CS 106X, Lecture 14
Classes and Pointers

reading:

*Programming Abstractions in C++, Chapter 6, 11*
Plan For Today and Friday

• Classes
• Announcements
• Implementing a Linked List
  – Pointers
  – Dynamic memory
  – Classes
  – Testing
Learning Goals

- Understand why classes are useful to encapsulate and abstract away logic.
- Understand why pointers and dynamic memory are necessary to implement a Linked List.
- Understand how to create our own classes, with unit tests.
Plan For Today and Friday

• Classes
• Announcements
• Implementing a Linked List
  – Pointers
  – Dynamic memory
  – Classes
  – Testing
A class is a definition of your own custom variable type!
The Classes Checklist

- **Specify instance variables.** What information is inside this new variable type?
- **Specify public methods.** What can this variable type do for others?
- **Specify constructor(s).** How do you create a new variable of this type?
// in ClassName.h
#pragma once
class ClassName {
public:
    ClassName(parameters);  // constructor
    returnType func1(parameters);  // member functions
    returnType func2(parameters);  // (behavior inside
    returnType func3(parameters);  // each object)

private:
    type var1;  // member variables
    type var2;  // (data inside each object)
    type func4();  // (private function)
};
In `ClassName.cpp`, we write bodies (definitions) for the member functions that were declared in the `.h` file:

```cpp
// ClassName.cpp
#include "ClassName.h"

// member function (may be multiple)
returnType ClassName::methodName(parameters) {
    statements;
}
```

- Member functions/constructors can refer to the object’s instance variables.
Example: BankAccount

• Let’s define a new variable type that represents a bank account.
• You should be able to create one by specifying the account name and the initial balance.
• You should be able to deposit and withdraw money, which should return whether or not that action was successful. You should also be able to update the account name.
• You should also be able to obtain the account balance and name.
class BankAccount {

public:

    // Step 3: how to create a BankAccount
    BankAccount(string accountName, double startBalance);

    // Step 2: the things a BankAccount can do
    bool withdraw(double amount);
    bool deposit(double amount);
    double getBalance();
    string getName();

private:

    // Step 1: the data inside a BankAccount
    string name;
    double balance;

};
#include "BankAccount.h"

bool BankAccount::withdraw(double amount) {
    if (amount <= balance && amount >= 0) {
        balance -= amount;
        return true;
    }
    return false;
}

bool BankAccount::deposit(double amount) {
    if (amount >= 0) {
        balance += amount;
        return true;
    }
    return false;
}

...
double BankAccount::getBalance() {
    return balance;
}

double BankAccount::getName() {
    return name;
}
The implicit parameter

- **implicit parameter:**
  The object on which a member function is called.

  - During the call `marty.withdraw(...)`,
    the object named `marty` is the implicit parameter.

  - During the call `mehran.withdraw(...)`,
    the object named `mehran` is the implicit parameter.

  - The member function can refer to that object's member variables.
    - We say that it executes in the *context* of a particular object.
    - The function can refer to the data of the object it was called on.
    - It behaves as if each object has its own *copy* of the member functions.
// BankAccount.cpp
void BankAccount::withdraw(double amount) {
    if (balance >= amount) {
        balance -= amount;
    }
}

// client program
BankAccount marty;
BankAccount mehran;
...
marty.withdraw(5.00);
mehran.withdraw(99.00);
Constructors

// Constructor
BankAccount::BankAccount(string accountName, double startBalance) {
    name = accountName;
    balance = startBalance;
}

...
C++ has a `this` keyword to refer to the current object.

- Syntax: `this->member`

- Common usage: In constructor, so parameter names can match the names of the object's member variables:

```cpp
BankAccount::BankAccount(string name,
                         double balance) {
    this->name = name;
    this->balance = balance;
}
```

This uses `->` not `. because it is a *pointer* to the current object.
The keyword const

• C++ const keyword indicates that a value cannot change.

```cpp
const int x = 4; // x will always be 4
```

• a const reference parameter can't be modified by the function:

```cpp
void foo(const BankAccount& ba) { // won't change ba
    // won't change ba
}
```

• Any attempts to modify d inside foo's code won't compile.

• a const member function can't change the object's state:

```cpp
class BankAccount {
    double getBalance() const; // won't change account

    // won't change account
}
```

• On a const reference, you can only call const member functions.
Static data

- **static**: Shared by all objects of a class.
  - Opposite of regular member, which are duplicated in each object.
  - Useful when a class has some class-global shared state.

```cpp
// BankAccount.h
class BankAccount {

    ...

private:
    static int ACCOUNT_ID = 1;
};
```
Class constants

• **class constant**: An unmodifiable static variable in the .h file.
  – Assign its value in the .cpp, outside of any method.
    • Don't write static when assigning the value in the .cpp.
  – For integral types, you can actually assign the variable in the .h file.

```cpp
// BankAccount.h
class BankAccount {
    static const int BANK_ROUTING_NUM = 006029593;
    static const double INTEREST_RATE;
};

// BankAccount.cpp
// set the constant to store 3.25%
const double BankAccount::INTEREST_RATE = 0.0325;
```
Plan For Today and Friday

• Classes
• Announcements
• Implementing a Linked List
  – Pointers
  – Dynamic memory
  – Classes
  – Testing
Announcements

• Midterm Exam is **Thurs. 11/1 7-9PM in 420-040**
  – Covering material through unit testing on Mon. 10/22
  – Open-book, closed note (reference sheet provided)
  – Administered via BlueBook software (on your laptop)
  – Practice materials and BlueBook download available on Friday
  – Review session **Tues. 10/30 5-6:30PM** in Hewlett 102
  – If you have a university or academic conflict, you **must** let us know by **tomorrow (Thurs. 10/25) @ 5PM**
  – If you have academic accommodations, e.g. through OAE, please let us know by **tomorrow (Thurs. 10/25) @5PM** if possible.
  – If you do not have a workable laptop for the exam, you **must** let us know by **Friday 10/26 @ 5PM**. Limited charging outlets will be available for those who need them during the exam.
Plan For Today and Friday

- Classes
- Announcements
- Implementing a Linked List
  - Pointers
  - Dynamic memory
  - Classes
  - Testing
(+): Fast to add/remove at any point
(−): Slow to access certain nodes
Nodes

```c
struct Node {
    ??? data;
    ??? next;
};
```
struct Node {
    int data;
    ??? next;
};
struct Node {
    int data;
    Node next;
};
struct Node {
    int data;
    Node next;
};

This would be infinitely recursive!
Addresses

42 Wallaby Way
Addresses
Hey! What is your address?

42 Wallaby Way

int x

3

int x = 3;
cout << &x << endl;
Hey! What is your address?

int x
3

int x = 3;
cout << &x << endl;

The & operator is the address of operator. It gets the address of a variable in memory.
Plan For Today and Friday

• Classes
• Announcements
• Implementing a Linked List
  – Pointers
  – Dynamic memory
  – Classes
  – Testing
Addresses

Hey! What is your address?

int x = 3;
int *xAddress = &x;

42 Wallaby Way
Hey! What is your address?

int x = 3;

int *xAddress = &x;

Addresses

42 Wallaby Way

This is a variable named xAddress...
 Addresses

Hey! What is your address?

42 Wallaby Way

```c
int x = 3;
int *xAddress = &x;
```

That stores the address of an int...
Hey! What is your address?

42 Wallaby Way

int x

3

Hey! What is your address?

int x = 3;
int *xAddress = &x;

And its value should be the address of x.
Addresses

42 Wallaby Way

int x

3

int *xAddress

42 Wallaby Way

int x = 3;
int *xAddress = &x;
Addresses

```
int x = 3;
int *xAddress = &x;
```
Addresses

```cpp
int x = 3;
int *xAddress = &x;
```

*xAddress* is a pointer to *x*. It is a variable that “points to” another variable, meaning that it stores the address of another variable.
int x = 3;
int *xAddress = &x;

x is the pointee of xAddress. It is being pointed to by xAddress.
int x = 3;
int *xAddress = &x;

*xAddress = 5;
int x = 3;
int *xAddress = &x;

*xAddress = 5;
int x = 3;
int *xAddress = &x;

*xAddress = 5;
Dereferencing

```c
int x = 3;
int *xAddress = &x;

*xAddress = 5;
```
Dereferencing

```c
int x = 3;
int *xAddress = &x;

*xAddress = 5;
```
int x = 3;
int *xAddress = &x;

*xAddress = 5;
```cpp
int x = 3;
int *xAddress = &x;

*xAddress = 5;
```

The * operator is the dereference operator. It tells C++ to go to the variable at the address stored in that pointer.
Dereferencing Structs

Node n = ...
Node *ptr = &n;

(*ptr).value = 7;
Dereferencing Structs

Node n = ...
Node *ptr = &n;

(*ptr).value = 7;
Dereferencing Structs

Node n = ...
Node *ptr = &n;

(*ptr).value = 7;
Node n = ...
Node *ptr = &n;

(*ptr).value = 7;
Dereferencing Structs

Node n = ...
Node *ptr = &n;

(*ptr).value = 7;
Node n = ...  
Node *ptr = &n;  

ptr->value = 7;
int *xAddress

“nothing”?
Dereferencing nullptr

```
int *xAddress = nullptr;
cout << *xAddress << endl;
```
Dereferencing nullptr

```cpp
int *xAddress = nullptr;
cout << *xAddress << endl;
```

```
*** STANFORD C++ LIBRARY
*** A segmentation fault occurred during program execution.
*** This typically happens when you try to dereference a pointer
*** that is NULL or invalid.
*** Stack trace (line numbers are approximate):
*** 0x10ff14086  main()
```
int *xAddress = nullptr;
if (xAddress != nullptr) {
    cout << *xAddress << endl;
} else {
    cout << "nullptr!" << endl;
}
Dereferencing nullptr

```cpp
int *xAddress = nullptr;
if (xAddress) {
    cout << *xAddress << endl;
} else {
    cout << "nullptr!" << endl;
}
```
Garbage Pointers

```c
int *xAddress; // initially garbage
```
int *xAddress; // initially garbage
cout << xAddress << endl; // ???
int *xAddress; // initially garbage
cout << xAddress << endl; // ???
cout << *xAddress << endl; // likely crash!
Garbage Pointers

```cpp
int *xAddress; // initially garbage ❌
cout << xAddress << endl; // ???
cout << *xAddress << endl; // likely crash!

// always initialize pointers!
// (even just to nullptr)
int *xAddress = nullptr; // ✅
```
Using a Linked List

ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?
frontPtr->next->next->next->data = 2;
Using a Linked List

ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
Using a Linked List

ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
Using a Linked List

ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?
frontPtr->next->next->next->data = 2;

```plaintext
frontPtr

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>nullptr</td>
</tr>
</tbody>
</table>
```
Using a Linked List

ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?
frontPtr->next->next->next->data = 2;
ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?
frontPtr->next->next->next->data = 2;
Using a Linked List

ListNode *frontPtr = ...t;
// How do we e.g. modify the data in the fourth node?
frontPtr->next->next->next->data = 2;
Using a Linked List

ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
Using a Linked List

ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?
frontPtr->next->next->next->data = 2;
Using a Linked List

ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
Using a Linked List

ListNode *frontPtr = ...;
// How do we e.g. modify the data in the fourth node?

frontPtr->next->next->next->data = 2;
Reassigning Pointers

\[ a->next = b->next; \]
Reassigning Pointers

Setting two pointers equal to each other means they both point to the same place.
Reassigning Pointers

Tip: the types on the left- and right-hand sides must always match!

```c
a->next = firstNode;
```
Linked node problem 1

• Which statement turns this picture:

`ListNode node = {30, nullptr};`

- list
  - data 10
  - next

- list
  - data 20

• Into this?

`ListNode node = {30, nullptr};`

- list
  - data 10
  - next

- list
  - data 20

- list
  - data 30

A. list->next = node;
B. list->next->next = &node;
C. list->next->next->next = node;
Linked node problem 2

• Which statements turn this picture:

```
list
```

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

• Into this?

```
list
```

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

ListNode temp = {30, nullptr};

A. temp.next = list;         list = &temp;
B. temp = &list;             list = temp.next;
C. temp.next = list->next;   list->next = &temp;
int main() {
    int x = 0;
    addTwo(x);
    cout << x << endl;  // 2
}

void addTwo(int& x) {
    x += 2;
}
int main() {  
    int x = 0;
    addTwo(&x);
    cout << x << endl;  // 2
}
void addTwo(int *x) {
    *x += 2;
}
How do we print out the entire list, regardless of its length?
while (list != nullptr) {
    cout << list->data << endl;
    list = list->next;    // move to next node
}
Traversing a list?

while (list != nullptr) {
    cout << list->data << endl;
    list = list->next; // move to next node
}

This modifies our only reference to the head of the list!
Traversing a list (12.2)

Instead, let’s make another node pointer, and modify that:

```cpp
ListNode* current = list;
while (current != nullptr) {
    cout << current->data << endl;
    current = current->next;  // move to next node
}
```
Creating a List

frontPtr

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>nullptr</td>
</tr>
</tbody>
</table>
Creating a List

ListNode *vectorToLinkedList(const Vector<int>& v) {
    if (v.size() == 0) return nullptr;
    ListNode head = {v[0], nullptr};
    ListNode *currPtr = &head;
    for (int i = 1; i < v.size(); i++) {
        ListNode node = {v[i], nullptr};
        currPtr->next = &node;
        currPtr = &node;
    }
    return &head;
}
Creating a List

ListNode *vectorToLinkedList(const Vector<int>& v) {
    if (v.size() == 0) return nullptr;
    ListNode head = {v[0], nullptr};
    ListNode *currPtr = &head;
    for (int i = 1; i < v.size(); i++) {
        ListNode node = {v[i], nullptr};
        currPtr->next = &node;
        currPtr = &node;
    }
    return &head;
}

**Problem:** local variables go away when a function finishes. These ListNode will thus no longer exist, and the addresses will be for garbage memory!
int main() {
    Vector<int> v = {42, -3, 17, 9};
    ListNode *headPtr = vectorToLinkedList(v);
    if (headPtr) {
        cout << *headPtr << endl;
    }
}
Creating a List

main

myVector

42  -3  17  9

headPtr

vectorToLinkedList

data  next  data  next  data  next  data  next
42  -3  17  9  nullptr
Creating a List

main

myVector

| 42 | -3 | 17 | 9 |

headPtr

vectorToLinkedList

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>nullptr</td>
</tr>
</tbody>
</table>
We need a way to have memory that doesn’t get cleaned up when a function exits.
Plan For Today and Friday

• Classes
• Announcements
• Implementing a Linked List
  – Pointers
  – Dynamic memory
  – Classes
  – Testing
A New Kind of Memory

```
main

myVector

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>-3</td>
<td>17</td>
<td>9</td>
</tr>
</tbody>
</table>

headPtr

```

```
vectorToLinkedList

```

- `data` | `next`
- 42     |     |
- -3     |     |
- 17     |     |
- 9      | nullptr |
A New Kind of Memory

main

myVector

| 42 | -3 | 17 | 9 |

headPtr

vectorToLinkedList

data | next
--- | ---
42 |
data | next
-3 | 17 | 9
| | nullptr

Us: hey
C++, is there
a way to
make these
variables in
memory that
isn’t
amtually
cleaned up?
A New Kind of Memory

C++: sure, but since I don’t know when to clean it up anymore, it’s your responsibility…