Namespace, Stack, Queue, Linked list

Prof. Seokin Hong

Agenda

- **Namespace**
- **Linked List, Stack, Queue**

Namespaces

- **A namespace is a collection of name definitions, such as class definitions and variable declarations**
  - If a program uses classes and functions written by different programmers, the same name might be used for different tasks
  - Namespaces help us deal with this problem

- **Example**
  - #include <iostream> places names such as **cin** and **cout** in the **std** namespace

- **"Using" Directive**
  - The program does not know about names in the std namespace until you add

using namespace std;

Name Conflicts

- **If the same name is used in two namespaces**
  - The namespaces cannot be used at the same time
- **Example:**
  - If myFunction() is defined in namespaces ns1 and ns2, the two versions of myFunction() could be used in one program by using local “using” directives

```
{
  using namespace ns1;
  using namespace ns2;
  myFunction( );
}
```

name conflict

```
{
  using namespace ns1;
  myFunction( );
}
```

```
{
  using namespace ns2;
  myFunction( );
}
```

Creating a Namespace

- **To place code in a namespace**
  - Use a namespace grouping

```
namespace Name_Space_Name
{
        Some_Code
}
```

- **To use the namespace created**
  - Use the appropriate “using” directive

```
using namespace Name_Space_Name;
```

Namespaces: Declaring and defining a Function

- **To add a function to a namespace**
  - Declare the function in a namespace grouping

```
namespace my_space
{
      void greeting( );
}
```

- **To define a function declared in a namespace**
  - Define the function in a namespace grouping

```
namespace my_space{
        void greeting( ){
                cout << "Hello from namespace savitch1.\n";
        }
}
```

Namespaces: Using a Function

- **To use a function defined in a namespace**
  - Include the “using” directive in the program where the namespace is to be used
  - Call the function as the function would normally be called

Using directive's scope

Qualifying Names

- **Suppose you have the namespaces below:**

```
namespace ns1
{
      fun1( );
      myFunction( );
 }
```

```
namespace ns2
{
      fun2( );
      myFunction( );
}
```

- **”Using” declarations allow us to select individual functions to use from namespaces**
  - using ns1::fun1; //makes only fun1 in ns1 avail
    - Means we are using only namespace ns1's version of fun1
    - Can overload the “using directives” (e.g., using ns2;)
- **If you only want to use the function once, call it like this**
  - **ns1::fun1( );**

Nested Namespace

- **A nested namespace is a namespace defined inside another namespace**

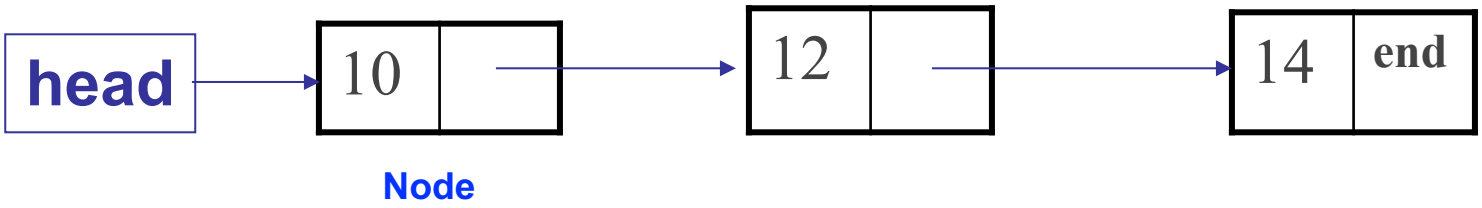
```
int x = 2;
namespace outer_space{
    int x = 1;
    int y = 2;
    namespace inner_space {
                int y = x;
    }
}

void main()
{
     std::cout<< outer_space::inner_space::y<<std::endl;
     std::cout<< outer_space::y <<std::endl;
}
```

```
int x = 2;
namespace outer_space{
    int y = 2;
    namespace inner_space {
                int y = x;
    }
}

void main()
{
     std::cout<< outer_space::inner_space::y <<std::endl;
     std::cout<< outer_space::y <<std::endl;
}
```

Linked Lists, Stack

Linked Lists

- A linked list is a list that can grow and shrink
- A linked list often consists of nodes that contain a pointer variable connecting them to other nodes



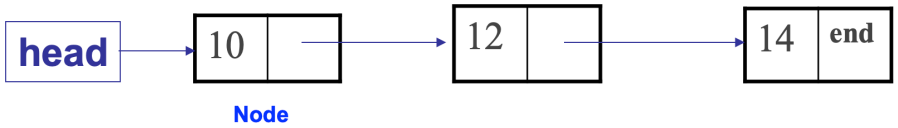
Implementing Nodes

- **Nodes are implemented in C++ as structs or classes**
  - Example:

```
struct ListNode
{
   string item;
   int count;
   ListNode *link;
 };

typedef ListNode* ListNodePtr;
```

  - Pointer variable **head** is declared as:

```
ListNodePtr head;
```



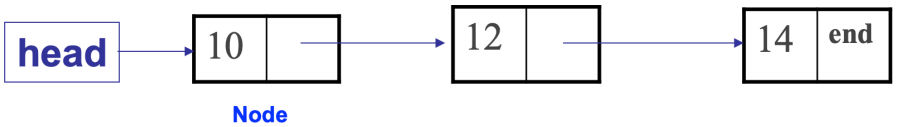
Accessing Items in a Node

- one way to change the number in the first node from 10 to 12:

```
(*head).count = 12;
```

- Using the arrow operator

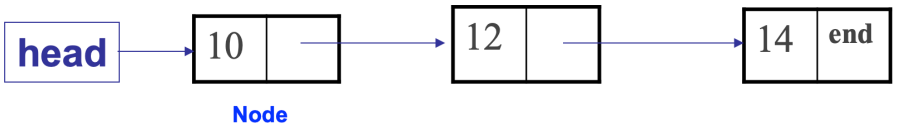
```
head -> count = 12;
```



NULL

- **The defined constant NULL is used as…**
  - An end marker for a linked list
  - The value of a pointer that has nothing to point to

- **The value of NULL is 0**



Linked List of Classes

Node.h

```
namespace My_linkedlist
{
  class Node
  {
    public:
      Node( );
      Node(int value, Node *next);
      // Constructors to initialize a node

      int getData( ) const;
      // Retrieve value for this node

      Node *getLink( ) const;
      // Retrieve next Node in the list

      void setData(int value);
      // Use to modify the value stored in the list

      void setLink(Node *next);
      // Use to change the reference to the next node

    private:
      int data;
      Node *link;
  };
  typedef Node* NodePtr;
} //My_linkedlist
```

Node.cc

```
#include <iostream>
#include "Node.h"

namespace My_linkedlist
{
      Node::Node( ) : data(0), link(NULL)
      {
            // deliberately empty
      }

      Node::Node(int value, Node *next) : data(value), link(next)
      {
            // deliberately empty
      }

            // Accessor and Mutator methods follow

      int Node::getData( ) const
      {
            return data;
      }

      Node* Node::getLink( ) const
      {
            return link;
      }

      void Node::setData(int value)
      {
            data = value;
      }

      void Node::setLink(Node *next)
      {
            link = next;
      }
}  //My_linkedlist
```

Linked List of Classes

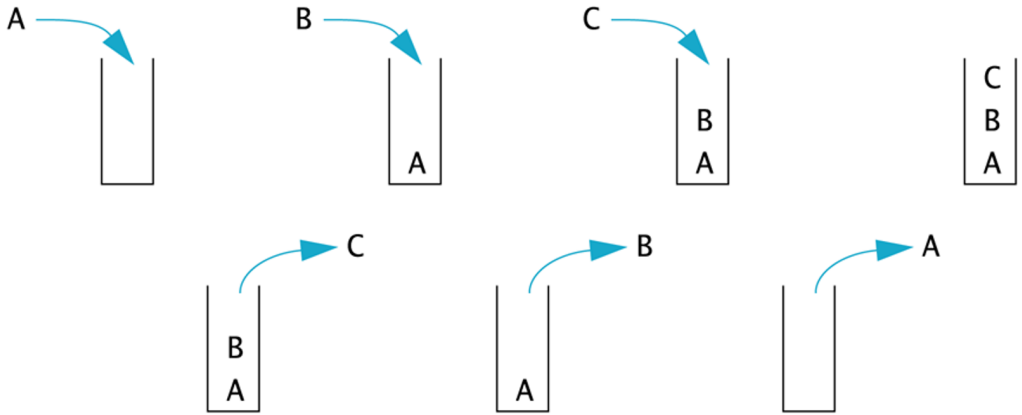
```
#include <iostream>
#include "Node.h"

using namespace std;
using namespace My_linkedlist;



void headInsert(NodePtr& head, int theNumber)
{
    NodePtr tempPtr;

    tempPtr = new Node(theNumber, head);
    head = tempPtr;
}
```

```
int main()
{
    NodePtr head, tmp;

    //Create a list of nodes 4 -> 3 -> 2 -> 1 -> 0
    head = new Node(0, NULL);
    for (int i = 1; i < 5; i++)
    {
        headInsert(head, i);
    }
    //Iterate through the list and display each value
    tmp = head;
    while (tmp != NULL)
    {
        cout << tmp->getData() << endl;
        tmp = tmp->getLink();
    }
    //Delete all nodes in the list before exiting
    //the program.
    tmp = head;
    while (tmp != NULL)
    {
        NodePtr nodeToDelete = tmp;
        tmp = tmp->getLink();
        delete nodeToDelete;
    }
    return 0;
}
```

Stack and Queue

Stack

- **Stack is a data structure that retrieves data in the reverse order the data was stored**
  - If 'A', 'B', and then 'C' are placed in a stack, they will be removed in the order 'C', 'B', and then 'A'
- **A queue is a last-in/first-out data structure**



A Stack Class

stack.h

```
//This is the header file stack.h. This is the interface for the class Stack,
//which is a class for a stack of symbols.
#ifndef STACK_H
#define STACK_H
namespace My_stack;
{
    struct StackFrame
    {
        char data;
        StackFrame *link;
    };
    typedef StackFrame* StackFramePtr;
    class Stack
    {
    public:
        Stack( );
        //Initializes the object to an empty stack.


        ~Stack( );
        //Destroys the stack and returns all the memory to the freestore.
        void push(char theSymbol);
        //Postcondition: theSymbol has been added to the stack.
        char pop( );
        //Precondition: The stack is not empty.
        //Returns the top symbol on the stack and removes that
        //top symbol from the stack.
        bool empty( ) const;
        //Returns true if the stack is empty. Returns false otherwise.
    private:
        StackFramePtr top;
    };
}//My_stack;
#endif //STACK_H
```

main.cc

```
#include <iostream>
#include "stack.h"
using namespace std;
using namespace My_stack;

int main( )
{
    Stack s;
    char next, ans;

    do
    {
        cout << "Enter a word: ";
        cin.get(next);
        while (next != '\n')
        {
            s.push(next);
            cin.get(next);
        }

        cout << "Written backward that is: ";
        while ( ! s.empty( ) )
            cout << s.pop( );
        cout << endl;

        cout << "Again?(y/n): ";
        cin >> ans;
        cin.ignore(10000, '\n');
    }while (ans != 'n' && ans != 'N');

    return 0;
}
```

A Stack Class

stack.cc

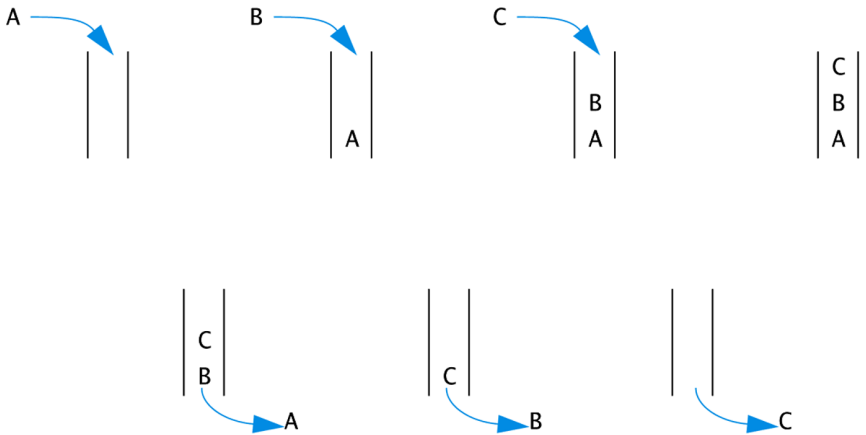
```
#include <iostream>
#include <cstddef>
#include "stack.h"
using namespace std;

namespace My_stack;
{
    //Uses cstddef:
    Stack::Stack( ) : top(NULL)
    {
        //Body intentionally empty.
    }



    Stack::~Stack( )
    {
        char next;
        while (! empty( ))
            next = pop( ); //pop calls delete.
    }

    //Uses cstddef:
    bool Stack::empty( ) const
    {
        return (top == NULL);
    }
```

⋮

```
    void Stack::push(char theSymbol)
    //Leave for practice


    char Stack::pop( )
    {
        if (empty( ))
        {
            cout << "Error: popping an empty stack.\n";
            exit(1);
        }
        char result = top->data;

        StackFramePtr tempPtr;
        tempPtr = top;
        top = top->link;

        delete tempPtr;

        return result;
    }
}//My_stack;
```

A Queue

- **A queue is a data structure that retrieves data in the same order the data was stored**
  - If 'A', 'B', and then 'C' are placed in a queue, they will be removed in the order ‘A', 'B', and then ‘C'
- **A queue is a first-in/first-out data structure**



A Queue Class

queue.h

```
#ifndef QUEUE_H
#define QUEUE_H
namespace My_queue;
{
    struct QueueNode
    {
        char data;
        QueueNode *link;
    };
    typedef QueueNode* QueueNodePtr;

    class Queue
    {
    public:
        Queue();
        //Initializes the object to an empty queue.


        ~Queue();

        void add(char item);
        //Postcondition: item has been added to the back of the queue.
        char remove();
        //Precondition: The queue is not empty.
        //Returns the item at the front of the queue and
        //removes that item from the queue.
        bool empty() const;
        //Returns true if the queue is empty. Returns false otherwise.
    private:
        QueueNodePtr front;//Points to the head of a linked list.
                           //Items are removed at the head
        QueueNodePtr back;//Points to the node at the other end of the
                          //linked list. Items are added at this end.
    };
}//My_queue;
#endif //QUEUE_H
```

main.cc

```
#include <iostream>
#include "queue.h"
using namespace std;
using namespace My_queue;

int main()
{
    Queue q;
    char next, ans;

    do
    {
        cout << "Enter a word: ";
        cin.get(next);
        while (next != '\n')
        {
            q.add(next);
            cin.get(next);
        }

        cout << "You entered:: ";
        while ( ! q.empty() )
            cout << q.remove();
        cout << endl;

        cout << "Again?(y/n): ";
        cin >> ans;
        cin.ignore(10000, '\n');
    }while (ans !='n' && ans != 'N');

    return 0;
}
```

A Queue Class

```
//This is the implementation file queue.cpp.
//This is the implementation of the class Queue.
//The interface for the class Queue is in the header file queue.h.
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include "queue.h"
using namespace std;

namespace My_queue;
{
    //Uses cstddef:
    Queue::Queue() : front(NULL), back(NULL)
    {
        //Intentionally empty.
    }



    Queue::~Queue()
     //Leave for practice
    //Uses cstddef:
    bool Queue::empty() const
    {
        return (back == NULL); //front == NULL would also work
    }
```

```
//Uses cstddef:
bool Queue::empty() const
{
    return (back == NULL); //front == NULL would also work
}

//Uses cstddef:
void Queue::add(char item)
{
    if (empty())
    {
        front = new QueueNode;
        front->data = item;
        front->link = NULL;
        back = front;
    }

    else
    {
        QueueNodePtr temp_ptr;
        temp_ptr = new QueueNode;
        temp_ptr->data = item;
        temp_ptr->link = NULL;
        back->link = temp_ptr;
        back = temp_ptr;
    }
}
  //Uses cstdlib and iostream:
  char Queue::remove()
    //Leave for practice
```

NEXT ?

Inheritance