CS32 Discussion
2019 Summer - Week 2
Qi Zhao
zhaoq0563@ucla.edu
Jul-5-2019
Outline

• Project 1
• Copy constructor
• Assignment overloading
• Forward declaration & include
• Linked list
Outline

• Project 1
• Copy constructor
• Assignment overloading
• Forward declaration & include
• Linked list
Project 1

• Task 1:
  • Define a class named `Before` with the following public interface:

```cpp
class Before
{
public:
    Before(int nRows, int nCols);
    bool keepTrack(int r, int c);
    void printWhatWasBefore() const;
};
```

• What *private members* you probably need?
Project 1

• Task 2:
  • Add a data member of type *Before* (*not of type pointer-to-\texttt{Before}*) to the Arena class, and provide this public function to access it; notice that it returns a \texttt{reference} to a *Before* object.

```c++
class Arena
{
    ...
    Before& whatWasBefore();
    ...
};
```
Project 1

• Task 3:
  • Have the player notify its arena’s *Before* object about the player's resulting position when the player stands or moves.

  • *Where should this notification be implemented?*
Project 1

• Task 4:
  • Have the Game recognize the new \texttt{b} command, tell the Arena’s \texttt{Before} object to display the grid beforehand, and then print the \textit{"Press enter to continue"} prompt and wait for the user to respond. (\texttt{cin.ignore(10000,'\n');} does that nicely.) Typing the \texttt{b} command does not count as the player's turn.
Outline

• Project 1
• Copy constructor
• Assignment overloading
• Forward declaration & include
• Linked list
Copy Constructor - Motivation

class School
{
public:
    School(const string &name);
    string getName() const; // accessor
    void setName(const string &name); // modifier
    void addStudent(const Student &student); // modifier
    Student *getStudent(const string &name) const; // accessor
    bool removeStudent(const string &name); // modifier
    int getNumStudents() const; // accessor

private:
    string m_name; // Name of the school.
    Student *m_students; // Dynamic array of students.
    int m_numStudents; // Number of students.
};
Copy Constructor - Motivation

Student st1("Brian");
Student st2("John");
School s1("UCLA");
s1.addStudent(st1);
s1.addStudent(st2);
Student *p = s1.getStudent("John");

We want to create a new School called s2, with exactly the same content as s1. In other words, we want to clone s1.
Copy Constructor - Motivation

• Candidate I: Use an assignment

School s2("");

s2 = s1;

- What are the issues with this method?
Copy Constructor - Motivation

• **Candidate I**

```cpp
School s2(""');
s2 = s1;
```

- **Correctness**: Every member variable gets copied – even the pointers (but *not* the pointees).
- **Efficiency**: It will first call the default constructor of `s2`, initialize members with default values, and then copy the values.
The Problem Here: Shallow Copy

- When we copy C-strings:

```cpp
char a[100] = "2016 Corvette Stingray";
char b[100];
b = a;
a[2] = '0';
cout << b;
```

2006 Corvette Stingray

- Deep copy: grab every character from a to b:

```cpp
for (int i = 0; i <= strlen(a); ++i)
    b[i] = a[i];
```
Copy Constructor - Motivation

• Candidate 2: Just grab values out of s1 and manually copy them into s2

School s2("");
s2.setName(s1.getName());
...

- What are the limits to this approach?
Copy Constructor - Motivation

- Candidate 2

School s2();
s2.setName(s1.getName());
// how do I get students out of s1?

- We may not have accessors and modifiers to all member variables!
- It is often not desirable to have the user (of a class) know all the internals.
- Too long to write!
Copy Constructor

```cpp
public:
    School(const School &aSchool);
```

• A constructor to define the behavior of copying from one instance to another
School::School(const School &aSchool) {
    m_name = aSchool.m_name;
    m_numStudents = aSchool.m_numStudents;
    m_students = new Students[m_numStudents];
    for (int i = 0; i < m_numStudents; ++i)
        m_students[i] = aSchool.m_students[i];
}
School Copy Constructor

- With the copy constructor defined, you can now use:

```
School s2(s1);
```

or equivalently,

```
School s2 = s1;
```
Outline

• Project 1
• Copy constructor
• Assignment overloading
• Forward declaration & include
• Linked list
But how about assignments?

\[ s2 = s1; \]
Assignment Operator Overloading

\[ s2 = s1; \]

- **Overload the operator** (in this case, we overload the **assignment operator**).

```cpp
gpublic:
    School& operator=(const School &aSchool)
```
Assignment Operator Overloading

```cpp
School& School::operator=(const School &aSchool)
{
    m_name = aSchool.m_name;
    m_numStudents = aSchool.m_numStudents;
    m_students = new Students[m_numStudents];
    for (int i = 0; i < m_numStudents; i++)
        m_students[i] = aSchool.m_students[i];

    return *this; // don't forget this!
}
```

I assume we have = operator properly defined in Student class.
Assignment Operator Overloading

```cpp
School& School::operator=(const School &aSchool) 
{
    m_name = aSchool.m_name;
    m_numStudents = aSchool.m_numStudents;
    delete[] m_students;
    m_students = new Students[m_numStudents];
    for (int i = 0; i < m_numStudents; i++)
        m_students[i] = aSchool.m_students[i];

    return *this; // don't forget this!
}
```

I assume we have = operator properly defined in Student class.
Assignment Operator Overloading

```cpp
global:School& School::operator=(const School &aSchool) {
    if (this != &aSchool) {
        m_name = aSchool.m_name;
        m_numStudents = aSchool.m_numStudents;
        delete[] m_students;
        m_students = new Student[m_numStudents];
        for (int i = 0; i < m_numStudents; i++)
            m_students[i] = aSchool.m_students[i];
    }
    return *this; // don’t forget this!
}
```

I assume we have = operator properly defined in Student class.
Outline

• Project 1
• Copy constructor
• Assignment overloading
• Forward declaration & include
• Linked list
Forward Declaration

- Forward class declaration
  - Used when class definition only uses a pointer to another class
  - Prevents the need for including the header file
  - Declares a class before it is referenced
  - Format:
    ```
    class ClassToLoad;
    ```
Forward Declaration

```cpp
#include <iostream>
#include "man.h"

int main()
{
    Man m;
    return 0;
}
```

```cpp
#include "person.h"

class Man : public Person
{
public:
    Man(void);
    ~Man(void);
private:
    Person child;
};
```

```cpp
#include "person.h"

Person::Person(void)
{
}

Person::~Person(void)
{
}
```
Forward Declaration

///////////Man.h///////////
#pragma once
#include "person.h"       //Delete
class Man : public Person
{
public:
    Man(void);
    ~Man(void);
private:
    Person child;
};
Forward Declaration

///////////Man.h///////////
#pragma once
//#include "person.h"     //Delete
class Person;
// Add
class Man : public Person
{
  public:
    Man(void);
    ~Man(void);
  private:
    Person child;
};
Forward Declaration

///////////Man.h///////////
#pragma once
#include "person.h"  // Delete

class Person;

class Man : public Person
{
    public:
        Man(void);
        ~Man(void);
    private:
        Person *child;  // Make it a pointer
};
Forward Declaration

///////////Man.h///////////
#pragma once
// #include "person.h"    // Delete
class Person;
// Add
class Man
{
public:
    Man(void);
    ~Man(void);
private:
    Person *child;  // Make it a pointer
};
Forward Declaration

**Summary**

- When you *don’t* need to use the implementation of the instance, you don’t need to `#include`
- Otherwise, you have to use `#include`
- Implementation includes: constructor, copy constructor, assignment operator, member function, even address-of operator
Include Guard

#include XXX
#define XXX

... If XXX has been defined, then these codes will not be included again
Otherwise, it can be included.
THEREFORE, once included, XXX will be defined and these codes will
not be included any more

... #endif
```cpp
#include <iostream>
#include "class_a.h"
#include "class_b.h"

int main()
{
    A aa;
    aa.get()->print();
    B bb;
    bb.get()->print();
}
```

```
#pragma once

class A
{
    public:
    void print()
    {
        std::cout << "This is class A\n";
    } // B* get()
    { // return b;
        return b;
    } //private:
    B* b;
};
```

```
#pragma once

class B
{
    public:
    void print()
    {
        std::cout << "This is class B\n";
    } // A* get()
    { // return a;
        return a;
    } //private:
    A* a;
};
```
Header Cross Include

- Once use the include guard, both header won’t recognize each other
- If not using include guard, might cause recursive inclusion
- Solution: using forward declaration
#include <iostream>
#include "class_a.h"
#include "class_b.h"

class A;
class B;

int main()
{   
    A aa;  
    aa.get()->print();
    B bb;  
    bb.get()->print();
}

ifndef CLASS_A_H
define CLASS_A_H
#include "class_b.h"

class B
{
    public:
        void print()
        {
            std::cout << "This is class A\n";
        }
        B* get()
        {
            return b;
        }
    private:
        B* b;
    }

endif

ifndef CLASS_B_H
define CLASS_B_H
#include "class_a.h"

class A
{
    public:
        void print()
        {
            std::cout << "This is class B\n";
        }
        A* get()
        {
            return a;
        }
    private:
        A* a;
    }

endif
Outline

• Project 1
• Copy constructor
• Assignment overloading
• Forward declaration & include
• Linked list
Before we talk about Linked list...

- **CS32** is all about organizing data. We call an organization scheme a **data structure**. For every data structure, we must define:
  - rules for organizing data items (e.g., array with integers stored in a nondecreasing order),
  - a method to **add** a new data item without breaking any of the rules,
  - a method to **remove** a data item without breaking any of the rules, and
  - most importantly, how to **search** for an item

- We will examine various data structures and algorithms, pros and cons of each, as well as their efficiency.
Linked Lists

- A **linked list** is a series of connected **nodes**
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- **Head**: pointer to the first node
- The last node points to **NULL**
A Simple Linked List Class

• We use two classes: **Node** and **List**

• Declare **Node** class for the nodes
  • data: **double**-type data in this example
  • next: a pointer to the next node in the list

```cpp
class Node {
public:
    double    data;    // data
    Node*     next;    // pointer to next
};
```
A Simple Linked List Class

• Declare **List**, which contains
  • **head**: a pointer to the first node in the list.
  Since the list is empty initially, **head** is set to **NULL**
• Operations on **List**

```cpp
class List {
public:
    List(void) // constructor
    ~List(void); // destructor

    bool IsEmpty() { return head == NULL; }
    Node* InsertNode(int index, double x);
    int FindNode(double x);
    int DeleteNode(double x);
    void DisplayList(void);

private:
    Node* head;
};
```
A Simple Linked List Class

• Operations of **List**
  • **IsEmpty**: determine whether or not the list is empty
  • **InsertNode**: insert a new node at a particular position
  • **FindNode**: find a node with a given value
  • **DeleteNode**: delete a node with a given value
  • **DisplayList**: print all the nodes in the list
Min. Requirements

- You need a description of a node, which must contain a next pointer.
- You need a head pointer that points to the first node.
- The list must be loop-free (unless it is a circularly linked list, in which case one (and only one) loop must exist).
See you next week!