp
```

NOTES:

1. Both main.cpp and class1.cpp have class1.h inclusion
2. func1() in main.cpp is DIFFERENT than the class member function, func1(), in class1.cpp!

REVIEW: CLASSES

A Class:

- Represents a concept in a program
- Is a user-defined type with its specific user-defined behaviors
- In C++, it is THE building block for large programs
- Is a way of:
  - Encapsulating data
  - Defining abstract data types along with initialization conditions and operations allowed on that data
  - Hiding implementation details
  - Sharing behavior and characteristics

REVIEW: CLASSES

```
class CRect {

   int x, y;

   public:
      void set_values (int,int);
      int area ();
};

CRect rect;     // rect is a variable of type CRect
rect.x = 2;     // access rect’s member variable x:
                // Really??? What is wrong?
rect.area();    // access rect’s member function area()
```

REVIEW: CLASS ANATOMY

```
// example: one class, two objects
#include <iostream>
using namespace std;
class CRectangle {
    int x, y;
    public:
    void set_values (int,int);
    int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b) {
    x = a;
    y = b;
}

int main() {
    …
}
```

Class name

Class member variable declarations

Class member function declarations

Class scope operator

REVIEW: CLASS ANATOMY

- Using the previously defined CRectangle class, we have

```
int main () {
   CRectangle rect, rectb;
   rect.set_values (3,4);
   rectb.set_values (5,6);
   cout << "rect area: " << rect.area() << endl;
   cout << "rectb area: " << rectb.area() << endl;
   return 0;
}
```

Two different instances of CRectangle

- Recall **main**() is your program’s entry
- We declare two instances of **CRectangle**, **rect** and **rectb**
- Each of **rect** and **rectb** is an object
- **rect** has its own life cycle, so is **rectb** having its own that is separate from **rect**’s

CLASS: CONSTRUCTOR

- Sometimes we need to ensure that certain variables in our object at run time has certain values before any member functions is called.

- We initialize these variables in the class' constructor

```
class CRectangle {
    int x, y;
    public:
    CRectangle(int, int);
    void set_values (int,int);
    int area () {return (x*y);}
};
```

// CRectangle(int, int) is only called when a CRectangle instance is created

CLASS: CONSTRUCTOR FROM PRIOR EXAMPLE

- If a custom constructor is available in a class, you use it.
- When `CRectangle rect(3,4)` is called in main(), an instance of the `CRectangle` class is created on the execution stack.
- That instance (or an object) is referred to by rect
- When rect was created, `CRectangle’s` constructor was called

```
CRectangle(int, int);
```

- If no constructor is explicitly called the program will invoke the default constructor:

```
CRectangle();
```

CLASS: CREATE AN OBJECT

- At run time you can create an object of class type CRectangle via the following statement:

```
 CRectangle rect;
```

- The above invokes CRectangle's constructor:

```
 CRectangle();
```

The above constructor can be explicitly declared and defined by you. If not the compiler provides it to you by default.

CLASS: CONSTRUCTOR EXAMPLE

```
class CRectangle {
   int width, height;
  Public:
   CRectangle ();
   CRectangle (int,int);
   ~CRectangle();
   int area () {return (width*height);}
};

CRectangle::CRectangle () {
   width = 5;
   height = 5;
}

CRectangle::CRectangle (int a, int b) {
   width = a;
   height = b;
}
```

CLASS: DESTRUCTOR

- `CRectangle()` default constructor is provided by the compiler if you don’t create your own constructor.
- `CRectangle` also has a destructor, which performs the opposite of what a constructor does
- By default that destructor is

```
 ~CRectangle();
```

- Note the ~ sign on a destructor, which has the same name as the class name
- Automatically called when an object is destroyed
- Destructor is where all of your dynamic variables are de-allocated