

Lecture 7.md

Lecture 7 - Classes and Objects

Access Control

C++ classes can have **public** and **private** members.

DayOfYear.h

```
class DayOfYear{
    private:
        int day;
        int month;
    public:
        void setDay(int d);
        void setMonth(int m);
        void print();
};
```

Note:

1. `int day` and `int month` are **private** members
 - They can only be accessed by **function members** in the class
2. `setDay(int d)`, `setMonth(int m)`, and `print()` are **public** members
 - They can be accessed anywhere in the source code via an object

main.cpp

```
int main(){
    DayOfYear FirstOfJuly;
    DayOfYear Christmas;
    FirstOfJuly.day = 1;
    FirstOfJuly.month = 7;
}
```

Note:

1. `FirstOfJuly.day = 1;` and `FirstOfJuly.month = 1;` are errors
 - **private members** cannot be accessed outside the `FirstOfJuly` object

Access Control works by class, not by object. If we define another **member function** called `AddOne`:

```
void DayOfYear::AddOne(){
    DayOfYear temp;
    temp.day = 1;
    temp.month = 1;
    day = day + temp.day;
    month = month + temp.month;
}
```

This member function can access any **private** member of any object of the same type

- In this case, the type is `DayOfYear`

- `temp.day` and `temp.month` are related to the object declared in `AddOne`, which is `temp`
- `day` and `month` are related to the object that `AddOne` is called on

For example, in main:

```
DayOfDayYear birthday;
birthday.setDay(18);
birthday.setMonth(6);
birthday.AddOne();
```

The day and month fields noted above are related to the object `birthday`.

Code Organization (Header Files)

Good practice and convention is to place class definitions in **header files** and to place member function implementations in **.cpp files**.

Organizing our `DayOfDayYear` class into separate `.h` and `.cpp` files:

DayOfDayYear.h

```
#ifndef _dayofdayyear_h
#define _dayofdayyear_h

class DayOfDayYear{
private:
    int day;
    int month;
public:
    int getDay();
    int getMonth();
    void setDay(int d);
    void setMonth(int m);
    void print();
};

#endif
```

Notes:

1. `#ifndef` checks if the given **preprocessor directive** is not defined
2. `#define` defines the given **preprocessor directive**
3. `getDay()` and `getMonth()` functions are considered **accessor**, or **getter** methods
 - they return/output some data (usually private) from the object
4. `setDay()` and `setMonth()` functions are considered **mutator**, or **setter** methods
 - they change some data (usually private) inside the object

DayOfDayYear.cpp

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

int DayOfDayYear::getDay(){
    return day;
}
int DayOfDayYear::getMonth(){
    return month;
}
void DayOfDayYear::setDay(int d){
    day = d;
}
```

```

void DayOfYear::setMonth(int m){
    month = m;
}
void DayOfYear::print(){
    cout << day << "/" << month << endl;
}

```

Notes:

1. Need to include `DayOfYear.h` header file
 - The **member function** declarations are located inside `DayOfYear.h`
 - Will throw a **compile time error** if the header is not included

Compiling

Compiling code with class definitions in .cc files is the same as normal cpp compiling:

compiling main.o

```
g++ -c main.cc
```

compiling DayOfYear.o

```
g++ -c DayOfYear.cc
```

linking all (main.o and DayOfYear.o)

```
g++ main.o DayOfYear.o -o myprog.exe
```

Need for Initialization - Constructors

Sometimes we would like to create variables and assign them initial values. For example, `int x = 0;`

- But how do we do that for objects?

Constructors are functions that you write and are automatically called upon creation of an object

- The **constructor** is used to initialize objects easily
 - You can pass in initial parameters to the argument with the **constructor**
- **Constructors must** have the same name as the class
 - Constructors are members of the class
 - Constructors have **no** return type
 - Constructors are usually public (although they can be private)
- Constructor selection (which constructor the compiler chooses) happens at **runtime**
- C++ 2011 standard has a natural mechanism that allows for **default member initializers**
 - This simplifies initialization

One special case: The default constructor

- It has all the same properties as the constructor listed above
 - The default constructor *takes no argument*

DayOfYear.h

```
#ifndef _dayofyear_h
#define _dayofyear_h
```

```

class DayOfYear{
private:
    int day;
    int month;
public:
    DayOfYear();
    DayOfYear(int d,int m);
    DayOfYear(string s);
    int getDay();
    int getMonth();
    void setDay(int d);
    void setMonth(int m);
    void print();
};

#endif

```

main.cpp

```

int main(){
    DayOfYear birthday;
    DayOfYear christmas(25,12);
    DayOfYear mybirthday("12 16");
}

```

Notes:

1. `DayOfYear()` is the default constructor
2. `DayOfYear(int d,int m)` and `DayOfYear(string s)` are additional constructors
 - You can define as many constructors as you would like
 - However, every constructor **must have different types or amounts of arguments**
3. `christmas(25,12)` is both object creation and initialization (via a constructor)

Properties of Default Constructor

Every class must contain at least one constructor

- If you *define no constructor*, the compiler will define the **default constructor** for you
- If you *do define a constructor*, the compiler will **not** define a **default constructor** for you

For example:

DayOfYear.h

```

#ifndef _dayofyear_h
#define _dayofyear_h

class DayOfYear{
private:
    int day;
    int month;
public:
    DayOfYear(int d,int m);
    ...
    void print();
};
#endif

```

main.cpp

```
int main(){
    DayOfYear birthday;
}
```

The above code will return a **compile time error**

- **default constructor** is no longer being generated for you

Options to fix this **compile time error**

- You must either define the default constructor
- You can change the object initialization to `DayOfYear birthday(16,12);`

Beep Boop

Lecture 8 - Classes and Objects

Default Member Initializers

Many times we need simple initialization for class data members

- Prior to C++11, initialization was only allowed through constructors
- This is defined starting with the 2011 C++ standard (C++11)

DayOfYear.h

```
class DayOfYear{
    private:
        int day = 1;
        int month = 1;
    public:
        void setDay(int d);
        void setMonth(int m);
        void print();
};
```

Compiling code using **default member initialization** requires telling the compiler to use the C++11 standard

```
| g++ -std=c++11 main.cc
```

Destructors

Constructors are called when objects are created

- C++ also automatically calls a function when an object is destroyed/deleted

Destructors are functions that you write and are automatically called when objects are about to disappear

- The **destructor** does not destroy the object or clear the memory
 - The **destructor** simply defines the "final" actions of the destructor
- **Destructors must** have the same name of the class, **and** have a tilda (~) in front of the name
 - Destructors are members of the class
 - Destructors have **no** return type

- Constructors are usually public
- **Constructors** have **no** parameters
 - A way to think about this is that there is only the *default constructor*
- There can only be **one** constructor
- If there is no **Constructors** defined, C++ will define an empty (default) one for you

DayOfYear.h

```
class DayOfYear{
private:
    int day = 1;
    int month = 1;
public:
    DayOfYear(); //constructor
    ~DayOfYear(); //destructor
    void setDay(int d);
    void setMonth(int m);
    void print();
};
```

DayOfYear.cpp

```
DayOfYear::~DayOfYear(){
}
```

Notes:

- The provided constructor does nothing

'This' Pointer

Every method is given the address of the object on which it is invoked

- The *address* is stored in the **this** pointer
 - Initialized to point to the object on which the method is invoked
- Think of it as (read-only) variable of type pointer to the class
 - `DayOfYear* this;`
- No way to initialize **this**
 - Compiler defines the **this** pointer for you
 - No way to redefine/modify **this** pointer
- The **this** pointer is always defined
 - Will never be `null` if object is defined
- Accessible only within member functions

DayOfYear.cpp

```
void DayOfYear::setDay(int d){
    day = d;
}
void DayOfYear::setDay(int d){
    day = d;
    (*this).day = d;
    this->day = d;
}
```

Notes:

1. The three lines of code in `DayOfYear::setDay` above have the *same* functionality
2. You need brackets for `(*this).day`
 - Field access operator `.` has higher precedence than dereference operator `*`
3. Arrow operator `->` similar idea to arrow operator in C
 - Access value of field at address of object
4. `this.day` is not valid
 - Syntax error, attempting to access field of address

Notes about printing objects:

- Can print address `this`
- No way to print `(*this)`
 - For example, `cout << *this << endl;`
 - `*this` refers to the value of the object, which doesn't have an identifiable print value (for the compiler)
 - Should printing this print the data members, or an arbitrary string?
 - Not defined
- One way to print objects
 - Overload insertion operator `<<`
 - No `toString()` like in Java

Type Conversion

Given the class definition and main.cpp:

DayOfYear.h

```
class DayOfYear{
private:
    int day = 1;
    int month = 1;
```

```

public:
    DayOfYear();
    DayOfYear(int d,int m);
    ~DayOfYear();
    void setDay(int d);
    void setMonth(int m);
    void print();
};
```

main.cpp

```
DayOfYear x("6","8")
```

Will result in compile-time error.

However,

main.cpp:

```
DayOfYear x(5.1,2.34)
```

Will compile successfully.

Why?

- C++ compiler attempts to match types to function parameters
 - In this case, `float` types can be converted easily to `int` types by flooring.
 - Whereas the **type conversion** from `string` to `int` is not defined

Address of Object

Outside of member functions, you can access address by using the referencing operator `&`

- Get address of object with `&object`
- Same as C for referencing pointers

Garbage Collection

C++ objects defined on the memory **stack**

- LIFO (Last-in, First-out)
- The first member functions to be deleted are the last member functions to be created
 - Deletion is *reverse* order of creation

Lec_9.md

Lecture 9 - C++ Input/Output (IO)

Sept.30/2020

Streams

What exactly is a stream

- A sequence of characters
- Allows connection between different hardware components
 - Keyboard
 - Input as a stream
 - Display
 - Output as a stream

However, there is disconnect between *input* and *output* streams

- Functionality of streams
 - Perform the necessary conversion between internal data representations and character streams
 - Act as agents to convert data for the hardware to interpret
 - Can send streams to different places
- Input Stream: `cin`
- Output Stream: `cout`
- Another output stream: `cerr`
 - Used for error messages

Input Stream

`cin` attempts to extract input values from input stream

`main.cpp`

```
int main(){
    int x;
    cin >> x;
}
```

Notes:

1. Input stream value placed inside variable `x`
 2. A keyboard buffer is created before `cin` receives input stream
- Values are stored temporarily, s.t. they can be changed before the *enter* key is pressed and the stream is sent to `cin`
 - Imagine deleting a character before pressing enter

Extraction from Input Stream

For example, imagine typing the integers

```
'1' '2' '3' '' '2' '0' '4'
```

into the executable for `main.cpp`

main.cpp

```
int main(){
    int x,y;
    cin >> x >> y;
}
```

Procedure for `cin` extracting from input stream:

1. `cin` reads '1'
2. `cin` moves onto next character
3. `cin` reads '2'
4. `cin` moves onto next character
5. `cin` reads '3'
6. `cin` moves onto next character
7. `cin` tries to read ' ', but since this is not an integer, it does not read this value into the integer
 - o `x` becomes the *combined value* of the read values from the existing stream
 - in this case, $x=123$
8. `cin` moves onto next character
9. And the same process continues for $y=204$

The values of `x,y` are

```
x=123
y=204
```

Note: Streams are *not flushed* after the line of `cin` completes.

main.cpp

```
int main(){
    int x,y;
    cin >> x
    ...
    ...
    cin >> y;
}
```

And the same input is typed into the executable

```
'1' '2' '3' '' '2' '0' '4'
```

The same behaviour holds as for the above case, so

```
x=123
y=204
```

Delimiters

Delimiters are values at which `cin` decides where a sub stream starts and ends

- **Delimiters** define where the values for `cin` assignment start/stop
 - For example, that 123 is an integer, but 12 3 is actually two integers

The value ' ', '\n' and '\0' are called **Delimiters**.

- ' ' (the space character) is a very common delimiter.

Basically,

If there has not been a stream given to `cin`

`cin` will look for stream (by prompting command prompt input)

If `cin` reaches a **delimiter** before the stream is **exhausted** (e.g. before all of the characters in the stream are read)

`cin` will assign the value it read into the variable

However if `cin` does *not* exhaust the stream (e.g. there is a **delimiter** in the middle of the stream), it will *keep* the stream (as an internal variable) for future `cin` calls;

Int and Floats in Stream Extraction

For the below `main.cpp`:

`main.cpp`

```
int main(){
    int x;
    float y;
    cin >> x >> y;
}
```

If the input stream is:

'1' '2' '3' '' '2' '.' '4'

Notes:

1. `cin` will read `x=123`
2. `cin` will read `y=2.4`

However, if the input stream is:

'1' '2' '3' '2' '' '4'

Notes:

1. `cin` will read `x=123`
 2. `cin` will read `y=0.4`
- In the first case, the ' ' character is the **delimiter**
 - In the second case, the '.' character is the **delimiter**
 - This is because '.' is a valid **delimiter** for `int` values
 - Whereas '.' is *not* a **delimiter** for float values
 - The decimal point is a *part* of float values

For the below `main.cpp`:

`main.cpp`

```
int main(){
    int x,y;
    cin >> x >> y;
}
```

If the input stream is:

'1' '2' '3' '.' '4'

Notes:

1. `cin` will read `x=123`
2. `cin` will **not** read `y`
 - In this case, `cin` tries to read from the '.' character but **fails**

Strings in Stream Extraction

`main.cpp`

```
int main(){
    string firstName;
    string lastName;

    cin >> firstName >> lastName;
}
```

If the input stream is:

'T' 'o' 'm' ' ' 'L' 'i'

Notes:

1. `cin` will read `firstName="Tom"`
2. `cin` will read `lastName="Li"`

What about errors in Input Stream?

`cin` fails silently

- Execution continues, leaving the variable and input stream unaffected;

`main.cpp`

```
int main(){
    int x = 9;
    cin >> x;
}
```

If the input stream is:

'T' 'e' 'n' ' ' '2'

`cin` will **not** affect the value of `x`, and fails silently without any indication

- On ECF computers, this is the behaviour of `cin`
- On *certain* compilers, `cin` will set the value of `x` to null

`cin` will not read space characters

- Since space characters (' ') are **delimiters**
- So how do we read strings with spaces?
 - Two ways:
 - Overwrite the `cin` function to treat '' not as **delimiters** but as **characters**
 - This is generally a bad idea. Think about it.
 - Use the `getline` function instead of `cin`

Flags (`cin`)

`cin` has certain **flags**

- **Flags** are set to true/false after every `cin` extraction operation
- These **flags** reflect what happened in the previous `cin` operation
- One specific flag is the `fail` flag
 - The `fail` flag is set to true if `cin` fails to extract
 - Otherwise, set to false
- Developer must check the flags
 - C++ `cin` fails silently (will *not* inform you)

The Fail flag

For the below code:

`main.cpp`

```
#include <iostream>
using namespace std;
int main(){
    int anInteger;
    cin >> anInteger;
    if(cin.fail()){
        cout << "bad input" << endl;
    }else{
        cout << "read from cin~!" << endl;
    }
    return 0;
}
```

Notes:

1. `cin.fail()` returns false if the line `cin >> anInteger` read successfully
 - `cin.fail()` returns true, the last call to `cin` failed
2. Notice that the `cin.fail()` uses the **field access operator**
 - So what is `cin` ??

Answer: `cin` is an object (instance) of **istream**

Remember that include statement we always use?

- `#include <iostream>`
 - `iostream` is split into two additional include files:
 - `istream`
 - `ostream`
- `istream` defines the `cin` object/instance

- `ostream` defines the `cout` object/instance

The Insertion Operator

So... what exactly is the **insertion operator** (this thing `>>`)

- It's an **operator**
- Acts as a function call
 - Calls with parameters (`cin`, `var`)
 - Assign the read characters from the stream into `var`

Lec_10.md

Lecture 10 - C++ Input/Output (IO)

Nov.1/2020

The Fail flag

One thing to note:

main.cpp

```
int main(){
    int x = 9;
    string y;
    cin >> x;
    ...
    cin >> y;
}
```

If the input stream is:

'T' 'e' 'n' ' ' '2'

cin will fail at cin >> x;

- Attempting to place characters into int

However, cin >> y will run and place 'ten' into y

- Buffer has not been cleared
- Fail flag is *still* set to true

Checking for Valid Cin

main.cpp

```
#include <iostream>
using namespace std;
int main(){
    int anInteger;
    bool retry = true;
    while(retry){
        cin >> anInteger;
        if(cin.fail()){
            cout << "bad input" << endl;
        }else{
            cout << "read from cin~!" << endl;
        }
    }
    return 0;
}
```

If the input stream is:

```
'T' 'e' 'n' '' '2'
```

Notes:

- This is an *infinite loop*
 - On the first iteration of `while` loop:
 - `cin` attempts to place 'T' into variable `anInteger`
 - Fails to do so, and raises **Fail flag** because of type mismatch.
 - The `cin` stream is **not modified** between consecutive `cin` calls
 - The `cin` stream remains because the initial call raised the **Fail flag**
 - `cin.fail()` will always evaluate to true, since the stream **has not changed**
 - `bool retry` never gets updated
- `anInteger` does not change if fail

So how do we deal with this issue?

- Flags are **persistent**
 - Flags remain until you reset them
 - No input until Flags are set

`main.cpp`

```
#include <iostream>
using namespace std;
int main(){
    int anInteger;
    bool retry = true;
    while(retry){
        cin >> anInteger;
        if(cin.fail()){
            cout << "bad input" << endl;
            cin.clear();
            cin.ignore(1000, '\n');
        }else{
            cout << "read from cin~!" << endl;
        }
    }
    return 0;
}
```

Notes:

- `cin.clear();`
 - This function *clears the flags*
 - Will turn off the **Fail flag**
 - Without calling `cin.clear();`, subsequent `cin` calls will not run
- `cin.ignore(1000, '\n');`
 - This function ignores characters in the stream
 - `cin.ignore();` takes two parameters. Either:
 - a. Will *ignore 1000 characters*
 - 1000 is an arbitrary example, some systems have a limit of 256 characters per stream
 - b. Will *ignore until '\n' is found*
 - Handle errors in this order:
 - Clear flags with `cin.clear();`, **and then** ignore stream characters with `cin.ignore();`
 - The reverse order may not work, since `cin.ignore()` depends on flags

The Ignore Function

`cin.ignore()` is used to discard part of the stream. For the below example:

`main.cpp`

```
int main(){
    int x = 9;
    cin >> x;
}
```

If the input stream is:

'T' 'e' 'n' ' ' '2'

There are different ways to implement `cin.ignore()`:

1. `cin.ignore(1000, '\n');`
 - Clears the first 1000 character of the stream, or until '\n'
2. `cin.ignore(1000, ' '');`
 - Ignores spaces
 - More elegant way to clear problematic parts of the stream
 - Will read the '2' into x
3. `cin.ignore(3, '\n')`
 - Ignores 3 characters or until '\n' is reached
 - Can be used to ignore first part of a word/name
 - Will read the '2' into x

End of File

We would like to read inputs until the user has no more inputs to enter

- In APS105, we sometimes used '-1' to indicate the *last element* of sequence of integer inputs
 - Limits us to only *positive* integers

Enter the **End of File** (eof) error

- Indicates that there is no more input to be expected
- Encountering the **eof** when more input is expected raises **two** flags in `cin`:
 - **fail** flag is raised
 - **eof** flag is raised

On the terminal (when running executables), **eof** can be induced by using **ctrl+d**

- Special character, like '\0' or '\n'
- Is not included after '\n'
 - User must *specifically* enter **eof** using **ctrl+d**

`main.cpp`

```
#include <iostream>
using namespace std;
int main(){
    int x,sum=0;
    bool more = true;
    while(more){
```

```

    cin >> x;
    if(cin.eof()) more = false;
    else sum=sum+x;
}
}

```

Notes:

- Read characters through `cin` when `more==true`
- As soon as `eof` is provided, stop
- `cin.eof()` is an **accessor** method that returns the `eof` flag

This works for input:

```
'2' '0' '4' '' '1' '1' '3' eof
```

`sum=317`

- $317=204+113$

However for this input:

```
'2' '0' '4' '' '' 't' 'e' 'n'
```

`sum=408`. Why?

- Notice the two spaces
 - Upon reaching the second space character (second **delimiter**), `cin` will read from stream again
 - The issue is the `eof` flag is still `false`, so the `if` statement goes to `else` again, so `sum=sum+x;` runs again.

How do we fix this?

`main.cpp`

```

#include <iostream>
using namespace std;
int main(){
    int x,sum=0;
    bool more = true;
    while(more){
        cin >> x;
        if(cin.fail()){
            if(cin.eof()){
                more = false;
            }else{
                sum=sum+x;
            }
        }
    }
}

```

■ Lec_11.md

Lecture 11 - C++ Input/Output (IO)

Oct.6/2020

The getline Function

Last lecture an interesting property of strings and **delimiters** was brought up:

- Trying to `cin "hi there"` will only place "hi" into the variable, as the " " character is a **delimiter**

The `getline` function

- Possible to get the entire line, *spaces included*, into a string
- Takes the **entire** input stream and places it into the string argument

`main.cpp`

```
#include <string>
int main(){
    string fullName;
    getline(cin,fullName); //or cin.getline(fullName,256)
    cout << fullName;
}
```

Notes:

1. You need to `#include <string>` to use `getline`
 - `getline` defined in `string` header
2. `getline(cin,fullName);`
 - Notice that we pass `cin` to `getline`
3. `cin.getline(fullName,256);`
 - Another way to call `getline` is `cin.getline(fullName,256)`
 - Equivalent funciton call to point 2
 - Reads 256 characters

User-Created Streams

The notion of streams in c++ is incredibly powerful, so what if we want to create our own input streams?

- **String Streams**
 - Read in and out of a string
- **File Streams**
 - Read in and out of a file on the hard drive

String Streams

We want to make **streams** out of strings

- This way we can read in/out of strings as **streams** easily

main.cpp

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main(){
    int anInteger;
    string inputLine;
    inputLine="204 113";

    stringstream myOwnStream(inputLine);
    myOwnStream >> anInteger;
    myOwnStream >> anInteger;

    return 0;
}
```

Notes:

1. Must include `<sstream>` header to use **string stream**
 - **String stream** defined in `<sstream>` header
2. `stringstream myOwnStream(inputLine);`
 - Creates a new **stream** called "myOwnStream"
 - Initializes this stream with `string "inputLine"`
3. `myOwnStream >> anInteger;`
 - Reads an integer from stream "myOwnStream" into "anInteger"
 - *Exactly* the same functionality as `cin`
 - General functionality: read from input stream into variable
 - `sstream` reads from **string stream** into variable
 - `cin` reads from **input stream** into variable

Properties of String Stream

- Just like `cin`, it **fails** silently
 - `sstream` has its own **fail flag**

Uses of String Stream

- Very useful when you have "line-oriented input"
 - Better able to deal with incorrect inputs
- Use `getline` to grab entire line
 - *Then* build `sstream` out of it and read values

main.cpp

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main(){
    int anInteger;
    string inputLine;

    getline(cin,inputLine);//OR cin.getline(inputLine,256);
```

```

stringstream myOwnStream(inputLine);
myOwnStream >> anInteger;
myOwnStream >> anInteger;

return 0;
}

```

Notes:

1. `getline` into a string variable, and then `sstream` the variable to make a **string stream**
2. Cannot use `>>` **insertion operator** on a **string**
 - Call `>>` on **streams**, like **string stream**

File Streams

What about if we wanted to read from a file on a hard drive?

- Different from `cin` or `sstream`, as reading from a file has no directly observable UI elements
- Use `ifstream` and `ofstream`

Input File Streams

Reading from file

main.cpp

```

#include <iostream>
#include <fstream>

using namespace std;

int main(){
    ifstream inFile;
    int a;

    inFile.open("inputfile");

    inFile >> a;
    cout << a << endl;

    inFile >> a;
    cout << a << endl;

    inFile.close();

    return 0;
}

```

If the file contains:

604 233 1233

First `cout` will print "604" Second `cout` will print "233"

Notes:

1. `#include <fstream>`
 - `<fstream>` includes methods for **file stream** input and output

- Contains `ifstream` (**input file stream**) and `ofstream` (**output file stream**)
2. `ifstream inFile;`
 - Defines an object of **input file stream** type with name "inFile"
 3. `inFile.open("inputfile");`
 - Opens a file with name "inputfile"
 - Usually "inputfile" includes the **extension**
 - For example, "inputfile.txt"
 - File "inputfile" **must** be in the same directory as the **executable** for the program
 4. `inFile >> a;`
 - Read from the **input file stream** into variable a
 - Same usage as `cin`
 - Can also check for **fail flags**, same as `cin`
 5. `inFile.close()`
 - close the **file stream** object
 - Just remember to call for **file streams**
 - May not noticeable break anything, but could be an issue for **output streams**

Properties of `ifstream`

- One `ifstream` object can open **1** file at once
 - Having multiple **file streams** open requires multiple `ifstream` objects
 - That being said, you can reuse `ifstream` objects to open multiple files
 - You just need to `.close()` a file before `.open()` a new file
- What happens if the file you are trying to open ("inputfile") has higher permissions (e.g. is not readable)
 - `ifstream` will throw an exception
 - If there is an issue with *opening the file* (e.g. `inFile.open();`)
 - For example, file could not be found, file is not readable, etc..
 - OS will **throw exception**
 - Program will **stop running**
 - If there is an issue with *reading from the file* (e.g. `inFile >> a;`)
 - `inFile` will raise **fail flag** silently
 - Same behaviour as `cin`
- What does `inFile >> a;` actually do?
 - Given file with first line: "604 233 1233"
 - `inFile >> a;` will put the first integer (since a is an integer) into a
 - In this case, "604"
 - It does not put the entire line into a
 - Remember, the **stream** is attached to the file
 - It puts the first variable (detecting type and noticing **delimiters**) into a
 - The next `inFile >> a;` call will put the second integer into a
 - In this case, "233"
- `inFile.close();`
 - Unattaches the stream from the file
 - *Cannot* use this stream any more

Output File Streams

Reading to (or printing to) file

`main.cpp`

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
    ofstream outFile;
    int a=8;

    outFile.open("outputfile");

    outFile << a;

    outFile.close();

    return 0;
}
```

After execution, the file will contain

8

Notes:

1. #include <fstream>
 - o <fstream> includes methods for **file stream** input and output
 - Contains **ifstream** (**input file stream**) and **ofstream** (**output file stream**)
2. ofstream outFile;
 - o Defines an object of **output file stream** type with name "outFile"
3. outFile.open("outputfile");
 - o Opens a file with name "outputfile"
 - Usually "outputfile" includes the **extension**
 - For example, "file.txt"
 - o File written to disk is sent to same directory as executable by default
4. outFile << a;
 - o Writes the variable a from the **output file stream** into defined file
 - o Uses the << operator
 - Similar semantics to cout
5. outFile.close()
 - o close the **file stream** object
 - Just remember to call for **file streams**
 - May not noticeable break anything, but could be an issue for **output streams**

Properties of **ofstream**

- Truncate vs Append
 - o By default, **ofstream** truncates by default
 - **ofstream** places output variables at *start* of file
 - o Can define how file is open
 - Can indicate "append" option to **ofstream.open()**
- **ofstream << a;** writes variable to file, on a single line
 - o Add a **\n** character to add multiple lines to file
 - o Can also use **endl** to end line and carriage return

beep boop~

■ Lec_12.md

Lecture 12 - C++ Input/Output (IO)

Oct.7/2020

Output Streams

The **output stream** `ostream` acts as an agent that transfers data between your screen and the program

- Similar to the **output file stream** `ofstream`, but *different*
- `cout` writes to **output stream**

Output Stream Manipulators

`cout` has **output manipulator** functions that allow for formatting of output

main.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;
int main(){
    int myVar = 503;
    cout << myVar << endl;
    cout << setw(8) << myVar << endl;
    cout << myVar << endl;
    cout << setfill('0') << setw(8) << myVar << endl;
    cout << setw(8) << myVar << endl;
    return 0;
}
```

Notes:

1. `#include <iomanip>`
 - header file containing `cout` **output manipulator** functions
2. `setw(8)`
 - By default, the output stream is **left aligned/justified**
 - This means each line starts from the *left* side of the console/terminal
 - `setw(x)` adds x amount of blank/fill spaces
 - Can use to format `cout` output
 - `setw(8)` adds 8 fill spaces
3. `setfill('0')`
 - `setfill('0')` replaces '' characters generated by `setw` with '0'
 - `setfill` is **persistent**
 - Calling `setfill` once will replace all blank space (fill characters) with given parameter
 - Will replace fill chars for all future `setw` function calls

main.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;
int main(){
    float myVar = 3.2876891;
    cout << myVar << endl;
    cout << setprecision(2) << myVar << endl;
    return 0;
}
```

Notes:

1. `setprecision(2)`
 - `setprecision(x)` sets the precision of floating point numbers to x
 - `setprecision(2)`, will set precision to 2
 - `cout` will print "3.3"
 - Note that `setprecision` *will round* the output number, *not truncate*

I/O Redirection

I/O streams can be redirected to files so you can I/O from files instead of standard input (keyboard) and standard output (screen)

- This works at the OS level (outside the program executable)
- Useful for debugging your code

Can be done at the command prompt:

```
% myprog.exe > outfile
all output ( cout ) goes to outfile

% myprog.exe < infile
all input ( cin ) taken from infile

% myprog.exe < infile > output
all input ( cin ) taken from infile and all output ( cout ) goes to outfile

% myprog.exe >& outfile
all output ( cout and cerr ) goes to outfile
```

Pointers, Scopes, and Arrays

Pointers! and Dynamic Allocation!

Pointers

How are variables stored?

- Stored in memory
 - But what does memory look like???

address	memory contents	symbol
---------	-----------------	--------

address	memory contents	symbol
0x0000AB00	5	x
0x0000AB04	3.8	y
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14		
0x0000AB18		

Something like this ^. Note:

- Each address (**hexadecimal number**) differs by 4
 - Each address is of size 4 bytes
 - Or, 32 bits, since
 - $4 \text{ bytes} * 8\text{bits/byte} = 32 \text{ bits}$
- Addresses in most computers are **byte addressable**
 - Meaning that the smallest unit of space that can be **addressed** is a **byte**

When you write `int x;`, a space in memory is reserved

- How much space?
 - Enough to hold an `int` variable

main.cpp

```
int main(){
    int x;
    float y;
    x = 5;
    y = 3.8;
    cout << x << endl;
    cout << y << endl;
}
```

address	memory contents	symbol
0x0000AB00	5	x
0x0000AB04	3.8	y
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14		
0x0000AB18		

Pointers are used to **address** other variables of the same type

- Pointers are variables that contain the address of other variables

Dereference Operator

main.cpp

```
int main(){
    int x;
    float y;
    x = 5;
    y = 3.8;
    cout << x << endl;
    cout << y << endl;
    ...
    int* px;
    float* py;
    px = &x;
    py = &y;
}
```

address	memory contents	symbol
0x0000AB00	5	x
0x0000AB04	3.8	y
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14	0x0000AB00	px
0x0000AB18	0x0000AB04	py

Notes:

1. int* px;
 - o Define **pointer** variable of type **int**
2. float* py;
 - o Define **pointer** variable of type **float**
3. & operator
 - o The **ampersand** & is the **reference** operator
 - Get the address of the thing the operator is operating on
 - o **reference** operator is an **overloaded** operator
 - This means that the functionality of the & operator depends on the *context* on which it is called
4. px = &x;
 - o Set value of px **to address** of x
 - o px, x of type **int***, **int** respectively
5. py = &y;
 - o Set value of py **to address** of y
 - o py, y of type **float***, **float** respectively

Reference Operator

main.cpp

```
int main(){
    int x;
    float y;
    x = 5;
    y = 3.8;
```

```

cout << x << endl;
cout << y << endl;
...
int* px;
float* py;
px = &x;
py = &y;
*px = 3;
*py = 5.2;
}

```

address	memory contents	symbol
0x0000AB00	3	x
0x0000AB04	5.2	y
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14	0x0000AB00	px
0x0000AB18	0x0000AB04	py

Notes:

1. * operator
 - o The asterisk * is the **dereference** operator
 - Access the value at the address of the thing the operator is operating on
2. *px = 3;
 - o Essentially saying:
 - Change the value at the address of px to 3
 - o Sets the value of x to 3
3. *py = 5.2;
 - o Essentially saying:
 - Change the value at the address of py to 5.2
 - o Sets the value of y to 5.2

Pointer Schematics

Just like schematics in circuits (ECE212), **pointer schematics** describe the *mapping* between pointers and variables

Lec_13.md

Lecture 13 - Pointers, Scopes, and Arrays

Oct.8/2020

Pointers and the New Keyword

Using pointers to change values of addresses and reference variables is cool

- But we can also use pointers to create *new variables* at **run-time**
 - These variables are **dynamically allocated**

In C programming (APS105), we use `malloc` to allocate a certain amount of memory

- In C++, there is a syntactically sweeter way to allocate memory
 - **new keyword**

`main.cpp`

```
#include <iostream>
using namespace std;
int main(){
    int x = 8;
    int* px = &x;
    cout << *px << endl;
    px = new int;
    *px = 5;
    cout << *px << endl;
    px = new int;
    *px = 6;
    cout << *px << endl;
    return 0;
}
```

address	memory contents	symbol
0x0000AB00	8	x
0x0000AB04	5	
0x0000AB08	6	
0x0000AB0c		
0x0000AB10		
0x0000AB14	0x0000AB08	px
0x0000AB18		

Notes:

1. `int* px = &x;`
 - Create a **pointer** variable of type `int*` and of name `px`
 - Point `px` to the *address* of `x`
2. `px = new int;`

- Defines a **dynamically allocated** piece of memory (of size `int` - 4 bytes)
 - The amount of memory allocated by `new` depends on the **type** of the variable
3. Notice above that there are no symbols for memory addresses `0x0000AB04` and `0x0000AB08`
- So how can we change the value of `0x0000AB04` (which is 5)?
 - We can't, because we do not have a pointer or address of this memory
 - This is called a **memory leak**

Memory leaks are bad for your program

- Cannot reference that memory address

main.cpp

```
#include <iostream>
using namespace std;
int main(){
    int x = 8;
    int* px = &x;
    cout << *px << endl;
    px = new int;
    *px = 5;
    cout << *px << endl;
    px = new int;
    *px = 6;
    cout << *px << endl;
    delete px;
    return 0;
}
```

address	memory contents	symbol
0x0000AB00	8	x
0x0000AB04	5	
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14	0x0000AB08	px
0x0000AB18		

Notes:

1. `delete px;`
 - **Important:**
 - The `delete` keyword **deletes** the value (frees the memory) at **the value** pointed to by `px`
 - It **does not** delete the address stored by `px`
 - `delete` cannot delete **non-dynamic** data
2. Notice that `px` **still** points to an address
 - That address (in this case, `0x0000AB08`) is **empty**
 - Attempting to use `px` to change the value at the address is a *bug*
 - Make sure to **not use this pointer address**

main.cpp

```
#include <iostream>
using namespace std;
int main(){
    int x = 8;
    int* px = &x;
    cout << *px << endl;
    px = new int;
    *px = 5;
    cout << *px << endl;
    px = new int;
    *px = 6;
    cout << *px << endl;
    delete px;
    px = nullptr;
    px = NULL;
    return 0;
}
```

address	memory contents	symbol
0x0000AB00	8	x
0x0000AB04	5	
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14		px
0x0000AB18		

Notes:

1. px=nullptr
 - o This will set the pointer variable to `nullptr`
 - A null pointer is an empty pointer - *evaluates to 0*: zero, nilch, nada.
 - Indicates that the pointer is not pointing to anything
 - Good practice to set the pointer to `null` after `delete`
2. px=NULL
 - o Identical to the command in 1. (`px=nullptr`)

Pointers to Structs

Also throwback to `structs` in C programming (APS105)

- We can also use structs in C++

main.cpp

```
#include <iostream>
using namespace std;

struct node{
    int ID;
    float value;
}

int main(){
```

```

struct node* ptr;
ptr = new struct node;

return 0;
}

```

Notes:

1. struct node
 - o Same struct definition as C
 - o Defines an abstract data type with multiple **fields**
2. ptr = new struct node;
 - o **Dynamically allocate** new struct node

main.cpp

```

#include <iostream>
using namespace std;

struct node{
    int ID;
    float value;
}

int main(){
    struct node* ptr;
    ptr = new struct node;

    (*ptr).ID = 2;
    (*ptr).value = 6.2;

    return 0;
}

```

Notes:

1. (*ptr).ID = 2;
 - o Access the **field ID** of the **data type** at address **ptr**
 - o Why are there parentheses around ***ptr** ?
 - Shouldn't ***ptr.ID = 2;** work as well?
 - The **field access operator .** has higher **precedence** than the **dereference operator ***
 - The **.** operator will run first, which will return an error, since **ptr** is not a variable that has fields that the compiler recognizes
 - o Access the **field ID** of ***ptr** and set it equal to 2
 - 2. (*ptr).value = 6.2;
 - o Access the **field value** of ***ptr** and set it equal to 6.2

main.cpp

```

#include <iostream>
using namespace std;

struct node{
    int ID;
    float value;
}

int main(){
    struct node* ptr;

```

```

ptr = new struct node;

ptr->ID = 2;
ptr->value = 6.2;

return 0;
}

```

Notes:

1. `ptr->ID = 2;`
 - o This is **syntactic sugar** in C++, as in
 - This operator `->` does the same thing as `*(variable)`.
 - o `ptr->ID` has ***identical*** functionality to `(*ptr).ID`

Pointer in Structs

We've seen pointers used to refer to structs and **dynamically allocate** them

- We can also have pointers *inside* structs

main.cpp

```

#include <iostream>
using namespace std;

struct node{
    int ID;
    struct node* next;
}

int main(){
    struct node* head;
    struct node* ptr;

    head = new struct node;
    head->ID = 0;
    head->next = nullptr;

    ptr = new struct node;
    (*ptr).ID = 1;
    (*ptr).next = nullptr;

    (*head).next = ptr;
    ptr = NULL;
    return 0;
}

```

Notes:

1. First, we can recognize that this is the basic implementation for a **linked list**
2. `head->next = nullptr;`
 - o Set the next item/node in the linked list (after the head) to a `nullptr`
3. `(*ptr).next = nullptr;`
 - o Set the next item/node in the linked list (after the ptr node) to a `nullptr`
 - o Remember that `(*ptr).` is ***identical*** to `ptr->`

What if we have more than two elements?

main.cpp

```
#include <iostream>
using namespace std;

struct node{
    int ID;
    struct node* next;
}

int main(){
    struct node* head;
    struct node* ptr;
    struct node* ptr2;

    head = new struct node;
    head->ID = 0;
    head->next = nullptr;

    ptr = new struct node;
    (*ptr).ID = 1;
    (*ptr).next = nullptr;

    ptr2 = new struct node;
    (*ptr2).ID = 2;
    (*ptr2).next = nullptr;

    (*head).next = ptr;
    (*ptr).next = ptr2;
    return 0;
}
```

Notes:

- How can we access the `ID` field of `ptr2` ?
 - We can easily use `ptr2->ID`, but what if we **only** know the address of the head node?
 - `head->next` will return `ptr`
 - next field in `head` points to address of `ptr`
 - `head->next->next` will return `ptr2`
 - next field in `head` points to address of `ptr`, same as above
 - Then next field in `ptr` points to address of `ptr2`
 - So `head->next->next` will give `ptr2`
 - Can then use `head->next->next` to reference **fields** in `ptr2`

Pointers to Pointers

We've already seen pointers pointing to regular variables

- `int* p = &x`, where `x` is a variable of type `int`

But what about pointers to pointers?

- e.g. what if we want a pointer to point to another pointer?

main.cpp

```
#include <iostream>
using namespace std;

struct node{
    int ID;
    struct node* next;
```

```

}

int main(){
    int** p2p;
    int* p;
    int* q;

    p = new int;
    *p = 5;

    p2p = &p;
}

```

address	memory contents	symbol
0x0000AB00	5	
0x0000AB04		
0x0000AB08		
0x0000AB0c		
0x0000AB10	0x0000AB00	p
0x0000AB14	0x0000AB10	p2p
0x0000AB18		q

Notes:

1. int** p2p;
 - o Create a **pointer to pointer** of type `int`
 - Where a **pointer** points to the address of a variable
 - A **pointer to pointer** points to the address of a **pointer**
2. int* p;
 - o Define a pointer that points to a memory address
3. *p = 5
 - o Set the value at the address pointed to by `p` to 5
4. p2p = &p;
 - o Take the **address** of `p` (a pointer), and store that address in `p2p`
 - Notice that `p2p` has the contents `0x0000AB10`, or the **address** of `p`

main.cpp

```

#include <iostream>
using namespace std;

struct node{
    int ID;
    struct node* next;
}

int main(){
    int** p2p;
    int* p;
    int* q;

    p = new int;
    *p = 5;

    p2p = &p;
}

```

```

q = *p2p;

*q = 8;

cout << **p2p;

return 0;
}

```

address	memory contents	symbol
0x0000AB00	8	
0x0000AB04		
0x0000AB08		
0x0000AB0c		
0x0000AB10	0x0000AB00	p
0x0000AB14	0x0000AB10	p2p
0x0000AB18	0x0000AB00	q

Notes:

1. q = *p2p;
 - o Assign to q the value at the address pointed to by p2p
 - In this case, assign 0x0000AB00 to q
 - Why is the assign 0x0000AB00 ?
 - Since 0x0000AB10 is the value in p2p .
 - This is the content/value of p2p
 - This is the address pointed to by p2p
 - The value at 0x0000AB10 is 0x0000AB00
 - q now points to 0x0000AB00
 - In other words, q and p now point to **exactly** the same thing
 - 2. *q = 8;
 - o Assign the value at the address pointed by q to "8"
 - Remember from part 1. that q and p point to the same thing.
 - Assign the value at 0x0000AB00 to "8"
 - *p would also now be "8", since q and p point to the same thing
 - 3. cout << **p2p;
 - o Print the value of the (the value store by pointer p2p)
 - o Lets break this down:
 - Recall that q is identical to *p2p
 - This means that *p2p contains the value 0x0000AB00
 - So, **p2p refers to the value at address 0x0000AB00
 - Which is "8"
 - o So cout << **p2p will print "8", or the value of *p or *q

■ Lec_14.md

Lecture 14 - Pointers, Scopes, Arrays

Oct. 13/2020

Note-takers Note: Happy Thanksgiving! I hope you had a wonderful long weekend and took some time off to spend with family/friends!

Moving onto the lecture:

We've learned how to dynamically allocate variables and structs

- But what about dynamically allocating **objects**?

Dynamic Allocation of Objects

main.cpp

```
class DayOfYear{
private:
    int day;
    int month;
public:
    DayOfYear();
    DayOfYear(int d,int m);
    void setDay(int d);
    void setMonth(int m);
    void print();
}

int main(){
    DayOfYear* day1;
    day1 = new DayOfYear;
}
```

Notes:

1. DayOfYear* day1;
 - Create an **pointer** of type `DayOfYear`
2. `day1 = new DayOfYear;`
 - **Dynamically allocate** enough memory for a `DayOfYear` object
 - Store the pointer to this `DayOfYear` object in `day1`
 - This does not **instantiate** the object
 - No **constructor** is called

main.cpp

```
class DayOfYear{
private:
    int day;
    int month;
public:
    DayOfYear();
    DayOfYear(int d,int m);
    void setDay(int d);
    void setMonth(int m);
```

```

void print();
}

int main(){
    DayOfYear* day1;
    day1 = new DayOfYear;
    DayOfYear* day2;
    day2 = new DayOfYear(1,1);
}

```

Notes:

1. DayOfYear* day2;
 - o Create another **pointer** of type DayOfYear
2. day2 = new DayOfYear(1,1);
 - o Create a *new* object and store the address in day2
 - o Remember that this **instantiates** an object
 - **Constructor** is called
 - But which **constructor**?
 - In this case, DayOfYear(int d,int m) because of parameters given in object **instantiation** (DayOfYear(1,1);)

main.cpp

```

class DayOfYear{
private:
    int day;
    int month;
public:
    DayOfYear();
    DayOfYear(int d,int m);
    void setDay(int d);
    void setMonth(int m);
    void print();
}

int main(){
    DayOfYear* day1;
    day1 = new DayOfYear;
    DayOfYear* day2;
    day2 = new DayOfYear(1,1);

    delete day1;
    day1 = nullptr;
    delete day2;
    day2 = nullptr;
}

```

Notes:

1. delete day2;
 - o **delete** frees the memory of the **value** at day1
 - But remember that when **objects** are deleted, the **destructor** is called
 - Which **destructor**?
 - Trick question, only one **destructor**
2. day2 = nullptr;
 - o Remember to set the pointer day2 to a **nullptr** as to avoid memory leaks/dereferencing invalid memory

Variable Scopes

The **scope** of a variable is the part of the program in which the variable can be used

- Variables are usually **scoped** inside the **code blocks** they are in
- Global variables are defined in 'main.cpp' and are available **globally**, e.g. anywhere in the code

Local vs Global Variables

main.cpp

```
int g;

int main(){
    int i;
    float x;

    return 0;
}
void func1(int y){
    float x;
    int z;
}
```

Notes:

1. `int g;` is defined **globally**
 - `g` can be used *anywhere* in the code
2. `int i; float x;` are defined in the **scope** of `int main()`
 - They can only be used/referenced in the **scope** of `main`
 - Within the code blocks `{ }` of `main`
3. `float x; int z;` are defined in the **scope** of `void func1()`
 - They can only be used/referenced in the **scope** of `func1()`
 - Within the code blocks `{ }` of `func1`

In general, variables are scoped to the **code block** they are declared in.

Scope Hiding/Masking/Eclipsing

main.cpp

```
int g;

int main(){
    int i;
    float x;
    if(c){
        int i;
        i = 5;
    }
    return 0;
}
```

Notes:

1. Notice that `int i;` is called twice, one inside `main` and one inside `if(c)`
 - This is called **Scope Hiding or Scope Eclipsing**
 - Unclear which `int i` we should be using
 - Bad coding practice
2. `i = 5;`

- Like mentioned above, which variable *i* are we trying to modify?
 - Not exactly clear, so avoid redefining variables like this.

Scope of Dynamic Data

main.cpp

```
void allocate_int(){
    int* q;
    q = new int;
    *q = 5;
}

int main(){
    allocate_int();
    return 0;
}
```

Notes:

1. `int* q; q = new int; *q = 5;`
 - Create a **pointer** `q`, allocate enough memory for an `int`, and set the value at address `q` to 5
 - But this **dynamic data** is defined in the **scope** of `allocate_int()`
 - So we cannot use the value of `q` outside `allocate_int()`
2. What exactly is the scope of **dynamic data**?

main.cpp

```
int* allocate_int(){
    int* q;
    q = new int;
    *q = 5;
    return (q);
}

int main(){
    int* p = allocate_int();
    *p = 8;
    return 0;
}
```

Notes:

1. `int* q; q = new int; *q = 5;`
 - Same as previous example.
2. `return q;`
 - Return `q` from the function `allocate_int()`
 - But what *exactly* is `allocate_int()` returning?
 - `return q;` returns the value of `q` (ok, this was obvious)
 - It returns the **address** of `q`
 - `q` is a pointer type, so its value is an **address**
 - After `return` runs, `q` goes out of scope.
 - We cannot use the value of `q` anymore.
3. `int* p = allocate_int();`
 - `p`, an integer pointer, is assigned the value of `allocate_int()`
 - Remember that `return q;` in `allocate_int()` returns an **address**
 - Specifically, the **address** of `q` (before `q` went out of scope)

- Now, `p` has the same address `q` *had*
4. `*p = 8;`
- We can assign the value at the address of `p` (or equivalently, the value at the address of `q` before `q` went out of scope)
5. The scope of **dynamic data** is anywhere
- Between the bounds of creation and deletion of the **dynamic data**
 - As long as you know the address of the **dynamic data**
 - You can access it

main.cpp

```
int* do_something(){
    int x;
    x = 5;
    return &x;
}

int main(){
    int* a = do_something();
    *a = 8;
    return 0;
}
```

1. `int x; x = 5;`
 - Defines a local variable (**not dynamic**)
 - Sets the value to '5'
2. `return &x;`
 - Return the **address** of `x`
3. `int* a = do_something();`
 - Set the value of **pointer** `a` to the value of `x`
 - Except:
 - `x` went out of scope in `do_something()`
 - The memory of `x` has *been freed*
 - `x` is *not dynamic*
 - So the **address** of `x` does *not* refer to `x`
 - Could be some other variable you defined after
 - Could be a random location of memory now
 - This does not have to generate a **Segmentation Fault**
 - Could give you a compiler error (depends on version and compiler)
 - **Pointer** `a` is now set to some random memory address
 - Since `x` is not defined
 - 4. `*a = 8;`
 - Set the value at the address of `a` to '8'
 - What is this value?
 - Still points to **address** of `x`
 - But unsure what the value is, since `x` is no longer in scope

In this example, `a` is called a **Dangling Pointer**. The data that `a` points to is no longer valid.

Variable Types

1. Global
2. Local
3. Function Arguments

4. Dynamic

type	classification	memory location	description
Global	Automatic Variable	stack	Declared outside all functions. Visible everywhere in file.
Local	Automatic Variable	stack	Declared inside a function or code block . Visible only within that code block .
Function	Automatic Variable	stack	Declared within function headers . Visible only within function .
Dynamic	User-Managed	heap	Allocated by new . Exist from allocation to deletion . visible anywhere so long as there is a pointer to it.

All of the memory used by the program (instructions, code, variables) are defined inside the memory in 4 distinct areas:

Memory	
Code	Instructions
Data	Global/Static Variables
Stack	Automatic Variables
Heap	Dynamic Variables

When the program finishes, all the memory is **reclaimed** by the OS (Operating System).

- **Reclaiming is not deletion**
 - The memory is freed for another program to use.
- So why do we bother deleting/checking for memory leaks? - If OS will handle all that on program completion?
 - What if OS has memory leaks?
 - What if your program is a running **process**?
 - e.g. a Web Server
 - Just check for **dangling pointers** and **memory leaks**

■ Lec_15.md

Lecture 15 - Pointers, Scopes, Arrays

Oct. 14/2020

Note-takers Note: Good luck on the 212 quiz! If you're taking that at 6pm EST, I have no idea.

Arrays

What if we want a **data-type** that carries multiple values of a **primitive type**?

- e.g. List of student ID's
- e.g. Consecutive prime numbers or something

Quick review of **arrays** from APS105:

main.cpp

```
int main(){
    int a[4];
    a[0] = 4;
    cout << a[3];
}
```

Notes:

1. `int a[4];`
 - In order to define the array this way, the size must be *known at compile-time*
 - e.g. cannot be **dynamically allocated**
2. `a[0]=4;`
 - Assign "4" to the first element of the array
3. `cout << a[3];`
 - Print the 3rd element of the array to the screen

Arrays and Pointers

Arrays and pointers have a special relationship:

- Name of the array *is also* a pointer to the first element of the array
- The following rows are equivalent:

Array Indexing	Pointer Dereferencing
<code>a[0]</code>	<code>*a</code>
<code>a[1]</code>	<code>*(a+1)</code>
<code>a[i]</code>	<code>*(a+i)</code>
<code>i[a]</code>	<code>*(i+a)</code>

Every row in the table above contains equivalent code.

Pointer Arithmetic

Notice when we index `a[1]` using the pointer `a`, we write `*(a+1)`

- What does 1 represent?
 - Is this a single integer value?

The 1 represents **one address unit in memory**

- Adding 1 to a pointer results in the *next* memory block
 - $0x0000AB14 + 1 = 0x0000AB18$
 - Depends on the **size** of the variable represented by the address

Pointer Arithmetic is not great coding practice, as it is not always clear what the size of the value at `a` is.

- Thus not clear how many bytes between `*(a+1)` and `*(a)`

Dynamic Allocation of Arrays

Because arrays and pointers have this special relationship, we can **dynamically allocate arrays**

main.cpp

```
#include <iostream>

using namespace std;

int main(){
    int* myarray;
    int size;

    cin >> size;
    myarray = new int[size];
}
```

Notes:

1. `myarray = new int[size];`
 - Create a **integer array pointer**
 - Allocates "size" number of "integer memory blocks" for the array
 - In this case, an "integer memory block" is usually 4 bytes large
 - So `new int[size]` will allocate "4*size" bytes
 - This does **not** define each individual array element

main.cpp

```
#include <iostream>

using namespace std;

int main(){
    int* myarray;
    int size;

    cin >> size;
    myarray = new int[size];

    myarray[0]=0;
    for(int i = 1;i < size;++i){
        myarray[i] = 1;
}
```

```

    }
}

```

Notes:

1. `for(int i = 1; i < size; ++i){ myarray[i] = 1; }[i]`
 - Loop through elements from $i=1$ to $i=size-1$
 - Set elements in `myarray` to '1'

main.cpp

```
#include <iostream>

using namespace std;

int main(){
    int* myarray;
    int size;

    cin >> size;
    myarray = new int[size];

    myarray[0]=0;
    for(int i = 1; i < size; ++i){
        myarray[i] = 1;
    }

    delete[] myarray;
    myarray = null;
}
```

Notes:

1. `delete[] myarray;`
 - The `delete` keyword has *slightly different* behaviour when used with arrays
 - De-allocates *all* memory that was originally allocated for `myarray` with `new` keyword
 - Do not need to de-allocate all individual elements of array
2. `myarray = null;`
 - Good practice
 - Avoid dangling pointers

Arrays of Structs

Allocating **arrays of structs**

```
struct node{
    int ID;
    struct node* next;
}
struct node a[4];
```

Dynamically allocating arrays of structs

main.cpp

```
struct node{
    int ID;
    struct node* next;
```

```

}

int main(){
    cin >> size;
    struct node* a;
    a = new struct node[size];
}

```

Notes:

1. struct node* a;
 - o Defines a pointer to a **struct node**
2. a = new struct node[size];
 - o Creates a **struct node array pointer**
 - Allocates "size" amount of "struct node memory blocks" for the array
 - In this case, a "struct node memory block" is
 - The size of an integer and
 - The size of a **struct node pointer**

main.cpp

```

struct node{
    int ID;
    struct node* next;
}

int main(){
    cin >> size;
    struct node* a;
    a = new struct node[size];

    delete [] a;
    a = nullptr;
}

```

Notes:

1. delete [] a;
 - o De-allocates **all** memory that was originally allocated for a with new keyword
2. a = null;
 - o Good practice
 - o Avoid dangling pointers

main.cpp

```

struct node{
    int ID;
    struct node* next;
}

int main(){
    cin >> size;
    struct node* a;
    a = new struct node[size];

    int i = 0;
    a = new struct node;

    a[i].ID = 2;
    *(a+i).ID = 2;
}

```

```
(a+i)->ID = 2;

delete [] a;
a = nullptr;
}
```

Notes:

1. a = new struct node;
 - o Create a node variable at a[0]
 - Remember that a is a pointer to a[0]
 - a[0] is the first element in the array
2. a[i].ID = 2;
 - o Set the ID of the i th struct node element to 2
3. *(a+i).ID = 2;
 - o Set the ID of the i th struct node element to 2
4. (a+i)->ID = 2;
 - o Set the ID of the i th struct node element to 2
5. (2.,3.,4.) do the exact same thing.
 - o Difference is in readability :)

Arrays of Pointers

How to allocate an array a of 100 integer pointers

```
int* a[100];
```

Or, dynamically:

main.cpp

```
int main(){

    cin >> size;
    int** a;
    a = new int*[size];
}
```

Notes:

1. int** a;
 - o Define a **double pointer** of type int called a
2. a = new int*[size];
 - o Dynamically create an array of **integer pointers** of size "size"

main.cpp

```
int main(){

    cin >> size;
    int** a;
    a = new int*[size];
    for(int i = 0;i < size;++i){
        a[i] = new int;
    }
}
```

Notes:

1. `for(int i = 0;i < size;++i){ a[i] = new int; }`
 - Loop through elements from i=1 to i=size-1
 - Define each element of a as an **integer pointer**

We've figured out how to make each pointer in the array point to an int

- In other words, we've figured out how to **initialize** each of the pointer elements in the array a
- How do we delete this **dynamically allocated** data?

main.cpp

```
int main(){
    cin >> size;
    int** a;
    a = new int*[size];
    for(int i = 0;i < size;++i){
        a[i] = new int;
    }

    for(int i = 0;i < size;++i){
        delete a[i];
        a[i] = null;
    }
    delete [] a;
    a = nullptr;
}
```

Notes:

1. `for(int i = 0;i < size;++i){ }`
 - Loop through elements from i=1 to i=size-1
2. In For Loop: `delete a[i];`
 - De-allocate the pointer element at a[i]
3. In For Loop: `a[i] = null;`
 - Set the recently deleted pointers to null
4. What if we don't delete all the individual array elements (**pointers**)
 - We end up with **dangling pointers**
 - The pointers are still allocated in memory
5. `delete [] a;`
 - De-allocates **all** memory that was originally allocated for a with new keyword
6. `a = null;`
 - Set array pointer a to null;

How do we allocate an array called a of 100 pointers (to struct nodes)

```
struct node* a[100];
```

main.cpp

```
int main(){
    cin >> size;
    struct node** a;
    a = new struct node*[size];

    for(int i = 0;i < size;++i){
```

```
a[i] = new struct node;  
}  
  
for(int i = 0;i < size;++i){  
    delete a[i];  
    a[i] = null;  
}  
  
delete [] a;  
a = nullptr;  
}
```

Lec_16.md

Lecture 16 - Pointers, Scopes, Arrays and Overloading

Oct. 15/2020

Note-takers Note: Good luck on the 241 midterm!

Announcements:

Quiz 3 Runs on Monday (October 19)

- 30 minutes in duration
 - Any time during the day
- You **cannot** go back to quiz if you have left
- Please scroll down to make sure you don't miss questions!

Material Covered:

- Everything from **start of term** until **end of today's lecture**

Arrays of Objects

How to declare an array `a` of 10 `DayOfYear` objects?

- Easy, use `DayOfYear a[10];`
 - Don't even need **dynamic allocation**

But what constructor gets called?

- Actually, the **default constructor** is called
 - So if you haven't defined the **default constructor**, you **can not** define an array this way
- **Default constructor** is called `n` times
 - In the above case, `n = '10'`

What about with **dynamic allocation**?

- `DayOfYear* a = new DayOfYear[n];`
 - The **default constructor** is called, `n` times

What about deleting **dynamically allocated** arrays of objects?

- `delete [] a;`
 - The **default destructor** is called, `n` times

So how can we specify the **constructor** that should run when the **objects** in the **array** are instantiated?

main.cpp

```
int main(){
    DayOfYear* a[10];
    for(int i = 0;i < 10;++i){
        a[i] = new DayOfYear(1,2);
    }
}
```

```
//delete [] a;
}
```

Notes:

1. DayOfYear* a[10];
 - What constructor is called?
 - None, the above only creates an **array of pointers**
 - No objects have been **instantiated**
2. a[i] = new DayOfYear(1,2);
 - Uses the `DayOfYear(int,int)` constructor to **instantiate** the `DayOfYear` objects
3. //delete [] a;
 - **Cannot** call `delete` on a
 - a has **not** been dynamically allocated
 - It is a **stack** variable, not a **heap** variable

`main.cpp`

```
int main(){
    DayOfYear* a[10];
    for(int i = 0;i < 10;++i){
        a[i] = new DayOfYear(1,2);
    }
    for(int i = 0;i < 10;++i){
        delete a[i];
    }
}
```

Notes:

1. `delete a[i];`
 - Can call `delete` on `a[i]` to delete individual `DayOfYear` objects

Overloading

Start with an example: the **Time Class**

- Want to be able to add `time` objects
 - e.g. Add one time object representing noon, and one time object representing 4pm

We can **overload** default operators to define custom behaviour for them.

- e.g. we can define what the `+` operator does for *objects*

What exactly is `x+y` ?

- What does the `+` represent?
 - It is **Syntactic Sugar** in C++
 - **Exactly the same** as `x.operator+(y)`
 - Where `x` and `y` are compatible **objects**
 - the `operator+` function is a **class member** of `x`

The `.operator` operator

- Is **always** called on the object *on the left*
 - `x+y` or `x.operator+(y)` is called **on x**

time.h

```

class Time{
private:
    int hour, minute, second;
public:
    Time();
    Time(int h, int m, int s);
    ~Time();

    int getHour();
    int getMinute();
    int getSecond();
    void setHour(int h);
    void setMinute(int m);
    void setSecond(int s);

    Time operator+ (Time rhs);
    Time operator- (Time rhs);
    void print();
};


```

Notes:

1. Time operator+ (Time rhs);
 - This **overloads** the + operator with a specific behaviour
2. Time operator- (Time rhs);
 - This **overloads** the - operator with a specific behaviour

What exactly is that behaviour??

- We define it in the **function definition**

time.cpp

```

Time Time::operator+ (Time rhs){
    int TotalSeconds;
    Time sum;

    TotalSeconds = second + 60*minute + 3600*hour;
    TotalSeconds += rhs.second + 60*rhs.minute + 3600*rhs.hour;

    sum.hour = TotalSeconds/3600;
    sum.minute = (TotalSeconds - 3600*sum.hour)/60;
    sum.second = TotalSeconds%60;

    return (sum);
}

```

Notes:

1. TotalSeconds = second + 60*minute + 3600*hour;
 - Set TotalSeconds to be the *value* of the current object
 - In this case, the *current object* is the one the operator is being *called on*
 - Or, the **left hand side**
2. TotalSeconds += rhs.second + 60*rhs.minute + 3600*rhs.hour;
 - Adds **right hand side** object values to TotalSeconds
 - TotalSeconds currently represents **left hand side**
3. sum.hour = TotalSeconds/3600;
 - Remember that the **operator+** operator is defined in the **scope** of the class

- Can access **data members** of the sum object without using **accessor** functions

4. `return sum;`
 - Return the sum object (of type `Time`)
 - So `Time z = x+y;` is valid
 - As long as `x` and `y` are `Time` objects
 - **Must** use this `return`
 - The **assignment operator** **must be evaluable**

What does it mean to be **evaluable**?

- First of, I just define the term **evaluable** to mean "able to be evaluated"
- Essentially, `(x+y)` must **evaluate** to something.
 - In other words, `z = (x+y)` must be defined
 - This means that the `operator+` **must have a return type**

`time.cpp`

```
Time Time::operator- (Time rhs){
    int TotalSeconds;
    Time sum;

    TotalSeconds = second + 60*minute + 3600*hour;
    TotalSeconds -= rhs.second +60*rhs.minute + 3600*rhs.hour;

    sum.hour = TotalSeconds/3600;
    sum.minute = (TotalSeconds - 3600*sum.hour)/60;
    sum.second = TotalSeconds%60;

    return (sum);
}
```

Notes:

1. `TotalSeconds -= rhs.second +60*rhs.minute + 3600*rhs.hour;`
 - Does the same thing as `operator+` but subtracts instead.

Default Operators

One specific operator is given by default

- The `operator=` operator, or the **assignment operator**

`time.cpp`

```
Time Time::operator= (Time rhs){
    hour = rhs.hour;
    minute = rhs.minute;
    second = rhs.second;
    return (*this);
}
```

Notes:

1. `operator=`
 - Sets the object on the left to the object on the right
 - Think about `int x = 5;`
 - Sets variable `x` equal to 5
2. `return (*this);`

- What is `(*this)` ??
 - Evidently, it's an **object** of type `Time`
 - Just look at function return type
- Why do we return `(*this)`?
 - Because `z=x=y` is valid
 - The expression `x=y` assigns `x` the value of `y`
 - But also has a evaluable value
 - `(x=y)` will evaluate to `x`
 - `z=(x=y)` is identical to `z=x`

Overloading Restrictions

We can now overload **most** operators. Which ones can we not overload?

- `.` : field access operator
- `::` : scope resolution operator
- `? :` ternary/conditional operator
- `sizeof` : object size operator

And many more. Check out the TB or online C++ reference for the full list. :)

Object Copying

What if we wanted an object to be created that is a *copy* of another object?

- C++ invokes the **Copy Constructor** of the class
- A new object is initialized by the **Copy Constructor**

But remember when you pass variables into functions via parameters, they are **passed by value**

- What is passing by value for **objects**?
 - Actually, it is **Object Copying**
 - Invocations of the **copy constructor** as new objects are created

Additional to the `operator=` assignment operator, the copy constructor is *also given to you by default by c++*

Examples of Object Copying:

```
Time X(Y);
Time *p = new Time(X);

Time X = Y;
```

Notes:

1. `Time X = Y;`
 - This calls the **copy constructor**
 - Creating a *new* object `X`

Wait, how is this different from `operator=` (the assignment operator)

```
Time X(1,1,1);
Time Y(0,0,0);
X = Y;
```

The above invokes the **assignment operator**

```
Time Y(0,0,0);
Time X = Y;
```

The above invokes the **copy constructor**

time.cpp

```
Time::Time(Time & source){
    hour = source.hour;
    minute = source.minute;
    second = source.second;
}
```

Notes:

1. This is the **copy constructor**
2. Time::Time(Time & source){
 - The source object **must** be passed by reference.
 - This is a **requirement** for the **copy constructor**

Lec_17.md

Operator Overloading

Oct. 20/2020

Note-takers Note: The fire alarm in my building is being tested today, please excuse notes if I miss anything. Sorry!

Object Assignment

main.cpp

```
int main(){
    Time x(5,2,30);
    Time y(10,30,48);

    return 0;
}
```

By default, if you try to assign `x=y`

- C++ provides a **default operator** to do this assignment
 - The `operator=` assignment operator

The Operator= Operator

```
Time Time::operator= (Time rhs){
    hour = rhs.hour;
    minute = rhs.minute;
    second = rhs.second;

    return (*this);
}
```

Notes:

1. `return (*this);`
 - We need to comply with **backwards-compatibility**
 - Specifically, with the rule *Assignments are also expressions*
 - `z = (x = y)` **must** be defined

The Copy Constructor

Notice in the function declaration for the `operator=` function:

- `Time Time::operator= (Time rhs)`
 - We are passing in a `Time rhs` object
 - Except this is a *pass by value*, not a *pass by reference*
 - So the object is *copied*, not *given*

Whenever an object is to be created that is a copy of another object, C++ invokes the **copy constructor** of the class

- Object initialized by the **copy constructor**

Examples:

1. Time X(Y);
 - o Y is a Time object
2. Time *p = new Time(X);
 - o X is a Time object
 - p points to a new object which is a *copy* of X
3. void do_something(Time x);
 - o Passing objects into functions *by value*
4. Time X = Y;
 - o This is actually the **copy constructor**
 - We are creating x, and would like to initialize it with Y
 - o This does *not* invoke the **assignment operator**
 - If x was initialized, then the **assignment operator** would be called instead of the **copy constructor**

Writing out the **copy constructor**:

```
Time::Time(Time & source){
    hour = source.hour;
    minute = source.minute;
    second = source.second;
}
```

Notes:

1. Time::Time(Time & source)
 - o Notice we are passing source *by reference*
 - Do we have to *pass by reference*? Can we instead *pass by value*?
 - Yes, you **must** pass by reference in this case
 - Otherwise, it will enter a **recursive loop**
 - Trying to create a new object Time source will result in a **recursive call** to the **copy constructor**
 - Which then attempts to make a new object with **copy constructor** which calls the function again
 - **Compile-Time Error** if you *try to pass by value*
 - 2. There is no return
 - o It is a constructor, after all

The **copy constructor** is invoked whenever a **new** object is created

- The new object is a copy of an *existing object* of the same type
- If the object exists, use the **overloaded assignment operator**

Pass by Reference

Pass by reference is necessary for the **copy constructor**

- We can also use it to *avoid* copying objects

```
Time Time::operator= (Time rhs){
    hour = rhs.hour;
    minute = rhs.minute;
    second = rhs.second;

    return (*this);
}
```

Notes:

1. Time Time::operator= (Time rhs)
 - o Time rhs is passed by value
 - This is a copy of an object
 - Invokes the **copy constructor**
 - At the end of operator=, rhs goes out of scope
 - The **default destructor** is called too
 - All of this is costly!

```
Time Time::operator= (Time & rhs){
    hour = rhs.hour;
    minute = rhs.minute;
    second = rhs.second;

    return (*this);
}
```

Notes:

1. Time Time::operator= (Time & rhs)
 - o Time rhs is passed by reference
 - This is **not** a copy of an object, it **is** the object

```
Time & Time::operator= (Time & rhs){
    hour = rhs.hour;
    minute = rhs.minute;
    second = rhs.second;

    return (*this);
}
```

Notes:

1. Time & Time::operator= (Time & rhs)
 - o Return value is returned by reference
 - No copying takes place on return

Pass by reference avoids cost of copying objects, but now any function with **pass by reference** can accidentally change the value of source objects

```
Time Time::(Time & source){
    hour = source.hour;
    minute = source.minute;
    second = source.second;
    source.hour = 0;
}

Time & Time::operator= (Time & rhs){
    hour = rhs.hour;
    minute = rhs.minute;
    second = rhs.second;
    rhs.hour = 0;

    return (*this);
}
```

Notes:

1. `source.hour = 0;`
 - This changes the value of `source.hour`
 - Outside the scope of the function
2. `rhs.hour = 0;`
 - Similarly, this changes the value of the `.hour` field on the right hand side of the `operator=` call

So how can we restore some protection?

- To avoid incorrectly rewriting/modifying member data fields for objects *passed by reference*

The Const Modifier

We can define the **pass by reference** parameter as a **constant**

- Locally, this means the value of the **pass by reference** parameter *cannot be changed*

```
Time Time::(const Time & source){
    hour = source.hour;
    minute = source.minute;
    second = source.second;
    source.hour = 0;
}

Time & Time::operator= (const Time & rhs){
    hour = rhs.hour;
    minute = rhs.minute;
    second = rhs.second;
    rhs.hour = 0;

    return (*this);
}
```

Notes:

1. `Time Time::(const Time & source)`
 - Define the **pass by reference** as a **constant**
 - Cannot change the member data fields of `source`
2. `Time & Time::operator= (const Time & rhs)`
 - Same as 1., for `rhs`
3. `source.hour = 0; , rhs.hour = 0;`
 - These are now **compile-time errors**

Applying this to other examples:

time.cpp

```
Time Time::operator+ (const Time & rhs){
    int TotalSeconds;
    Time sum;

    TotalSeconds = second + 60*minute + 3600*hour;
    TotalSeconds += rhs.second + 60*rhs.minute + 3600*rhs.hour;

    sum.hour = TotalSeconds/3600;
    sum.minute = (TotalSeconds - 3600*sum.hour)/60;
    sum.second = TotalSeconds%60;

    rhs.second = 0;
    second = 0;
```

```

    return (sum);
}

```

Notes:

1. Time Time::operator+ (const Time & rhs)
 - o Define the **pass by reference** as a **constant**
2. rhs.second = 0;
 - o This is a **compile-time error**
3. second = 0;
 - o This is still allowed, but...

When we say `z = x+y` (`x, y, z` all `Time` objects)

- Do we really want the value of `x` to be changed accidentally?
 - o No, we don't.
 - o So we want to provide modify protection to `x`
 - `x` is the object that `operator+` is called on
 - Similar to how we provided `const` protection to `rhs` earlier

`time.cpp`

```

Time & Time::operator+ (const Time & rhs) const{
    int TotalSeconds;
    Time sum;

    TotalSeconds = second + 60*minute + 3600*hour;
    TotalSeconds += rhs.second +60*rhs.minute + 3600*rhs.hour;

    sum.hour = TotalSeconds/3600;
    sum.minute = (TotalSeconds - 3600*sum.hour)/60;
    sum.second = TotalSeconds%60;

    rhs.second = 0;
    second = 0;

    return (sum);
}

```

Notes:

1. Time & Time::operator+ (const Time & rhs) const { }
 - o Define the function that is being called on as a **const**
 - We cannot edit the values of `x` (or the left hand side of `x+y`)
2. second = 0;
 - o This is *now also* a **compile-time error**

■ Lec_18.md

Lecture 18 - Operator Overloading

Oct. 21/2020

Const Modifier

We want to provide protections to X and Y when calling `x+y`

`time.cpp`

```
Time & Time::operator+ (const Time & rhs) const{
    int TotalSeconds;
    Time sum;

    TotalSeconds = second + 60*minute + 3600*hour;
    TotalSeconds += rhs.second +60*rhs.minute + 3600*rhs.hour;

    sum.hour = TotalSeconds/3600;
    sum.minute = (TotalSeconds - 3600*sum.hour)/60;
    sum.second = TotalSeconds%60;

    rhs.second = 0;
    second = 0;

    return (sum);
}
```

Notes:

1. `Time & Time::operator+ (const Time & rhs) const`
 - Return by reference
 - However, `sum` is a local object -> goes out of scope

So when we pass the `sum` value out *by value*, the value of `sum` inside `operator+` goes out of scope

- But what defines what memory gets kept in/out of scope?
 - The compiler
- In this case, the value of `sum` is copied outside the scope of `operator+`
 - A **copy constructor** is called and a **destructor** is called
 - Certain compilers will extend/prolong the life of the `sum` object to avoid repeated copy calls
 - This is called an **optimization**, don't count on it

Complex Numbers Class

We want to design and write a C++ class that allows us to create and manipulate complex numbers

- Consists of *real part* and *imaginary part*
- Example usage:

```
complex a(5.0,8.3);
complex b(a);
complex c(3.9,17.8);
```

```

complex d;

d = a + c;
c = b - a;
a = c / d;
b = c * a;
if (b == d) c.print();

```

Notes:

1. Want to be able to add, subtract, divide, multiply
 - Overload basic operators
 - operator+, operator-, operator/, operator*
2. Need two constructors
 - Default constructor
 - Want to be able to call complex d;
 - Constructor that takes two floats
 - Want to call complex c(3.9,17.8);
3. Want to overload == equality comparison operator
4. Want **accessors/mutators** to get fields of complex values
5. Want .print() to print value of complex class

Additionally, if we want to use += or -= operators on the Complex Number class

- We would have to individually **overload** these operators as well!

Class Definition

Implementing the requirements:

complex.h

```

class complex{
private:
    float real;
    float imag;
public:
    complex();
    complex(float r, float i);

    float getReal() const;
    float getImag() const;
    void setReal(float r);
    void setImag(float i);

    complex operator+ (const complex & rhs) const;
    complex operator- (const complex & rhs) const;
    complex operator* (const complex & rhs) const;
    complex operator/ (const complex & rhs) const;
    bool operator== (const complex & rhs) const;

    void print() const;
};

```

Notes:

1. Do we need to write the **copy constructor**?
 - No, C++ can provide this by default
2. Do we need to write the **default destructor**?

- No, C++ can provide this by default
3. Do we need to overload the **assignment operator**?
- No, C++ can provide this by default

Constructors/Destructors

```
complex::complex(){
    real = 0.0;
    imag = 0.0;
}

complex::complex(float r, float i){
    real = r;
    imag = i;
}
```

Accessors/Mutators/Printing

```
float complex::getReal() const {
    return (real);
}
float complex::getImag() const {
    return (imag);
}
void complex::setReal(float r) {
    real=r;
}
void complex::setImag(float i) {
    imag=i;
}
void complex::print() const {
    cout << "(" << real << "," << imag << ")";
}
```

Notes:

- The `const` term is used in certain function definitions to avoid overwriting variables

Operator+

```
complex complex::operator+ (const complex & rhs) const {
    complex tmp;
    tmp.real = real + rhs.real;
    tmp.imag = imag + rhs.imag;
    return tmp;
}
```

Notes:

1. Argument passed *by reference*
2. Return passed *by value*
 - We want `tmp` to last beyond scope of the function
3. This does **not** define behaviour for `x+(-Y)`

Operator-

```
complex complex::operator- (const complex & rhs) const {
    complex tmp;
    tmp.real = real - rhs.real;
    tmp.imag = imag - rhs.imag;
    return tmp;
}
```

Operator*

```
complex complex::operator* (const complex & rhs) const {
    complex tmp;
    tmp.real=(real*rhs.real)-(imag*rhs.imag);
    tmp.imag=(real*rhs.imag)+(imag*rhs.real);
    return tmp;
}
```

Operator/

```
complex complex::operator/ (const complex & rhs) const {
    complex tmp;
    float mag;
    mag = (rhs.real*rhs.real)+(rhs.imag*rhs.imag);
    tmp.real = (real*rhs.real)+(imag*rhs.imag);
    tmp.real = tmp.real/mag;
    tmp.imag = (imag*rhs.real)-(real*rhs.imag);
    tmp.imag = tmp.imag/mag;
    return tmp;
}
```

Operator==

Another operator we want to define is the **equality** operator.

- Returns a `bool` (primitive) type

```
bool complex::operator== (const complex & rhs) const {
    if ( (real==rhs.real) && (imag==rhs.imag) ) return (true);
    else return (false);
}
```

Notes:

1. `bool complex::operator== (const complex & rhs) const{ }`
 - This **overloaded operator** returns a `bool`
 - Not every overloaded operator needs to return an object
2. Is `real==rhs.real` and `imag==rhs.imag` recursion?
 - Are we calling `operator==` again?
 - No, the types of `real`, `rhs.real`, `imag`, `rhs.imag` are floats

Exercises

Exercise 1

```
{
    complex a(5.0, 8.3);
```

```

complex b(a);
complex c(3.9, 17.8);
complex d;
complex* p = new complex();
p = new complex(1.8,2.9);
if (b == d) p->print();
delete p;
}

```

Q:

1. Which constructors are called?
 - o **default constructor**, constructor w/ two floats, **copy constructor**
2. How many times is the destructor called?
 - o 5
 - a, b, c, d, p
3. How many times is an object copied?
 - o 1
 - p->print()

Exercise 2

```

{
    complex a(5.0, 8.3);
    complex b(a);
    complex c(3.9, 17.8);
    complex d;
    complex* p = new complex();
    p = new complex(1.8,2.9);
    c = b - a;
    a = c + b;
    *p = c / (*p);
    if (b == d) p->print();
    delete p;
}

```

Q:

1. Which constructors are called?
 - o First write operators out in their true form
 - c = b - a; -> c.operator=(b.operator-(a));
 - a = c + b; -> a.operator=(c.operator+(b));
 - o Figure out which constructors are called by the **overloaded operators**
2. How many times is the destructor called?
 - o 11
 - This might be wrong, my personal counting
 - a, b, c, d, p
 - Inside scope of:
 - 3x operator=
 - operator-
 - operator+
 - operator/
3. How many copies of object a are made?
 - o 1
 - complex b(a);
4. How many times is an object copied?

■ Lec_19.md

Lecture 19 - Operator Overloading

Oct. 22/2020

Complex Numbers Class

Using the Complex Class as an example

`complex.h`

```
class complex{
private:
    float real;
    float imag;
public:
    complex();
    complex(float r, float i);
    complex(const complex & src);
    ~complex();

    float getReal() const;
    float getImag() const;
    void setReal(float r);
    void setImag(float i);

    complex operator+ (const complex & rhs) const;
    complex operator- (const complex & rhs) const;
    complex operator* (const complex & rhs) const;
    complex operator/ (const complex & rhs) const;
    bool operator== (const complex & rhs) const;

    void print() const;
};
```

Notes:

1. `complex(const complex & src);`
 - This is the **copy constructor**
 - Want to define the implementation for the **copy constructor**
 - Instead of using the default implementation
2. `~complex();`
 - Want to define implementation for the **destructor** as well

You can use a `waitForKey()` function to temporarily block program execution

`waitForKey`

```
void waitForKey(){
    string anyKey;
    cout << endl << "main: press any key to continue...";
    cin >> anyKey;
    cout << endl;
}
```

Extending the Complex Class

We want to be able to do this:

```
using namespace std;
#include <iostream>
int main() {
    complex c;
    // Would like to do this
    cout << c;
    // instead of this
    c.print();
    :
}
```

Overloaded Operators and Friends

Alternative way to **overload** operators is to use non-member functions

- As opposed to member functions
- **Overloaded operators** are outside the class

```
class Foo {
private:
    int x;
public:
    Foo(int);
    int getX() const;
    void setX(int i);
};

int main () {
    Foo a(0), b(1), c(2);
    c = a + b;
    :
}

Foo operator+(const Foo& lhs, const Foo& rhs) {
    Foo t(lhs.x + rhs.x);
    return(t);
}
```

Notes:

1. `Foo operator+(const Foo& lhs, const Foo& rhs);`
 - Notice this `operator+` takes *two* parameters
 - left hand side and right hand side
2. `Foo t(lhs.x + rhs.x);`
 - Notice that `.x` are member data fields of `lhs` and `rhs`
 - The `operator+` function cannot access these fields
 - Recall **Access Control!**

One way to fix this issue is to use accessor methods:

```
class Foo {
private:
    int x;
public:
    Foo(int);
    int getX() const;
    void setX(int i);
```

```

};

int main () {
    Foo a(0), b(1), c(2);
    c = a + b;
    :
}

Foo operator+(const Foo& lhs, const Foo& rhs) {
    Foo t(lhs.get_x() + rhs.get_x());
    return(t);
}

```

Notes:

1. Foo t(lhs.get_x() + rhs.get_x());
o Assuming `get_x()` is an **accessor** method that returns the value of `x`

Another more elegant way to approach this

- Break access control rule
- Friends

Friends

Friends are **non-member functions** of a class

- Able to access **private members** of the class
- Breaking a rule
 - o **Encapsulation** (Access Control)
 - o Private members should be private
 - However, they can be accessed by friends

```

class Foo {
private:
    int x;
public:
    Foo(int);
    int getX() const;
    void setX(int i);
    friend Foo operator+(const Foo& lhs, const Foo& rhs);
};

int main () {
    Foo a(0), b(1), c(2);
    c = a + b;
    :
}

Foo operator+(const Foo& lhs, const Foo& rhs) {
    Foo t(lhs.x + rhs.x);
    return(t);
}

```

Notes:

1. `friend Foo operator+(const Foo& lhs, const Foo& rhs);`
o You can make the `operator+` function a friend of the `foo` class
 - `foo` has to declare this friendship
2. `Foo t(lhs.x + rhs.x);`
o Now the `operator+` function can use the `lhs.x` and `rhs.x` data fields

Overloading the `<<` Operator

We want to be able to write:

```
complex c;
cout << c;
```

First, understanding what the `<<` operator (the **insertion operator**) does:

- The function definition for the `<<` operator:
 - `ostream& operator<<(ostream& os, int i);`
 - Must pass `ostream& os` by reference
 - Too big to copy
 - C++ wants to throw a *run-time error* instead of a *compile-time error* for passing `ostream` by value
 - Make the `ostream` **copy constructor** private.

```
int main(){
    int x;
    cout << x;
}
```

Translates into:

```
int main(){
    int x;
    operator<<(cout, x);
}
```

So the function `ostream& operator<<(ostream& os, int i);` must exist

Same thing for `floats`:

```
int main(){
    float x;
    cout << x;
}
```

Translates into:

```
int main(){
    float x;
    operator<<(cout, x);
}
```

So the function `ostream& operator<<(ostream& os, float i);` must exist

Notice that the `operator<<` function is already overloaded *for primitive data types*

- `int`, `float`, `char`
 - Can print all of these to screen Extending this for the `complex` class we have written
- We need to define:
 - `ostream& operator<<(ostream& os, const complex & x);`

complex.cpp

```
ostream& operator<<(ostream& os, const complex & x){
    os << "(" << x.real << "," << x.imag << ")";
```

```
    return os;  
}
```

Note:

1. `os << "(" << x.real << "," << x.imag << ")";`
 - Uses the **insertion operator** to add to the `os` `ostream`
 - Print to the terminal
2. Notice that `x.real` and `x.imag` are private member data fields for the `complex` class

Must declare the `operator<<` function a friend inside the `complex` class

- Why not define this function as a member function of `complex`?

■ Lec_20.md

Lecture 20 - When Objects have Pointers

Oct. 27/2020

Pointers are Wiggly - Prof Abdelrahman 2020

Shallow vs Deep

- Shallow deals with data that is stored inside an object
 - This is defined for class members which are non-dynamic
 - Remember that **dynamic** data exists outside of classes
- Deep deals with data stored outside object
 - Dynamic data

Classes with Pointers

Imagine a C++ class that contains data members which are **pointers**

- What happens?
 - Problem arise with C++'s **shallow** operations
- When pointers exist, the programmer must provide **deep** versions of these operators
 - Otherwise only the memory area for those pointers exists

Shallow Operations

The following operations are **shallow** by default:

1. Object Creation
 - Creates area to hold class members
 - No initialization for these class members
2. Object Destruction
 - Erases area that holds class members
3. Object Copying
 - Member by Member copying
4. Object Assignment
 - Member by Member assignment

This works well for **non-pointer** class members

- Like in the Time class we wrote previously
- But *not* for pointer class members

Implementations for deep operations for class Time:

Time.h

```
struct _time { int hour, minute, second; };

class Time {
```

```

private:
    struct _time* time_ptr;
public:
    Time ();
    Time (int h, int m, int s);
    Time (const Time & source);
    ~Time();
    int getHour();
    int getMinute();
    int getSecond();
    void setHour(int h);
    void setMinute(int m);
    void setSecond(int s);
    Time operator+ (Time rhs);
    Time operator- (Time rhs);
    void print ();
};


```

Deep Constructors

The C++ **default constructor** initializes *only* the dynamically allocated pointers

```

Time::Time() {
    time_ptr = new struct _time;
}

```

We want to write **deep constructors**

- Assign data to dynamically allocated class data members

```

Time::Time() {
    time_ptr = new struct _time;
    time_ptr->hour=0;
    time_ptr->minute=0;
    time_ptr->second=0;
}

```

Notes:

1. `time_ptr` is now **deep data**
- The value at the address pointed by `time_ptr` exists *outside* the class

```

Time::Time(int h, int m, int s) {
    time_ptr = new struct _time;
    time_ptr->hour = h;
    time_ptr->minute = m;
    time_ptr->second = s;
}

```

Notes:

1. `time_ptr` is now **deep data**
- The value at the address pointed by `time_ptr` exists *outside* the class

Deep Destructors

The C++ **default destructor** does not provide any implementation

- Invoked just before object is deleted
- **shallow delete**
 - Leaves a **dangling pointer** (memory leak)

```
Time::~Time() {
}
```

We want to avoid **memory leaks**

- Delete dynamically allocated, **deep** data

```
Time::~Time() {
    delete time_ptr;
}
```

Deep Assignment

For an assignment `x=y;`

The C++ **default assignment operator** `operator=` directly *updates* the value at the address of the pointer

```
Time & Time::operator= (const Time & t) {
    time_ptr = t.time_ptr;
    return (*this);
}
```

The default version of the `operator=` operator has two large problems

1. Results in a **memory leak** * `x.time_ptr` now points to value at address of `y.time_ptr` * No way to access the initial value of **dynamically allocated** `x.time_ptr`
2. Results in a **shared object data** * Any updates to `x.time_ptr` directly affect the value of `y.time_ptr` * We want both objects (`x, y`) to have separate data

We want a **deep assignment** that addresses both of these problems

```
Time & Time::operator= (const Time & t) {
    time_ptr->hour = t.time_ptr->hour;
    time_ptr->minute = t.time_ptr->minute;
    time_ptr->second = t.time_ptr->second;
    return (*this);
}
```

Deep Copying

The C++ **default copy** does member copying

- **Shallow copying**

```
Time::Time(const Time & src) {
    time_ptr = src.time_ptr;
}
```

Results in the same issue with the **assignment operator**

- **Shared object data**

We want a **deep copy** that addresses the issue of **shared object data**

```
Time::Time(const Time & src) {
    time_ptr = new struct _time;
    time_ptr->hour = src.time_ptr->hour;
    time_ptr->minute = src.time_ptr->minute;
    time_ptr->second = src.time_ptr->second;
}
```

Notes:

1. Time::Time(const Time & src)
 - Must pass *src* *by reference*
 - Avoid cost of copying and recursive loop regarding copying

Const Revisited

pass by reference avoids the cost of copying objects

- removes protection of caller from callee when reassigning data values
- `const` modifier restores some of that protection

```
Time & Time::operator= (const Time & t) {
    t.time_ptr = NULL; // Compile-time error
    t.time_ptr->hour=0; // Not a compile-time error
    return (*this);
}
```

Notes:

1. `t.time_ptr = NULL;`
 - Compile-time error
2. `t.time_ptr->hour=0;`
 - Not a compile-time error

Lec_21.md

Lecture 21 - An Abstract String Class

Oct. 28/2020

Previously, we've dealt with objects with **pointer** data members

- Let's implement a custom/abstract `String` class as an example
 - `String`, uppercase S
 - Different from built in `string`

String Class

Want to be able to do:

Create Strings

```
String FirstName("Tarek");
String LastName = "Abdelrahman";
String name;
```

Notes:

1. `String FirstName("Tarek")`
- "Tarek" is a c-string
 - C style, **null-terminated** string

Access Strings

```
FirstName.length();
char c = LastName[0];
FirstName[3] = 'i';
```

Notes:

1. `FirstName.length();`
- Get length of string
2. `FirstName[3] = 'i';`
- Modify and access strings as if they were arrays

Operate on Strings

```
if(FirstName=="Tarek"){ ... }
if(LastName==FirstName){ ... }
if(FirstName < LastName){ ... }
FirstName = name;
```

Notes:

1. if(FirstName<LastName);

- Compare strings

Print Strings

```
cout << FirstName;
```

String Object

Need to have:

Data Members

1. char* str

- Character array just large enough to hold characters of string

- Null-terminated

- Contains a '\0' character to indicate end of string

- char* str is **shallow** data

- The value pointed to by char str* should be **deep** data

2. int len;

- Contains length of string

Initial Class Definition

```
class String{
private:
    char* str;
    int len;
public:
    ...
};
```

Constructors

1. String();

- Default Constructor

2. String(const char *s);

- C-string Constructor

3. String(const String & s);

- Copy constructor

Constructors must dynamically allocate data to hold characters of string

- Initialize dynamically allocated data (**deep** data)

Default Constructor

```
String::String(){
    len=0;
```

```

str = new char[1];
str[0] = '\0';
}

```

C-String Constructor

```

String::String(const char* s){
    len = strlen(s);
    str = new char[len+1];
    strcpy(str,s);
}

```

Copy Constructor

```

String::String(const String & s){
    len = s.len;
    str = new char[len+1];
    strcpy(str,s.str);
}

```

Default Destructor

```

String::~String(){
    delete [] str;
}

```

Accessors

What if we want to find

- Length of the string
- Character at specific index

length()

```

int String::length() const{
    return len;
}

```

To get the character of a `char*` array at a specific index, we can call

- `arrayName[i]` for `i` in bounds
 - But what can we do for objects?
 - Overload the `operator[]` operator!

Operators

operator[]

```

char & String::operator[](int i){
    if((i<0)|| (i>len-1)){
        cerr << "Error: out of bounds";
        exit(0);
    }
    return str[i];
}

```

Notes:

1. `char & String::operator[](int i){ }`

 - Notice the *return by reference*
 - The character itself is returned by `operator[]`
 - Not just a copy of the character
 - The usage of this operator is `ObjectName[i]`
 - The array index `i` is the parameter `i` in `operator[](int i)`

2. `if((i<0)|| (i>len-1))`

 - Check if `int i` param is in bounds

3. `cerr`

 - Similar to `cout`, but used for errors
 - Not a great implementation
 - Would rather *throw an exception* (and let that be handled)

operator=

```
String & String::operator=(const String & rhs) {
    if (this == &rhs) return (*this);
    delete [] str;
    len = rhs.len;
    str = new char[len + 1];
    strcpy(str, rhs.str);
    return (*this);
}
```

Notes:

1. `if (this == &rhs) return (*this);`

 - Imagine if we called `fname=fname`
 - `delete [] str;` would *delete* the value that we are trying to copy a few lines later.
 - Need some guard in case we try to assign an object to itself.
 - The `&` is an **overloaded symbol**
 - In the function definition `const String & rhs`, `&` defines the pass type for parameters
 - In `this == &rhs`, `&` is the reference operator
 - Standalone, `&` is the bitwise AND operator
 - Paired up, `&&` is the logical AND operator

2. `delete [] str;`

 - Delete the original string stored by `this`

3. `len = rhs.len;`

 - First copy the length of `rhs` to `this`

4. `return (*this);`

 - Return the lhs of the `=` call, or the `this`

Comparison Operators

Define:

1. operator<
2. operator>
3. operator==

operator< and operator>

```
bool String::operator<(const String & rhs) const {
    return (strcmp(str, rhs.str) < 0);
}
```

Notes:

1. `strcmp(str,rhs.str)<0`
- Using the built in C library functions
- `strcmp` compares objects **lexicographically**
 - Based on order in the alphabet
2. Pretty much the same definition for `operator<`

operator==

There are multiple cases for the definition of `operator==`. Compare:

1. `fname == "Stewart";`
2. `"Tarek" == FirstName;`
3. `fname == FirstName;`

So need to define multiple `operator==` calls

Comparing String and String

Comparing two objects of class `String`:

```
bool String::operator==(const String & rhs) const {
    return (strcmp(str, rhs.str) == 0);
}
```

Comparing String and const char*

```
bool String::operator==(const char* s) const {
    return (strcmp(str, s) == 0);
}
```

■■ Lec_22.md

Lecture 22 - An Abstract String Class and Linked Lists

Oct. 29/2020

String Class

Continuing the string class implementation from last lecture...

Comparison Operators

`operator==`

There are multiple cases for the definition of `operator==`. Compare:

1. `fname == "Stewart";`
2. `"Tarek" == FirstName;`
3. `fname == FirstName;`

So need to define multiple `operator==` calls

Comparing String and String

Comparing two objects of class `String`:

```
bool String::operator==(const String & rhs) const {
    return (strcmp(str, rhs.str) == 0);
}
```

Comparing String and `const char*`

```
bool String::operator==(const char* s) const {
    return (strcmp(str, s) == 0);
}
```

Comparing `const char*` and String

What if we have a string on the LHS?

- e.g. `"Stewart" == fname`
 - Build an `operator==` **overload** that is a **non-member function**

```
bool operator==(const char* lhs,const String & rhs) const {
    return (strcmp(lhs, rhs.str) == 0);
}
```

Note:

1. `bool operator==(const char* lhs,const String & rhs) const { }`
- `operator==` is a **non-member function**

2. `return (strcmp(lhs, rhs.str) == 0);`
- `operator==` is trying to access `rhs.str`
 - But this overload is a **non-member function**
 - So cannot access by default
 - Need to make `operator==` a **friend** function of class `String`

Printing String Objects

```
ostream & operator<<(ostream & os, const String & s)
{
    os << "(" << s.str << ")";
    return (os);
}
```

Notes:

1. `ostream & operator<<(ostream & os, const String & s){ }`
- `operator<<` is a **non-member function**
 - Need to make `operator<<` a **friend** function of class `String`
 - Can pass `ostream & os` as a `const` if you would like :)

Overall Class Header File

```
class String {
private:
    char *str;
    int len;
public:
    String();
    String(const String & s);
    String(const char * s);
    ~String();
    // Member functions
    :
    :
    friend bool operator==(const char *, const String &);
    friend String operator+(const char *, const String &);
    friend ostream & operator<<(ostream &, const String &);
};
```

Notes:

1. `friend bool operator==(const char *, const String &);`
- Friend definition for `operator==` with LHS `const char *`
2. Note two other friend definitions
- `operator+`
 - `operator<<`

And that concludes our `String` class definition!!

- Full code should be on quercus

Linked List

Moving on to a new subject: **Linked Lists**

- A class of data structures that are used as building blocks in many applications
- **Dynamic Structures**
 - Can shrink and grow
 - Unlimited/Unfixed size
- More efficient addition/deletion of elements

Operations on **Linked Lists**:

- **Traversal**
- Locating nodes
- Inserting nodes
- Deleting nodes

Linked List Definition

Linked Lists:

- Arbitrarily long sequence of **nodes**
 - Each **node** contains
 - a. A data field (**key**)
 - b. A pointer to the next **node** in the list
- **Linked List's** have a **head** which points to the first node in the list
- If $\text{key}_1 < \text{key}_2 < \text{key}_3 < \dots < \text{key}_n$
 - The list is **sorted/ordered**

Linked List Class Definitions

```
class listNode{
public:
    int key;
    listNode* next;
};

listNode node;
```

1. Members are public only for presentation purposes
2. `int key;`
- `listNode` contains a key
3. `listNode* next;`
- `listNode` also contains pointer to next element in the **Linked List**

```
class linkedList{
public:
    listNode* head;
};

linkedList myList;
```

Notes:

1. Members are public only to simplify presentation

2. `listNode* head;`
- `linkedList` object contains a pointer to the *first* or **head** node of the **linked list**

Traversing a Linked List

The idea is to visit each node once and process the data

```
void linkedList::traverseList( ) {
    listNode* tptr = head;
    while (tptr != NULL) {
        cout << tptr->key << endl;
        tptr = tptr->next;
    }
}
```

Notes:

1. `listNode* tptr = head;`
- Start by setting a **Traversal Pointer** to the **head** node
2. `while(tptr!=NULL){ }`
3. `tptr = tptr->next;`
- Set the **traversal pointer** to the *next* node in the list
 - On next iteration of `while`, `tptr` refers to the *next* node in list
4. This works even if the list is empty
 - Since `tptr!=NULL` will evaluate to false

Locating a Node in a Linked List

```
listNode* linkedList::LocateInList(int k) {
    listNode* tptr = head;
    while (tptr != NULL) {
        if (tptr->key == k) break;
        tptr = tptr->next;
    }
    return tptr;
}
```

Process:

1. Iterate through the list by using **traversal pointers**
2. Check if `tptr->key == k`
 - Check if we have found our node
 - If so, we have found our node, so `break`;
3. Set `tptr = tptr->next;`

Notes:

- This works as well if list is empty, since `tptr!=NULL` will also evaluate to false

Inserting a value at Head

```
void linkedList::insertAtHead(listNode* nptr) {
    nptr->next = head;
    head = nptr;
}
```

Notes:

1. nptr->next = head;
- First set the *next* node for *nptr* to the current **head**
2. head = nptr;
- Set the **head** node to be *nptr*

The operations for inserting at head need to be carried out in this order

- Think about if the operations were flipped
 - head=nptr;
 - nptr->next = head;
 - Except at this point you would have lost the original value of *head*

If you want to traverse backwards through a **linked list**, you can implement a **doubly linked list**

■ Lec_23.md

Lecture 23 - Linked Lists

Nov. 03/2020

Announcements:

Final Summative Assessment

- 2.5 hour final exam
- Monday, December 14, 2020 starting at 9:30 am EST
- Synchronous Exam
 - Everyone writes the exam at the same time
- Covers all the lecture material from start of term until end of term
 - Covers all lab assignments
- Open textbook and open notes, but no calculators, compilers, IDEs
- Will use Quercus Quizzes
- Past finals posted on Quercus
 - Good practice, coverage varies

Quiz 4

- 45 minute quiz (longer than usual)
- Friday, November 6, 2020
 - Take it any time in the day
- If you leave the quiz you can't go back to it
- Cover all the lecture material from start of semester until end of this lecture
- Covers Lab 1 to Lab 3
- Open textbook and open notes

Linked Lists

```
class listNode{  
public:  
    int key;  
    listNode* next;  
};
```

```
listNode node;
```

```
class linkedList{  
public:  
    listNode* head;  
};
```

```
linkedList myList;
```

Inserting a value at Tail

Given a new node, want to insert the node so it is the *last* node in the list

- Must start traversing **linked list** at head
 - Use `tptt` to traverse the **linked list**
 - Stopping when `tptt->next` is `nullptr`

```
void linkedList::insertAtTail(listNode* nptr) {
    listNode* tptt = head;
    while (tptt->next != NULL)
        tptt = tptt->next;
    tptt->next = nptr;
    nptr->next = NULL;
}
```

Notes:

1. `while (tptt->next != NULL)` and `tptt= tptt->next;`
- Traverse through the list using `tptt`
- Stop when `tptt->next` is `NULL`
 - This means `tptt` points to the **last** node in the **linked list** or the **tail** node
2. `tptt->next = nptr;`
- Set new **tail** node to `nptr`
3. `nptr->next = NULL;`
- Set the `next` field for the **tail** node to null (new **tail** node)
4. What if the **linked list** is empty?
- e.g. the **head** node is `null` ?
 - Then you get a segmentation fault when running `tptt->next` as `tptt` currently points to `head`, and `head` has no value
- So we need to protect against this case

```
void linkedList::insertAtTail(listNode* nptr) {
    if (head == NULL) {
        head = nptr;
        nptr->next = NULL;
    }
    else {
        listNode* tptt = head;
        while (tptt->next != NULL)
            tptt = tptt->next;
        tptt->next = nptr;
        nptr->next = NULL;
    }
}
```

Notes:

1. `if(head == NULL){ ... }`
- Check if **linkedlist** is empty
- Prevents the seg fault from earlier

Inserting a value in the Middle

Given a key `k`, located a node with this key and then insert the new node pointed to by `nptr` after the located node

- Assuming k exists
 - You can define behaviour if k DNE
- Need to *break* the **linked list** chain
 - Make `nptr->next` point to the element after insertion point
 - Make `tptp->next` point to the inserting node, `nptr`

```
void linkedList::insertInMiddle(int k, listNode* nptr) {
    listNode* tptp = head;
    while (tptp != NULL) {
        if (tptp->key == k) break;
        tptp = tptp->next;
    }
    nptr->next = tptp->next;
    tptp->next = nptr;
}
```

Notes:

1. `while (tptp != NULL){ ... }`
- Traverse list until you find node with key k
2. `nptr->next = tptp->next;`
- Assign the `nptr->next` to point to `tptp->next`;
3. `tptp->next=nptr;`
- Assign the `tptp->next` to point to `nptr`
4. 2 and 3 *must* be executed in this order
- If 3 is executed first, then the value of `tptp->next` will be lost

Deleting a Node

Given a **key** k, locate a node with this key and delete it

- Idea: Use *two* traversing pointers, `tptp` and `pptp`
 - `pptp` lags `tptp` by 1 node

```
void linkedList::deleteNode(int k) {
    listNode* tptp = head;
    listNode* pptp = NULL;
    while (tptp != NULL) {
        if (tptp->key == k) break;
        pptp = tptp;
        tptp = tptp->next;
    }
    pptp->next = tptp->next;
    delete tptp;
}
```

Notes:

1. `pptp->next = tptp->next;`
- Since `pptp` lags `tptp` by 1 node, this detaches/removes one node from the list
2. Tons of edge cases for this lmao

- head is null (list empty)
- key not found
- trying to delete head node

Addressing key not found:

```
void linkedList::deleteNode(int k) {
    listNode* tptr = head;
    listNode* pptr = NULL;
    while (tptr != NULL) {
        if (tptr->key == k) break;
        pptr = tptr;
        tptr= tptr->next;
    }
    if(tptr == NULL) return; //fix
    pptr->next = tptr->next;
    delete tptr;
}
```

Addressing head is null, or the list is empty:

```
void linkedList::deleteNode(int k) {
    if(tptr == NULL) return; //fix
    listNode* tptr = head;
    listNode* pptr = NULL;
    while (tptr != NULL) {
        if (tptr->key == k) break;
        pptr = tptr;
        tptr= tptr->next;
    }
    if(tptr == NULL) return;
    pptr->next = tptr->next;
    delete tptr;
}
```

Notice that this covers the *key not found* case above, so we can replace this implementation with one that covers both cases, in fewer lines of code

```
void linkedList::deleteNode(int k) {
    if(tptr == NULL) return; //kept fix here, this covers fix below
    listNode* tptr = head;
    listNode* pptr = NULL;
    while (tptr != NULL) {
        if (tptr->key == k) break;
        pptr = tptr;
        tptr= tptr->next;
    } //deleted fix below
    pptr->next = tptr->next;
    delete tptr;
}
```

Addressing deleting the head node

```
void linkedList::deleteNode(int k) {
    if(tptr == NULL) return;
    listNode* tptr = head;
    listNode* pptr = NULL;
    while (tptr != NULL) {
        if (tptr->key == k) break;
        pptr = tptr;
        tptr= tptr->next;
    }
```

```

    }
    if (tptr == head) {
        head = head->next;
        delete tptr;
        return;
    }
    pptr->next = tptr->next;
    delete tptr;
}

```

Notes:

1. if (tptr == head) { ... }
- Adds a case for if the node we are deleting is the head node

listNode Class Definition

```

class listNode {
private:
    int key;
    listNode* next;
public:
    listNode();
    listNode(int k);
    listNode(int k, listNode* n);
    listNode(const listNode& other);
    ~listNode();
    int getKey() const;
    listNode* getNext() const;
    void setKey(int k);
    void setNext(listNode* n);
    void print() const;
};

```

listNode Accessors/Mutators/Print

```

int listNode::getKey() const {
    return (key);
}
listNode* listNode::getNext() const {
    return (next);
}
void listNode::setKey(int k) {
    key = k;
}
void listNode::setNext(listNode* n) {
    next = n;
}
void listNode::print() const {
    cout << "(" << key << "," << next << ")";
}

```

listNode Constructors

```

listNode::listNode() {
    key = 0;
    next = NULL;
}
listNode::listNode(int k) {
    key = k;
}

```

```

    next = NULL;
}
listNode::listNode(int k, listNode* n) {
    key = k;
    next = n;
}
listNode::listNode(const listNode& other) {
    key = other.key;
    next = other.next;
}
listNode::~listNode() {
    //Nothing to do
}

```

Notes:

1. listNode::listNode(const listNode& other)
 - Method shallow on purpose
 - Design decision
2. listNode::~listNode()
 - Method shallow on purpose
 - When one node is deleted
 - Should all the *next* nodes also be deleted?
 - Would be a **deep** implementation
 - Or rather keep *next* nodes
 - **Shallow** implementation
 - You have control: you can define if you want **shallow** or **deep** behaviour

linkedList Class Definition

Moving on to the **linked list** class definition

```

class linkedList {
private:
    listNode* head;
public:
    linkedList();
    linkedList(const linkedList& other);
    ~linkedList();
    linkedList & operator=(const linkedList& rhs);
    bool insertKey(int k);
    bool deleteKey(int k);
    bool keyExists(int k);
    void print() const;
};

```

linkedList Constructors

```

linkedList::linkedList(){
    head = NULL;
}

linkedList(const linkedList& other){
    listNode* ptr = other.head;
    listNode* nptr = NULL;
    head = NULL;
    while(ptr != NULL) {
        nptr = new listNode(ptr->getKey());

```

```
// insert *nptr at end of list
ptr = ptr->getNext();
}
}
```

Notes:

1. `linkedList::linkedList()`
 - Create a new **linked list**, with head `NULL`
2. We want the **copy constructor** to have a **deep** implementation
 - If we copy a **list**, we want to keep all the *next* nodes
3. `linkedList(const linkedList& other)`
 -

■■ Lec_24.md

Lecture 24 - Linked Lists and Recursion

Nov. 04/2020

Linked Lists

From previously, we built the **Linked List** class definitions:

listNode

```
class listNode {
private:
    int key;
    listNode* next;
public:
    listNode();
    listNode(int k);
    listNode(int k, listNode* n);
    listNode(const listNode& other);
    ~listNode();
    int getKey() const;
    listNode* getNext() const;
    void setKey(int k);
    void setNext(listNode* n);
    void print() const;
};
```

linkedList

```
class linkedList {
private:
    listNode* head;
public:
    linkedList();
    linkedList(const linkedList& other);
    ~linkedList();
    linkedList & operator=(const linkedList& rhs);
    bool insertKey(int k);
    bool deleteKey(int k);
    bool keyExists(int k);
    void print() const;
};
```

linkedList Copy Constructor

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

linkedList(const linkedList& other){
    listNode* ptr = other.head;
    listNode* nptr = NULL;
    head = NULL;
    while(ptr != NULL) {
```

```

nptr = new listNode(ptr->getKey());
// insert *nptr at end of list
ptr = ptr->getNext();
}
}

```

Notes:

1. linkedList::linkedList()

 - Create a new **linked list**, with head `NULL`

2. We want the **copy constructor** to have a **deep** implementation
 - If we copy a list, we want to keep all the *next* nodes
 - But don't want to have error of shared object data
3. linkedList(const linkedList& other)

Alternatively,

```

linkedList::linkedList(const linkedList& other) {
    listNode* ptr = other.head;
    listNode* last = NULL;
    listNode* nptr = NULL;
    head = NULL;
    while(ptr != NULL) {
        nptr = new listNode(ptr->getKey());
        if (last == NULL) head = nptr;
        else last->setNext(nptr);
        ptr = ptr->getNext();
        last = nptr;
    }
}

```

linkedList Destructor

If we don't write a destructor, the default one is given

- `listNode* head` is deleted
 - Results in **memory leaks**

So we must implement the destructor **deeply**

```

linkedList::~linkedList() {
    listNode* ptr;
    while(head != NULL) {
        ptr = head;
        head = ptr->getNext();
        delete ptr;
    }
}

```

linkedList operator=

If we don't write an overload for the `operator=` operator

- C++ provides a **shallow** implementation
 - Results in *shared object data*

So we must also implement the `operator=` **deeply**

```
linkedList& linkedList::operator=(const linkedList& rhs) {
    if (this == &rhs) return (*this);
    // Delete the list of *this, see destructor's code
    // Allocate new listNode's and copy from rhs, see copy
    // constructor's code
    return (*this);
}
```

Notes:

1. `if (this == &rhs) return (*this)`
 - Prevents deletion of current object

Recursion

Recursion is a programming mechanism for implementing **divide-and-conquer** algorithms

- Implemented using **recursive** methods/functions
 - **recursive** methods/functions are those that call themselves
- Will examine:
 - **Recursive definition**
 - **Recursive functions**
 - Tracing **recursive functions**
 - How to use **recursion** to solve a problem

Recursive Definition

How do you calculate a factorial?

```
f(n) = 1      , if n = 1 //basis
f(n) = n*f(n-1) , if n > 1 //recursive
```

Notes:

1. The **basis**, or the **base case** is the simplest version of the problem
2. The **recursive** portion is the **recursive** call that runs
 - This divides the problem until reaching the **basis**

Implementation:

```
int factorial (int n) {
    if (n == 1) return (1);
    return (n*factorial(n-1));
}
```

Notes:

1. Notice that this implementation models the mathematical function above
2. `return n*(factorial(n-1))`
 - Returning a call to the function itself (**recursive call**)

Recursive Process

Lets trace out the recursive calls (**Recursive Trace**)

Downwards

```
//for example, calculate 4! (or factorial(4))
int factorial (int n) { // n = 4
    if (n == 1) return (1);
    return (n*factorial(n-1));
}
int factorial (int n) { // n = 3
    if (n == 1) return (1);
    return (n*factorial(n-1));
}
int factorial (int n) { // n = 2
    if (n == 1) return (1);
    return (n*factorial(n-1));
}
int factorial (int n) { // n = 1
    if (n == 1) return (1); // returns n = 1 here
    return (n*factorial(n-1));
}
```

Notes:

1. There are 4 **stack frames** created here
 - A **stack frame** can be thought of a "copy" of a function in memory
 - The **stack frames** are written out above, but **don't** exist in code

Upwards

```
//using same example above
//the last function that ran was factorial(1)
//continuation of above
int factorial (int n) { // n = 1
    if (n == 1) return (1); // returns n = 1 here
    return (n*factorial(n-1));
}
int factorial (int n) { // n = 2
    if (n == 1) return (1);
    return (n*factorial(n-1)); // returns (n=2)*(1) = 2
}
int factorial (int n) { // n = 3
    if (n == 1) return (1);
    return (n*factorial(n-1)); // returns (n=3)*(2) = 6
}
int factorial (int n) { // n = 4
    if (n == 1) return (1);
    return (n*factorial(n-1)); // returns (n=4)*(6) = 24
}
```

So finally, `factorial(4)` returns 24

- Which is the value of 4!

Thinking Recursively

Given a problem, what steps are there to develop a recursive solution?

1. Find the Basis

- Identify the simplest size of the problem you can solve

2. Define the Recursion

- For any case larger than the **basis**
 - Think about dividing the problem into parts that look like the original problem, but smaller in size
 - Goal is to reach the basis

3. Write the Code

4. Trace the Code to validate

Summing an Array

1. Define **basis**:

- The sum of an array with size 1 is
 - Just the value of that element

2. Define **recursive**

- The sum of an array with size 2 is
 - The value of the left element + the *sum of the array on the right*
 - $\text{sum}(a[0:n]) = \text{sum}(a[0]) + \text{sum}(a[1:n])$
 - In this case, $\text{sum}(a[0]) = a[0]$
 - Same process as **basis**
 - The *sum of the array on the right* is the **basis**
- Expanding to higher dimensions,
- The sum of an array with size n is
 - The value of the left element + the *sum of the array on the right*
 - $\text{sum}(a[0:n]) = \text{sum}(a[0]) + \text{sum}(a[1:n])$
 - $\text{sum}(a[1:n]) = \text{sum}(a[1]) + \text{sum}(a[2:n])$
 - $\text{sum}(a[2:n]) = \text{sum}(a[2]) + \text{sum}(a[3:n])$
 - ...
 - $\text{sum}(a[n]) = \text{sum}(a[n]) = a[n]$
 - This is the **basis**

Implementation:

sum_array

```
int sum_array(int* a, int left, int right) {
    if (left == right) return a[left];
    else return a[left] + sum_array(a, left+1, right);
}
```

Tracing the Function

Given an array

a = [5,3,7]

Trace `sum_array(a,0,2)` (sum a from index 0 to 2, or just sum all of a)

Downwards

```
//left = 0
//right = 2
int sum_array(int* a, int left, int right) {
    if (left == right) return a[left];
    else return a[left]+ sum_array(a, left+1, right); //recursive call
}

//left = 1
//right = 2
int sum_array(int* a, int left, int right) {
    if (left == right) return a[left];
    else return a[left]+ sum_array(a, left+1, right); //recursive call
}

//left = 2
//right = 2
int sum_array(int* a, int left, int right) {
    if (left == right) return a[left]; //now start traversing back up recursion
    else return a[left]+ sum_array(a, left+1, right);
}
```

Upwards

```
//left = 2
//right = 2
int sum_array(int* a, int left, int right) {
    if (left == right) return a[left]; //return 7
    else return a[left]+ sum_array(a, left+1, right); //recursive call
}

//left = 1
//right = 2
int sum_array(int* a, int left, int right) {
    if (left == right) return a[left];
    else return a[left]+ sum_array(a, left+1, right);
    //return a[1] + 7
    //a[1] + 7 = 3 + 7
    //return 10;
}

//left = 0
//right = 2
int sum_array(int* a, int left, int right) {
    if (left == right) return a[left]; //now start traversing back up recursion
    else return a[left]+ sum_array(a, left+1, right);
    //return a[0] + 10;
    //a[0] + 10 = 5 + 10;
    //return 15;
}
```

So finally, `sum_array(a,0,2)` returns 15!

■ Lec_25.md

Lecture 25 - Recursion

Nov. 05/2020

Announcements:

Reminder that we have a quiz tomorrow

- 45 minutes (usually they are 30 minutes)
 - More questions?

Recursion

Thinking Recursively

Recall the process for thinking recursively:

1. Find the **Basis**
- Identify the simplest size of the problem you can solve
2. Define the **Recursion**
- For any case larger than the **basis**
 - Think about dividing the problem into parts that look like the original problem, but smaller in size
 - Goal is to reach the basis
3. Write the Code
4. Trace the Code to validate

Palindrome

A **palindrome** is a word that reads the same from

- left to right
- right to left

Examples:

- bob
- deed
- level

Non-examples:

- tarek
- hello

Assuming an array of n characters that stores a word, lets write a recursive function that detects if this word is a **palindrome**

1. Basis

- Single characters are immediately **palindromes**

2. Recursive

- Check if the **left** and **right** characters of the single character **basis** are equal (**palindrome** condition)

```
bool isPalindrome (char* seq, int left, int right) {
    if (left == right) return true;
    if (seq[left] == seq[right]) return isPalindrome (seq, left+1, right-1);
    else return (false);
}
```

Notes:

1. `if (left == right) return true;`
- **Basis**
- For a single character, `left == right` evaluates to `true`
2. `if (seq[left] == seq[right]) return isPalindrome (seq, left+1, right-1);`
- If at any point, `seq[left]==seq[right]` evaluates to `false`
 - Then the values on the **left** and **right** of the **basis** are not equal
 - So the word is not a **palindrome**

3. Issue

- What if the word has *even length*?
 - Then the **basis** is not *one character*, but rather *two characters*

Fix:

```
bool isPalindrome (char* seq, int left, int right) {
    if (left == right) return true;
    if ( ((left+1) == right) && (seq[left] == seq[right]) ) return true;
    if (seq[left] == seq[right]) return isPalindrome (seq, left+1, right-1);
    else return (false);
}
```

Notes:

1. `if (((left+1) == right) && (seq[left] == seq[right])) return true;`
- **base case** for when the input word has *even length*

Recursion on a Linked List

As an exercise, we want to print a **linked list** in reverse order

- Print the **tail** node first
 - And consecutive nodes until we hit the **head** node

1. Basis

- The simplest case is *not* when **linked list** has size 1
- Simplest case is when the **linked list** has size 0
 - e.g. `p == NULL`, where `p` is the current node

2. Recursive

- Print the value of the previous node
- Can implement this behaviour using the code:

```
void rprint (listNode * p) {
    if (p == NULL) return;
    rprint(p->next);
    cout << p->data;
}
```

Notes:

1. if (*p* == NULL) return;
 - **Basis**
 - *p==NULL* evaluates to *true* on the *n+1* node
 - *n* the size of the **linked list**
 - Or in other words, on the *next* node for the *last* node in the **linked list**
 - This node has no data (it is *NULL*, after all)
 - So we *return*; , without doing anything
2. rprint(*p*->next)
 - **Recursive call**
 - This will **recurse downwards** (traversing through the linked list)
 - Until we hit the **Basis**
3. cout << *p*->data;
 - When the **recurse upwards** happens, this will print the **list** starting with the *last element*
 - Will print the **tail** first, and then every consecutive node until finally printing the **head** node
4. Notice there is no return
 - *return*; statement is good to have
 - *Not required* for **void** functions
5. Can we pass *listNode * p* *by reference*?
 - Yes, but it doesn't add any extra value

Quicksort

Want to sort an *N*-element array of integers *A* in ascending order.

A :

Index	Element
0	
1	
2	
3	
...	
<i>N</i> -2	

Index	Element
N-1	

Using the **Quicksort** algorithm:

1. Select one element of array
 - Identify this element as the **pivot**
2. Determine the final location of the pivot in the sorted array
 - Identify that location as **p**
3. Shuffle the elements of the array
 - All elements of the array with values *less than or equal* to value of pivot have indices *less than p*
 - All elements of the array with values *greater than or equal* to value of pivot have indices *greater than p*

This lends itself to recursion greatly:

A :

Index	Element
0	
1	
2	
3	
...	
p-1	
p	pivot , which is sorted
p+1	
...	
N-2	
N-1	

We can now consider two separate arrays

- One with the range `A[0] to A[p-1]`
- Another with the range `A[p+1] to A[N-1]`

Quicksort the left half, **Quicksort** the right half until both halves are sorted

QuickSort PseudoCode

```
void QuickSort (int *A, int left, int right) {
    int PivotIndex;
    PivotIndex = SelectAndShuffle (A, left, right);
    if (PivotIndex > left) QuickSort (A, left, PivotIndex-1);
    if (PivotIndex < right) QuickSort (A, PivotIndex+1, right);
}
```

Initial call to Quicksort: QuickSort(A,0,N-1);

Tracing Quicksort

Start with A :

Index	Element
0	26
1	33
2	35
3	29
4	19
5	12
6	22

1. Select 26 as pivot
2. Place 26 in A[3]
3. Shuffle

Index	Element
0	19
1	12
2	22
3	26
4	33
5	35
6	29

For the left side:

Index	Element
0	19
1	12
2	22

1. Select 19 as pivot
2. Place it in A[1]
3. Shuffle

Index	Element
0	12
1	19

Index	Element
2	22

For the right side:

Index	Element
0	29
1	33
2	35

1. Select 33 as **pivot**
2. Place 33 in $A[1]$
3. Shuffle

Index	Element
0	29
1	33
2	35

Upwards Traversal

Merge **left side**, **right side**, and original **pivot**

Index	Element
0	12
1	19
2	22
p	26
0	29
1	33
2	35

Which becomes

Index	Element
0	12
1	19
2	22
3	26
4	29
5	33
6	35

And now the array is *completely sorted!*

SelectAndShuffle Implementation

1. Sweep through the array
 - Start at left and end at upper bound of array $N-1$
2. When an element is found that is *smaller* than the **pivot**, place it onto the left side
3. Repeat this until the array is *pseudo-sorted* into two sections
 - One section where every value is *less* than pivot, but is *not necessarily sorted*
 - Another section where every value is *greater* than pivot, but is *not necessarily sorted*
4. Place **pivot** in between these two sections

```
int SelectAndShuffle (int *A, int left, int right) {
    int Lastsmall = left; int temp;
    for (i = left+1 ; i <= right ; ++i) {
        if (A[i] <= A[left]) {
            Lastsmall = Lastsmall + 1;
            temp = A[i];
            A[i] = A[Lastsmall];
            A[Lastsmall] = temp;
        }
    }
    temp = A[left];
    A[left] = A[Lastsmall];
    A[Lastsmall] = temp;
    return (Lastsmall);
}
```

Notes:

1. `temp = A[i]; A[i] = A[Lastsmall]; A[Lastsmall] = temp;`
 - This is code to swap two values in an array
2. `temp = A[left]; A[left] = A[Lastsmall]; A[Lastsmall] = temp;`
 - This places the **pivot** in the right location

■ Lec_26.md

Lecture 26

Nov. 17/2020

Note-takers Note: Hi, I hope you enjoyed your reading week!

Announcements

Quiz 5

- Quiz 5 is on November 20th (This Friday)
- Covers Lab 1 to 3

Binary Trees

Trees are a class of **data structure** used in many applications

- Dynamic Structures like **linked lists**
- Offer advantages over linear structures like arrays and lists
 - More efficient in *addition* and *deletion* of elements

First, will define a **Binary Tree**

- Then will look at
 - Traversals
 - Ordering (BST - **Binary Search Trees**)
 - Insertion and Deletions on BST's
 - Object-oriented implementation

Binary Tree Description

- A **binary tree** is a structure that is either
 - Empty
 - Consists of one node connected to two disjoint structures
 - Each of which is a binary tree

Any node in a **binary tree** can only have a *minimum* of 0 connections and a *maximum* of 3 connections

- One supernode
 - Nodes above it
 - **Ancestor**
- Two subnodes
 - Nodes below it
 - **Descendent**

Given a node with two subnodes:

- The node is considered a **parent** node
- The two subnodes are considered **children**

- **Left child and right child**
 - The children are considered **siblings**
- Think of a family tree
 - Immediate connections in the up/down direction correspond to parents and children

The top (first node) in the tree is called the **root** node

- The tree with **root** node of **left child** of another tree is called the **left subtree**
- Similarly, The tree with **root** node of **right child** of another tree is called the **right subtree**

A node with no subnodes (no children) is called a **leaf** node

Anything that is not a **leaf** node is an **interior node**

- **leaf** nodes are also known as **exterior nodes**

Binary Tree Properties

Unique Parent

There is a unique parent for each node in the tree, except for the root

Prove by Contradiction

- Let **A** be a Node with two parents, **B** and **C**
 - Then **A** has two **supernodes**
 - Then the nodes are *not* disjointed
 - And this is no longer a **binary tree**
- Thus, **A** can only have 1 parent (unique)

Path

The set of nodes $n_1, n_2, n_3, \dots, n_k$ is said to be the path from n_1 to n_k iff (if and only if) n_i is the parent of $n_{(i+1)}$ for $1 \leq i < k$

- Length of a path is $k-1$, or *one less* than the number of nodes in the path

Unique Path

The path between the root and each node in the tree is unique

Prove by Contradiction

- Let **A** be a tree where there are two paths from **A** to a **descendent** **B**
 - Then **B** has two **supernodes**
 - Since there are two paths that reach **B**
 - Then some two nodes are *not* disjointed
 - And this is no longer a **binary tree**
- Thus, the path from **root** node **A** to a **descendent** node **B** **must be unique**

Binary Tree Implementation

TreeNode

```
class treenode{
public:
    char data;
```

```
treenode *left;
treenode *right;
:
};
```

Notes

1. Tree **node** definition
2. char data;
- Value of a given **node**
3. treenode* left; and treenode* right;
- **children** of a given node

Tree Traversal

Given a binary tree, we would like to visit each node *once and only once*

With a **linked list**:

- There is **only one option** for traversing the **data structure**
 - ->next
 - **linked lists are linear data structures**

With **binary trees**:

- At any node, there are **three options** for traversing **data structure**
 - Visit Node
 - Traverse Left
 - Traverse Right

There are then $3!$ traversal options. Examining the most useful 3:

- **In-Order Traversal (LNR)**
 - i. Traverse Left
 - ii. Visit Node
 - iii. Traverse Right
- **Pre-order Traversal (NLR)**
 - i. Visit Node
 - ii. Traverse Left
 - iii. Traverse Right
- **Post-order Traversal (LRN)**
 - i. Traverse Left
 - ii. Traverse Right
 - iii. Visit Node

Preorder Implementation

```
void preorder(treenode* root){
    if(root!=NULL){
        cout << root->data;
        preorder(root->left);
        preorder(root->right);
```

```
 }  
 }
```

Notes:

1. You can tell this is a NLR implementation

- cout << root->data;
 - Prints **N** data
- preorder(root->left);
 - Traverses left
- preorder(root->right);
 - Traverses right

2. if(root!=NULL)

- Basis

Binary Search Tree

Binary Trees are useful, but become more useful when *order* is applied to them

A **Binary Search Tree (BST)** is a binary tree with special properties:

- Each node has a **key**
- The **key** of any node is greater than the keys of the nodes in the **left subtree**
- The **key** of any node is less than the keys of the nodes in the **right subtree**

The left and right subtree are also **BST**'s

In general, for a **BST**:

L < N < R

■ Lec_27.md

Lecture 27

Nov. 18/2020

Announcements

Quiz 5

- Quiz 5 is on November 20th (This Friday)
- Covers
 - Lab 1 to 3
 - Covers all lecture material until **end of this lecture**
- Uses "Classical Quizzes" on Quercus
- Open textbook and open notes

Binary Search Trees

A **binary search tree** has the following properties

- Each node has a key
- The key of any node is greater than **any** key in the left subtree
- The key of any node is smaller than **any** key in the right subtree
- Descendants of the root node in a **BST** is also a **BST**

BST Traversal

For a tree:

14

10 16

8 11 15

1. LNR (**In-Order Traversal**)

- LNR traversal will print:
 - 8 10 11 14 15 16
 - Notice that this is printing the tree key's in order of magnitude
 - This is how we have defined LNR (**In-Order**)

2. NLR

3. LRN

BST Searching

Objective: Given a BST with root N, find if the node with key x exists in the BST

Recall how **BST**'s are defined.

- Everything in the left subtree of a node has *smaller keys*
- Everything in the right subtree of a node has *larger keys*
- Descendants of the root node are also **BST**

Basic Search Algorithm:

1. Compare x to key(N)
2. If(x = key(N))
 - x found in BST, return
3. If(x < key(N))
 - x in left subtree, traverse left node
4. If(x > key(N))
 - x in right subtree, traverse right node

BST Search Algorithm

```
treenode* SearchBST(treenode* N, int k){
    if(N==NULL){
        return NULL;
    }
    if(N->key==k){
        return N;
    }else if(N->key<k){
        return SearchBST(N->R,k);
    }else{
        return SearchBST(N->L,k);
    }
}
```

1. if(N==NULL){ return NULL; }
- This is the recursive basis
 - If the node doesn't exist, return NULL
2. if(N->key==k){ return N; }
- Node with key k found
3. else if(N->key<k){ return SearchBST(N->R,k); }
 - Traverse right if the key k is larger than the key of node
4. else{ return SearchBST(N->L,k); }
 - Traverse right if the key k is smaller than the key of node

BST Insertion

Objective: Given a BST with root N, where do we place a *new* node with key k ?

Really very similar to searching

- Instead of returning NULL if the key does not exist, we place the new node there.

Do you want to build a BST?

Build a BST from a sequence of nodes

For a set of numbers with same values but different order, you will get a different (unique) tree

BST Sorting

Can use a BST to sort a given sequence of keys.

1. Build BST from sequence of keys
2. Traverse the BST using **In-order Traversal**

BST Deletion

Objective: Given a BST with root N and key k , we want to delete the key k in the tree such that the tree is still a BST

1. Search for node with key k
 - Denote this node r
2. If r is a leaf node
 - Can *delete immediately*
3. If r is an interior node, with one subtree
 - More complicated to deal with
4. If r is an interior node, with two subtrees
 - Even more complicated

Case: If r is an interior node with one subtree

- Can delete r , and replace r with the **root** node of the one subtree under r
 - Since the subtree under r contains values that are *all larger* than the parent of r

Case: If r is an interior node with two subtrees

- Can delete r
- Need to combine the two subtree's under r
 - Remember that because of the BST properties,
 - key (x) < key (p) < key (r) < key (q)
 - Denote p and q to be the **left subtree** and **right subtree** of r respectively
 - **4 ways** to dealing with combining these two subtrees:

1. Replace r with p , and make q the right child of the largest node in p
2. Replace r with q , and make p the left child of the smallest node in q
3. Replace r with the smallest node in q
4. Replace r with the largest node in p

Notice:

- Methods 1 and 2 **skew the tree** (make the tree deeper)
 - Tree is not nice and "balanced" like earlier
 - Balanced has a different meaning for BST's, so maybe we should use the phrase "the tree is not nice and stable like earlier"
- Methods 3 and 4 **require the smallest/largest nodes to be leaf nodes**
 - Can only replace r with single nodes to maintain BST

- Need to implement a check to make sure they are **leaf nodes**

■■ Lec_28.md

Lecture 28

Nov. 19/2020

Binary Search Trees

TreeNode Class Definition

```
class treeNode {
private:
    int key;
    treeNode* left;
    treeNode* right;
public:
    treeNode();
    treeNode(int k);
    treeNode(int k, treeNode* l, treeNode* r);
    int getKey() const;
    treeNode* getLeft() const;
    treeNode* getRight() const;
    void setKey(int k);
    void setLeft(treeNode* l);
    void setRight(treeNode* r);
    void print() const;
};
```

TreeNode Constructors

```
treeNode::treeNode() {
key = 0;
left = NULL;
right = NULL;
};

treeNode::treeNode(int k) {
key = k;
left = NULL;
right = NULL;
};

treeNode::treeNode(int k, treeNode* l, treeNode* r) {
key = k;
left = l;
right = r;
};
```

Notes:

1. treeNode()
- Can define/instantiate a BST as empty
2. treeNode(int k)

- Or with a **root** key **k**

3. `treeNode(int k, treeNode* l, treeNode* r)`

- Or with a **root** node and left/right subtrees

TreeNode Accessors/Mutators

```
int treeNode::getKey() const {
    return (key);
}
treeNode* treeNode::getLeft() const {
    return (left);
}
void treeNode::setLeft(treeNode* l) {
    left = l;
}
```

Tree Class Definition

```
class Tree {
private:
    treeNode* root;
public:
    Tree ();
    Tree (const Tree& other);
    ~Tree ();
    Tree& operator=(const Tree& rhs);
    :
    :
    void insert(treeNode* p);
};
```

Tree Constructor

```
Tree::Tree() {
    root = NULL;
};
```

Tree Destructor

Must be a deep implementation

- We need to de-allocate the tree before de-allocating the object

Deallocating the tree

- The node **root** should be deleted **last**
 - To avoid memory leaks (dangling pointers)

```
Tree::~Tree(){
    if(root==NULL){
        return;
    }
    delete_tree(treeNode* root);
}
```

Notes:

1. `delete_tree(treeNode* root);`
- Why do we need this?
 - Because we cannot explicitly call `~Tree`
 - And we want to use recursion to delete the tree
- `delete_tree` is a **helper function** that facilitates recursion
- Also, because `~Tree` is a member of class `Tree`
 - We want to deallocate all the nodes of *first*

```
Tree::delete_tree (treeNode* myroot) {
    if (myroot == NULL) return;
    delete_tree(myroot->getLeft());
    delete_tree(myroot->getRight());
    delete myroot;
}
```

Notes:

1. `if (myroot == NULL) return;`
- **basis** for recursion
2. `delete_tree(myroot->getLeft());`
- Delete left subtree
3. `delete_tree(myroot->getRight());`
- Delete right subtree
4. `delete myroot;`
- Delete **root** node
- The **root** node is deleted *last*

Tree Insertion

```
void Tree::insert(treeNode* p) {
    if (p == NULL) return; // Nothing to insert
    if (root == NULL) { // basis
        root = p; root->setLeft(NULL); root->setRight(NULL);
        return;
    }
    // Helper function to facilitate the recursion
    insert_bst(p, root);
}
```

Notes:

1. `insert_bst(p,root);`
- Why do we need to use a **helper** function again?
 - Need an additional parameter for recursion (determine where to place the inseritng node)

```
void Tree::insert_bst(treeNode* p, treeNode* r) {
    if (p->getKey() == r->getKey()) return;
```

```

if (p->getKey() < r->getKey()) {
    if (r->getLeft() == NULL) {
        r->setLeft(p);
        return;
    }
    else insert_bst(p, r->getLeft());
}

if (p->getKey() > r->getKey()) {
    if (r->getRight() == NULL) {
        r->setRight(p);
        return;
    }
    else insert_bst(p, r->getRight());
}

}

```

Notes:

1. `insert_bst` can be a **private** function
- `insert_bst` serves only to facilitate recursion
 - Make this **private** to prevent incorrect access
2. `if (p->getKey() == r->getKey()) return;`
- Recursive basis

Copy Constructor

```

void preorder (treenode *root) {
    if (root != null) {
        cout << root->data;
        preorder (root->left);
        preorder (root->right);
    }
}

```

Inheritance in C++

Inheritance is a C++ mechanism that facilitates code reuse

- Shows up in other programming languages as well (e.g. Java, Python)

Allows programmers to extend/enhance existing classes without modifying the code in these classes

- **Open Closed Principle**
 - Open for extension, Closed for modification

Inheritance Example: Name Class

```

class Name {
private:
    char * theName;
public:
    Name();
    Name(constchar* name);
    Name(Name & r);
    ~Name();
    void setName(constchar* newName);
}

```

```
Name & operator=(Name & r);
void print();
};
```

Notes:

- Think of the string class we implemented a while ago
 - Base class for an **inheritance** example

Name Class Implementation

Constructors

```
Name::Name() {
    theName = new char [1];
    theName[0] = '\0';
}
Name::Name(const char* name) {
    theName = new char[strlen(name)+1];
    strcpy(theName, name);
}
Name::~Name() {
    delete [] theName;
}
```

Notes:

1. `theName[0] = '\0';`
- Null terminated strings (cstrings)
2. `strcpy(theName, name);`
- String copy (from C standard library)

Accessors/Mutators

```
void Name::setName(const char* newName) {
    delete [] theName;
    theName = new char[strlen(newName)+1];
    strcpy(theName, newName);
}
void Name::print() {
    cout << theName << endl;
}
```

Inheritance Example: Contact Class

Now, say we want to implement a contact class, which contains the following:

- Name of a contact
 - Set name
 - Get name
- Address of a contact
 - Set address
 - Get address

We could copy over the functions/data members from String

- No code reuse
- Start from scratch

Or, we could ask to pay for a license to use the original `Name` class source code

- This works, and we get original `Name` class code
 - However, imagine if the `Name` class has 1000000000 lines
 - We need to learn how it works
 - Can sometimes takes months to understand the class
- What if the original class updates to a new version with more functionality?

But what about **Inheritance**?

- Allows programmers and engineers to *extend* the capability of a class
- Reuse the code *even when source is not provided*
- Only need to understand what `Name` does
 - Not how it does it
 - **Complexity management**
 - Only need to understand what the public data members are (object functionality)
- Updates to `Name` *automatically* reflected in `Contact`

Contact Class Implementation

```
#include "Name.h"

class Contact: public Name{
private:
    char * theAddress;
public:
    Contact();
    ~Contact();
    Contact(Contact & r);
    Contact(const char* newName,
            const char* newAddress);
    void setNameAddress(const char* newName,
                        const char* newAddress);
    Contact & operator=(Contact & r);
    void print();
};
```

Notes:

1. `class Contact: public Name`
 - This line reads as:
 - class `Contact` inherits `Name` publicly
 - This defines inheritance: `Contact` inherits members from `Name` (publicly)
2. All of the other `Contact` methods are available to use for `Contact` objects
 - And now *all* of the `Name` methods are *available to use* for `Contact` objects

`Contact` inherits from `Name` all the data members

- As well as function members
 - There are a couple of exceptions

A `Contact` object looks like this:

```

theName
:
setName(...)
print()

theAddress
:
setNameAddress...
print()

```

Note:

- Both methods in `Name` and `Contact` accessible with `Contact` Object

Contact Class Usage

```

#include "Name.h"
#include "Contact.h"

int main() {
    Name n;
    Contact c;

    n.setName("Tarek Abdelrahman");

    n.print();

    c.setName("Tarek Abdelrahman");

    c.setNameAddress("John Smith", "123 Main Street");

    n.setNameAddress("Tarek Abdelrahman", "123 Main Street");

    c.print();

    return (0);
}

```

Notes:

1. `n.setName("Tarek Abdelrahman");`
- Can use `Name` object as normal
2. `n.print();`
- This works normally as well
3. `c.setName("Tarek Abdelrahman");`
- This works, since `Contact` inherits `Name`
4. `c.setNameAddress("John Smith", "123 Main Street");`
- This works, since `setNameAddress` defined in `Contact` class
5. `n.setNameAddress("Tarek Abdelrahman", "123 Main Street");`
- This is a **compile-time error**
 - `Name` class has no method `setNameAddress`
 - **Inheritance** works one way

6. `c.print();`

- This works, since there is a `print()` method in `Contact`; however,
 - `print()` is also defined in `Name`
 - So which `print()` gets called?
 - The general idea is that the sub gets called
 - The function in the class that is inheriting (`Contact`) gets called
- `print()` in `Contact` eclipses `print()` inherited from `Name`

Note:

- We can choose which overloaded function we want to call
 - There is a way to call `print()` from `Name`
 - By default, `print()` from `Contact` is called (as it eclipses the inherited `print()`)

☰ Lec_29.md

Lecture 29

Nov. 24/2020

Inheritance

A property of Object-oriented programming and C++ mechanism that *facilitates code reuse*

Name/Contact Inheritance Example

```
class Name {
private:
    char * theName;
public:
    Name();
    Name(const char* name);
    Name(Name & r);
    ~Name();
    void setName(const char* newName);
    Name & operator=(Name & r);
    void print();
};
```

Notes:

1. Standard methods in this class

- Name();
◦ Empty constructor
- Name(const char* name);
◦ char array constructor
- Name(Name & r);
◦ Copy constructor
- ~Name();
◦ Destructor

```
class Contact: public Name{
private:
    char * theAddress;
public:
    Contact();
    ~Contact();
    Contact(Contact & r);
    Contact(const char* newName, const char* newAddress);
    void setNameAddress(const char* newName,
                        const char* newAddress);
    Contact & operator=(Contact & r);
    void print();
};
```

Notes:

1. class Contact: public Name

- This line reads as:
 - class Contact inherits Name publicly

2. Standard methods in this class

- Contact();
◦ Empty constructor
- Contact(const char* newName,const char* newAddress);
◦ char array constructor
◦ takes both a newName string field and a newAddress string field
- Contact(Contact & r);
◦ Copy constructor
- ~Contact();
◦ Destructor

Sub Class Usage

```
#include "Name.h"
#include "Contact.h"

int main() {
    Name n;
    Contact c;

    n.setName("Tarek Abdelrahman");

    n.print();

    c.setName("Tarek Abdelrahman");

    c.setNameAddress("John Smith", "123 Main Street");

    n.setNameAddress("Tarek Abdelrahman", "123 Main Street");

    c.print();

    return (0);
}
```

Notes:

1. c.setName("Tarek Abdelrahman");
- This works, since Contact inherits Name
2. c.setNameAddress("John Smith", "123 Main Street");
- This works, since setNameAddress defined in Contact class
3. n.setNameAddress("Tarek Abdelrahman", "123 Main Street");
- This is a *compile-time error*
 - Name class has no method setNameAddress
 - Inheritance works one way
- Inheritance is a *one-way street*
4. c.print();

- This works
 - But the `print()` in `Contact` gets called, since it eclipses the `print()` in `Name`

Contact Class Implementation

```
void Contact::setNameAddress(const char* newName, const char * newAddress) {
    setName(newName);
    delete [] theAddress;
    theAddress = new char[strlen(newAddress)+1];
    strcpy(theAddress, newAddress);
}

void Contact::print() {
    Name::print();
    cout << theAddress << endl;
}
```

Notes:

1. Can invoke and use member functions of `Name` as if they were member functions of `Contact`

- No need to define **objects**
- `setName(newName);`
 - Calling the `setName` function in `Name`
- `Name::print();`
 - Calling the `print()` function in `Name`
 - Notice the use of the **scope resolution operator** `::`

2. Functions in `Contact` *override* functions with the same **signature** in `Name`

- `Name::print();`
 - Calling `print()`; by itself would call `Contact::print()`

Aspects of Inheritance

General template for **inheritance**:

```
class Derived: public Base{
    ...
};
```

In general, the **derived** class inherits from the **base** class

- Questions we may have:
 - What does the **derived** class inherit?
 - What does the **derived** class *not* inherit?
 - What can the **derived** class add?
 - What happens when variables of **derived** class are created?
 - What happens when variables of **derived** class are destroyed?
- Special questions:
 - What is the relationship between objects of **base** and **derived**?
 - What is the relationship between pointers of **base** and **derived** objects?
 - What are **virtual** functions and **abstract classes**?

What is Inherited

All the data members of the base class are inherited (both private and public)

- In `Contact`, `theName` is inherited from `Name`
 - `theName` is a private data member

However, private data members of base are **not accessible in derived**

- So how do we access private data members?
 - Use public functions inherited from superclass
 - Use `setName` and `print()` from `Name`
 - Specifically, `Name::print()`

Another exception: private function members of base are **not accessible in derived**

```
void Contact::print() {
    cout << theName << endl; //NOT VALID
    cout << theAddress << endl;
}
```

Instead, use

```
void Contact::print() {
    Name::print();
    cout << theAddress << endl;
}

void Contact::setNameAddress(const char * newName,const char * newAddress) {
    setName(newName); //can call Name::setName()
    delete [] theAddress;
    theAddress = new char[strlen(newAddress)+1];
    strcpy(theAddress, newAddress);
}
```

What is NOT Inherited

The **constructors and destructors** are not inherited

- Think from procedural point of view
 - `Contact` object is an object *with* data members of `Name`
 - Need to instantiate (construct) `Contact` object
- `Contact()`, `~Contact()` must be defined
 - Default constructor is provided if none are defined

Overloaded assignment operator `operator=` is *not* inherited

- One is provided by default in `Contact`, as usual

Friend functions are *not* inherited

- Unfortunately, we can't steal (inherit) friends

What can be added by Derived

Derived can have new data members (private and public)

- `Contact` adds `theAddress()`

Derived can add new function members (private and public)

- Contact adds setNameAddress()

Object Creation

An object of type **derived** class is created

- By default, the default constructor of the **base** class is called
 - So if the **base** class has no default constructor, a *compile-time error* is thrown

Process (In Order):

1. **Derived** object is created
2. Default constructor of **base** class is called
3. Constructor of **derived** class is called

```
Contact::Contact() {
    theAddress = new char [1];
    theAddress[0] = '\0';
}
```

Notes:

1. The default constructor for the **Name** class is called
- More generally: You, the programmer, are responsible for the *new* data added by the **derived** class on the **base** class
 - Assume **Name** part is already constructed

But what if we want a *non-default constructor* of the **base** class to be called?

- How to call **Name(const char* name)** ?
 - Use an **Initializer List**

Initializer List

Initializer lists are used to initialize certain variables when a function is called

- Can be used to call the constructor of the **base** class

```
Contact::Contact(const char * newName,const char *newAddress):Name(newName) {
    theAddress = new char[strlen(newAddress)+1];
    strcpy(theAddress, newAddress);
}
```

Notes:

1. Notice in the function name line:
 - **:Name(newName) { ..function body.. }**
 - This is an **initializer list**
 - Invokes the **Name::Name(const char* name)** constructor
 - Instead of the default constructor for **Name**

Object Destruction

The order for **object destruction** is the mirror of the **object creation** process

Process (In Order):

1. Destructor of **derived** class is called
2. Destructor of **base** class is called
3. The **derived** class is deleted

Relationship between Base objects and Derived objects

This part is *important*.

C++ is a **strongly typed language**

- Variables have a single, defined type
- int variables are *only* int's, not strings, floats, or chars;
- Even auto derives a *single* type upon assignment.
 - The type of the variable *cannot* change

However, for **inheritance**, there is a special relationship between **base** and **derived** classes

1. Objects of type **derived** are *also* of type **base**

- The objects can simply *ignore* the new methods in the **derived** class
 - Pretend it is a **base** object
 - Contact can *ignore* new methods to appear as a Name class

2. Objects of type **base** are *not* of type **derived**

- Name class *cannot expand* to appear as a Contact class

Implications of Derived and Base Objects

```
Name n;
Contact c;

//assume n.operator=(Name &) defined
//assume c.operator=(Contact &) defined
n = c;
c = n;
```

Notes:

1. Question: Are the following defined?

- Assuming n.operator=(Name &) is defined
 - Notice the parameter takes a Name object by ref
- Assuming c.operator=(Contact &) is defined
 - Notice the parameter takes a Contact object by ref

2. n=c;

- Yes, n=c is defined
 - Since c is of type Contact, which is **derived** from Name
 - c can shrink (ignore new data members) to appear as a Name object
- n=c can *only* copy values defined in the Name class

3. c=n;

- No, c=n is not defined
 - n is of type Name, which is the **base** for Contact

- Larger class (Contact) can shrink
- Smaller class (Name) cannot grow

In **general**, one can use a **derived** object anywhere a **base** object can be used

```
Contact c;  
bool foo(Name x); //for any function foo  
val = foo(c);
```

Is **defined** (since Contact is **derived** from Name)

■■ Lec_30.md

Lecture 30

Nov. 25/2020

Inheritance

Implications of Derived and Base Objects

A **derived** object is *also* a **base** object

- The **derived** object has all the function/data members (public and private) of **base**
 - Can contract/shrink to form a **base** object

On the other hand, A **base** object cannot grow into a **derived** object

Inheritance Demo Notes

Polymorphism

A property of **object-oriented programming**

- Objects can behave as different objects (based on the **inheritance chain**)

Source Files

1. Constructor prototype in the `.h` file *do not* need initializer lists
- Write initializer list in function implementation in `.cpp` file
2. Source files needed to implement inheritance
- Need to provide `.h` and `.o` files
 - *Do not* need to provide `.cpp`
3. Need to provide *all* `.h` and `.o` files for *all* inherited classes in the **inheritance chain**
- Need to provide `Name.o`, `Name.h`, `Contact.o`, `Contact.h` when implementing `LongContact` which inherits `Contact`
 - And `Contact` inherits `Name`

Object Constructor

1. Constructor prototype in the `.h` file *do not* need initializer lists
- Write initializer list in **function implementation** in `.cpp` file
2. Object creation is bottom up for inherited classes.
- Use initializer lists to call base object constructors
3. Using initializer lists calls constructors on the *same* object
 - The base and derived constructors are called on *same* object

- Same location in memory
4. Forgetting the initializer list in the derived class
- Calls the **default** constructor

Object Destruction

1. Object destruction is top down for inherited classes.
- Do not need to delete base objects in derived object
 - C++ handles this object deletion for you
 - Write destructor *only* for **derived** class

Pointers to Derived and Base Objects

Pointer to objects of type **base** are *also* pointers to objects of types derived

```
Name* name_ptr;  
Contact* contact_ptr = new Contact();  
name_ptr = contact_ptr;
```

Reasoning: The derived object has **everything** the base object has

Pointer to objects of type **derived** *can not* point to objects of type **base**

```
Contact* contact_ptr;  
Name* name_ptr = new Name();  
contact_ptr = name_ptr;
```

Notes:

1. Problem: `contact_ptr` may be used to access **derived** members (`Contact`), which *potentially not exist* in **base** members (`Name`)

Lec_31.md

Lecture 31

Nov. 26/2020

Announcements

Quiz 6 runs on December 2nd

- Make sure to start before 11:30 pm EST (30 minutes before Midnight)
 - Classical quercus quizzes submit *immediately* at 12 Midnight
- Otherwise, same format as other quizzes!

Inheritance

Pointers to Derived and Base Objects

Pointer to objects of type **base** are *also* pointers to objects of types **derived**

Pointer to objects of type **derived** **can not** point to objects of type **base**

Problems with Polymorphism

If the data is purely static (e.g. no dynamic allocation), then function calls can be traced easily

```
Contact c;
Name n;

c.print();
n.print();

1. c.print() calls Contact::print()
2. n.print() calls Name::print()
```

But because **base** pointers can point to multiple types (derived and base), there arise certain problems.

```
Name* np;
Contact* cp;

cp = new Contact();
cp->setNameAddress("Tarek", "123 Main St");
np = cp;

cp->print(); //1.
np->print(); //2.
```

Notes:

1. What gets printed in `cp->print()` ?

- Does this evoke `Contact::print()` ?

2. What gets printed in `np->print()` ?

- Does this evoke `Name::print()` ?

Turn to understanding **object binding**

Object Binding

Function calls are **binded** to specific functions

Static/Early Binding

Determine the function call (in our example, either `Contact::print()` or `Name::print()`) based on the *type of the pointer*

- This is called **static binding** or **early binding**

Except, it can be *impossible* to figure out the object that a pointer is pointing to at run-time

- Base pointer can point to base, or derived
 - If `np` points to a `Contact` object, calling `np->print()` will call `Name::print()` because of **static binding**
 - We don't want this behaviour, because `np` is a `Contact` object

Dynamic/Late Binding

Determine the function call (in our example, either `Contact::print()` or `Name::print()`) based on the *type of the object* that the pointer points to.

- This is called **dynamic binding** or **late binding**

We want to tell the compiler to generate code that inspects the *type of the objects* that a pointer points to at *run-time*.

- Want to *implement* dynamic binding

Virtual Functions

First, define **function signature** to be the declaration of a function and the types of its parameters:

```
void setName(constchar* newName);
```

has the **function signature**

```
void setName(constchar*);
```

basically, the **function signature** is like a *unique ID* for a function

- No two functions can have the same **function signature**

Virtual functions allow us to implement **dynamic binding**

- Declaring a function as **virtual** also makes all of the functions with the same **function signature** *also virtual*
 - In other words, functions in derived classes with the same **signature** are *also virtual*

```
class Name {
private:
    char * theName;
```

```

public:
    Name();
    virtual ~Name();
    ...
    virtual void print(); //base class virtual print()
};

class Contact : public Name {
    private:
        char * theAddress;
    public:
        Contact();
        virtual ~Contact();
        ...
        virtual void print(); //derived class virtual print()
};

```

1. **virtual ~Name();**

- This is the declaration for a **virtual** function

2. **virtual void print();**

- Notice this is called twice
 - Once in Name
 - Once in Contact
- Only the **base** class needs to have the **virtual** keyword
 - A **virtual** function definition makes all of the functions with the same **signature** also **virtual**.

Virtual functions are dynamically binded

- A virtual function that is defined multiply in derived classes will be called *based on the type of the object* that the pointer the function is called on points to

```

Name* np;
Contact* cp = new Contact();
np = cp;
np->print();

```

1. **np->print();** prints **Contact::print()**

- If **Name::print()** is defined as **virtual void Name::print()**

Virtual Destructors

Destructors are not virtual by default

Why do we want to make destructors **virtual**?

```

Name* np;
Contact* cp = new Contact();
np = cp;

delete np;

```

Notes:

1. What gets deleted when **delete np;** is called?

- If the destructor is not **virtual**, this is actually *undefined behaviour*
- If the destructor is **statically binded**, calling **delete np;** actually calls **Name::~Name();**

- Potential to leave memory leaks
 - What if there are dynamically allocated data members in `Contact` ?

OOP Language Differences

In Java, *all methods* are defined as **virtual** by default.

- Much slower, since **virtual** functions require checks (of the object type) on **dynamic binding**

In C++, the programmer must *define* functions as **virtual**.

- Different programming philosophy:

 | Don't pay for something you don't use

- |
 - Must define something as **virtual**
 - Pay the price only if needed

■ Lec_32.md

Lecture 32

Dec. 01/2020

Announcements

Final Exam

- Summative Assessment
- 35% of ECE244 Mark
- December 14, 2020
 - Starting at 9:30am ET
 - 2.5 hour duration
 - Synchronous Exam
 - Everyone takes exam at same time, regardless of time zone

Final Exam Review Session

- December 13, 2020
 - 10am-12pm (ET)
- Come with questions!

Teaching Evaluation

- Please provide feedback!
- Completely anonymous

Inheritance

Pointers to Base/Derived Classes

From previously:

During run-time, a **base** pointer can point to either a **base** object or a **derived** object

Binding

Static Binding:

- Function bindings are determined at *compile-time*

Dynamic Binding:

- Function bindings are determined at *run-time*

Virtual Functions

Defining a function with the **virtual** keyword allows functions to *bind dynamically*

- A virtual function that is defined multiply in derived classes will be called *based on the type of the object* that the pointer the function is called on points to

virtual functions address the problem of calling the right functions with the same **signature** in **derived/base** classes; however,

- What about functions that are specific to **derived** classes?
 - e.g. `setNameAddress()`
 - Not implemented in **base** `Name`

So even with **virtual** functions, there remains a challenge with **base/derived** pointers:

```
Contact *cp;
cp = new Contact();
cp->setNameAddress("Tarek", "123 Main St");
np = cp;
np->setNameAddress("Tom", "2 Eva St");
```

Notes:

`np->setNameAddress("Tom", "2 Eva St");` is a *compile-time error*

Since the type of `np` is *unknown* at compile-time, and could be **derived** but also could be **base**

- And **base** object (`Name`) does *not* contain a function `setNameAddress`
- Error regardless of **static/dynamic** binding

Turn to **Dynamic casting**

Dynamic Casting

We can use `dynamic_cast` to determine the type of the object a **base** pointer is pointing to

- Returns a (cast) pointer to object if `*ptr` is of type `t`, otherwise returns `nullptr`

```
dynamic_cast<t>(ptr)
```

Checking type of object with `dynamic_cast`

```
Name* np;
...
if(Contact* cp = dynamic_cast<Contact*>(np)){
    cout << "np is pointing to Contact object";
} else if(Name* np = dynamic_cast<Name*>(np)){
    cout << "np is pointing to Name object";
}
```

Can use `dynamic_cast` to then correctly call **derived** functions

Type ID

Another way to `dynamic_cast` is to use **Type ID's**

- `typeid` is *compiler specific*
 - Returns a string with the *internal compiler name* for the type of a variable
 - On ECF, these names are of the format "`Xname`",
 - where X is the number of characters of the name of the object,
 - followed by the name

- e.g. 4Name , 7Contact , 11LongContact

```
#include <typeinfo>

typeid(variable).name()

#include <typeinfo>

Name* np;
...
cout<< "np is pointing to a " << typeid(*np).name()<< " object" << endl;
```

Notes:

1. typeid(*np).name()
- Returns a string with the *internal compiler name* for variable type

In general, `dynamic_cast` is better

- Not compiler-specific?

ArrayDB Example

Want to create a database system for Skule

- Student, Staff, Prof records

```
Record* _arrayDB[ _maxsize];
If ( _arrayDB[i]->getKey() == ....)
....
arrayDB[i]->print();
```

Notes:

1. arrayDB needs to know the *type* of Record
2. arrayDB needs a key to sort the Record s
3. arrayDB needs a print function that prints the Record

Base Record Class

```
class Record {
private:
    int key;
public:
    Record();
    virtual ~Record();
    void setKey(int k);
    int getKey();
    virtual void print();
};
```

Staff Record Class

```
class staffRecord: public Record {
private:
```

```

int performance[12];
float salary;
public:
Record();
virtual ~staffRecord();
void setSalary(float k);
...
virtual void print();
};

```

Notes:

1. Additional private data members (performance, salary) defined in StaffRecord
- Inheritance usages

Professor Record Class

```

class profRecord : public Record {
private:
...
public:
profRecord();
virtual ~profRecord();
...
virtual void print();
};

```

Polymorphism

```
Record* _arrayDB[ _maxsize];
```

The Record arrayDB can store 4 types of pointers:

1. studentRecord*
2. staffRecord*
3. profRecord*
4. Record*