COMP2004 Programming Practice 2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

Classes

- C++ supports OO programming
- Allows user-defined classes
- There are some similarities to Java
- There are just as many differences

- Introduce classes via an integer list class

Grouping Data

- The `struct` is the basic mechanism
- structs act as types

```
struct Student {
    std::string sid;
    std::vector<std::string> courses;
};  // The semicolon is needed
Student s;
s.sid = "9222194";
s.courses.push_back("comp2004");
```

Pointers to structs

```
Student *sp = &s;
(*sp).sid = "9222194";
(*sp).courses.push_back("comp2004");
```

- This notation is awkward
- Use `->` instead
- `sp->sid` is equivalent to `(*sp).sid`

```
Student *sp = &s;
sp->sid = "9222194";
sp->courses.push_back("comp2004");
```

The List Class

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

class List {
private:
    Node *head;
public:
    // we'll fill this in as we go
};  // The semicolon is needed
```

Constructors

- Called when an object is created
- Normal methods, except they have
  - No return type (not even void)
  - The same name as the class
- Can have several constructors using overloading
  - ie. different parameters, eg:
  - `List()`
  - `List(int n)`
  - `List(int n, double x)`

Default Constructor

- Takes no parameters

```
class List {
private:
   Node *head;
public:
   List();
}
```

- Used for default objects, eg:
  - std::vector<List> v(10);

Method code

- Can be inline or external
- Inline:

```
class List {
private:
   Node *head;
public:
   List() {
      head = NULL;
   }
}
```

Method code

- External:

```
class List {
private:
   Node *head;
public:
   List();
}

List::List() {
    head = NULL;
}
```

Constructor initialisers

- Another way of initialising members

```
List::List() : head(NULL) {
}
```

- This way is preferred where possible
  - Reason why will be shown later

Other Constructors

- Do we need any other constructors?
- A copy constructor is often needed
  - Used when passing parameters
  - Used when initialising objects
  - Compiler creates a default which is often wrong

Copy Constructor

```
List::List(const List &o) {
   if (o.head == NULL) head = NULL;
   else {
      head = new Node;
      head->number = o.head->number;
      for (Node *c1=head,*c2=o.head;
           c2;c1=c1->next,c2=c2->next) {
         c1->next = new Node;
         c1->next->number = c2->number;
      }
      c1->next = NULL; } }
```

Destructor

- A destructor is often required
- It is called when the object is destroyed
  - A variable going out of scope
  - delete being called on a pointer
  - The program ending (static variables)
- Need one if resources are allocated
  - Memory is a resource
  - Network connections
  - File handles, etc

List Destructor

```
List::~List() {
  Node *c = head;
  while (c) {
    Node *remove = c;
    c = c->next;
    delete remove;
  }
}
```

Assignment

- Assignment is another form of copying
- C++ allows operator overloading
  - We can write code for operators
  - There is an assignment operator
- A default assignment operator exists
- Having copy constructor or destructor means you need assignment operator

Assignment Operator

```
List& List::operator=(const List &o) {
  if (this == &o) return *this;
  Node *c = head;
  while (c) {
    Node *r = c;
    c = c->next; delete r;
  }
  if (o.head == NULL) head = NULL;
  else {
    // insert rest of copy code
  } return *this; }
```

A Problem

- Can you see a problem?
- We have duplicated code already
- Copying code is bad
- Instead we should make it a function
- That way it is easier to change later
- So we'll write a few more functions
- And rewrite the existing ones

Structs and Classes

- In fact, structs can have methods too
- By default
  - structs have public members
  - classes have private members
- Can override with public: and private:
- Typically
  - structs used for data only
  - classes used for data and methods
- So will convert Node to a class

Node Constructor

```
Node::Node(int val) : number(val),
                      next(NULL) {
}
```

- Now we can allocate and initialise in one step
- Before:

```
head = new Node;
head->number = o.head->number;
```

- After:

```
head = new Node(o.head->number);
```

Another Node Constructor

- Additionally initialises the next pointer

```
Node::Node(int val, Node *next)
        : number(val), next(next) {
}
```

- For example

```
head = new Node(o.head->number,
                o.head->next);
```

Merging the Constructors

- Can combine these two constructors using default parameter values

```
Node::Node(int val, Node *next = NULL)
        : number(val), next(next) { }
```

- Can use as

```
head = new Node(o.head->number,
                o.head->next);
head = new Node(o.head->number,
                NULL);
head = new Node(o.head->number);
```

More default parameters

```
Node::Node(int val = 0, Node *next
  = NULL) : number(val), next(next) { }
```

- Can use as

```
head = new Node(o.head->number,
                o.head->next);
head = new Node(o.head->number,
                NULL);
head = new Node(o.head->number);
head = new Node();
```

- This is also the default constructor

Node Copying

```
Node* Node::deep_copy() {
  Node *r = new Node(number);
  for (Node *c1 = r, *c2 = next; c2;
       c1 = c1->next, c2 = c2->next) {
    c1->next = new Node(c2->number);
  }
  return r;
}
```

New Copy Constructor

```
List::List(const List &o) {
  head = o.head->deep_copy();
}
```

- But what if the list o is empty?
- ie. o.head == NULL

New Copy Constructor

```
List::List(const List &o) {
  if (o.head)
    head = o.head->deep_copy();
  else
    head = NULL;
}
```

- Always testing like this is tedious and repetitive

Node Copying

- Make deep_copy() static and pass in a Node*

```
Node* Node::deep_copy(Node *n) {
  if (n == NULL) return NULL;
  Node *r = new Node(n->number);
  for (Node *c1 = r, *c2 = n->next; c2;
       c1 = c1->next, c2 = c2->next) {
    c1->next = new Node(c2->number);
  }
  return r;
}
```

New Copy Constructor

```
List::List(const List &o) {
  head = Node::deep_copy(o.head);
}
```

- Much better

Node Deletion

- Similarly, a static delete_chain()

```
void Node::delete_chain(Node *n) {
  while (n != NULL) {
    Node *remove = n;
    n = n->next;
    delete remove;
  }
}
```

New Destructor

```
List::~List() {
  Node::delete_chain(head);
}
```

Node Struct

- So far we have the following

```
class Node {
public:
  int number;
  Node *next;
  Node(int val = 0, Node *next = NULL)
      : number(val), next(next) { }
  static Node* deep_copy(Node *n);
  static void delete_chain(Node *n);
};
```

New Assignment Operator

```
List& List::operator=(const List &o) {
   if (this == &o) return *this;
   Node::delete_chain(head);
   head = Node::deep_copy(o.head);
   return *this;
}
```

- Much better

Copying Aside

- Most classes need these four things
  - Default Constructor
  - Copy Constructor
  - Assignment Operator
  - Destructor
- The compiler and library use them

Inserting An Element

```
void List::insert_at_front(int val) {
   Node *node = new Node(val);
   node->next = head;
   head = node;
}
```

- or more concisely

```
void List::insert_at_front(int val) {
   head = new Node(val, head);
}
```

Iterators

- We need insertion at other places
- Removal of an element
- Retrieval of an element
- All of these are best done with iterators
- Our iterator will wrap a Node*
  - Allow limited access to the elements
  - While hiding the Node objects

Simple Iterator

```
class Iterator {
friend List;
private:
   Node *node;
public:
   Iterator() : node(NULL) { };
   Iterator(Node *n) : node(n) { };

   int& operator*() {
      return node->number;
   }
```

Simple Iterator

```
   Iterator& operator++() {
      node = node->next;
      return *this;
   }

   Iterator operator++(int) {
      Iterator result = *this;
      ++(*this);
      return result;
   }
```

Simple Iterator

```
    bool operator==(const Iterator &o)
                                    const {
      return node == o.node;
    }

    bool operator!=(const Iterator &o)
                                    const {
      return node != o.node;
    }
};
```

Const correctness

- A "const" method
  - Has `const` after its parameters
  - Cannot modify the object's contents
- const references can only call const methods

Accessing Iterators

```
Iterator List::begin() {
    return Iterator(head);
}
Iterator List::end() {
    return Iterator();
}
```

- Now we can test out this basic implementation

Test Code

```
int main() {
   List list;
   for (int i = 0; i < 10; ++i)
         list.insert_at_front(i);

   for (Iterator i=list.begin(); i!=list.end(); ++i)
         *i *= 5;

   for (Iterator i=list.begin(); i!=list.end(); ++i)
         std::cout << *i << std::endl;
}
```

File layout

- Each class definition in its own .h file
- Each class code in its own .cc file

- Compile each .cc with:
  `g++ -Wall -g -c list.cc`
- Creates corresponding object file `list.o`

- Then link .o files into a program with:
  `g++ -Wall -g -o listprog list.o main.o`

Include guards

- Class definitions can only appear once
- #includes may appear many times
- Include guards ensure one definition

```
#ifndef LIST_HH_INCLUDED
#define LIST_HH_INCLUDED
// rest of file goes here
#endif
```

- Now safely `#include "list.h"` in any .cc file which uses the List class