

Specific goals for the class:

- Program in the C++ programming language
 - Implement concepts learned in previous classes in C++
 - Understand and apply the concept of encapsulation
 - Understand and apply the concept of inheritance
 - Understand and apply the concepts of polymorphism including generics
 - Understand how to read and design basic UML diagrams
 - Create a simple Command Line User Interface
 - Create a simple Graphical User Interface
 - Understand the concepts of multithreading
 - Understand the basic concepts of Software Engineering design methods
 - Compare and Contrast OO languages and procedural languages in terms of security, reliability, and reusability
 - Be able to choose an appropriate type of language structure for a given problem
-

Topics: **(VOCAB LIST AT END)**

- Basic C++ programming
 - C++ Standard Library containers and algorithms
 - Streams
 - File input/output
 - Namespaces
 - Classes/Encapsulation (Public vs Protected vs Private)
 - Inheritance
 - Polymorphism
 - Command-Line Interfaces
 - Graphical-User Interfaces
 - Multi-threading
 - UML diagrams (Class, Activity)
 - Software Design Pattern, Anti-Patterns
 - Exception Handling
 - Makefiles
 - Generics/Templates
-

Basic C++ programming

Replicate any topic covered in CSE 1310 or CSE 1320 in C++.

C++ Standard Library containers and algorithms

Know to implement the C++ standard library containers and algorithms (e.g. `std::vector<>`, `std::sort()`).

C++ standard library algorithms:

What is it? Algorithm functions kept in the `<algorithm>` header

Know how to use: `std::sort()`, be able to sort an unordered vector

Example:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

int main()
{
    //use c++11
    std::vector<int> nums = {56, 32, -43, 23};
    std::vector<std::string> names = {"John", "Jakob", "Annie", "Robbie"};
    std::vector<char> letters={'z','g','a','y'};

    std::sort(nums.begin(), nums.end());
    std::sort(names.begin(),names.end());
    std::sort(letters.begin(),letters.end());

    for (int i = 0; i < nums.size(); i++)
    {
        std::cout << nums[i]<<" ";
    }

    std::cout<<"\n"<<std::endl;

    for (std::string &s : names)
    {
        std::cout << s <<" ";
    }

    std::cout<<"\n"<<std::endl;

    for(std::vector<char>::iterator it=letters.begin();it!=letters.end();it++)
    {
        std::cout<<*it<<" ";
    }

    return 0;
}
```

C++ standard library containers:

What is it? Collection of containers (used to store elements)
Know specifically how to use: vectors and maps

Vectors:

std::vector<>
Create a vector (both empty and predetermined values values)
Use of an iterator on a vector

Check vector's size
Check if the vector is empty
Front and back commands
Push_back and Pop_back

Example:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

int main()
{
    //use c++11
    std::vector<int> nums = {56, 32, -43, 23}; //create vector with predetermined values
    std::vector<std::string> names; //create empty vector

    //use an iterator on a vector:
    for(std::vector<int>::iterator it=nums.begin();it!=nums.end();it++)
    {
        std::cout<<*it<<" ";
    }

    //using front and back
    std::cout<<"\nFirst num: "<<nums.front()<<std::endl;
    std::cout<<"Last num: "<<nums.back()<<std::endl;

    names.push_back("Sally"); //adds element to vector
    names.push_back("Mary");
    names.push_back("Harry");

    while(!names.empty()) //check if vector is empty
    {
        std::cout<<names[names.size()-1]<<" "; //check vector's size
        names.pop_back(); //gets rid of last element
    }

    return 0;
}
```

Maps:

std::map<,>
Create a map
Use of an iterator on a map
Check the map's size
Check if the map is empty
Operator[] or at commands

Search by value
Insert a pair into the map
find, search by key
count

Example:

```
#include <iostream>
#include <string>
#include <map>

int main ()
{
    //creating a map-remember we can also just create an empty map:
    // std::map<std::string,int> map_example;
    std::map<std::string,int> map_example = {{ "Jon", 100 }, { "Jane", 90 }, { "Jill", 50 }};

    //use at function vs [] operator
    map_example.at("Jon") = 20; //assign value (20) at this key (Jon)
    map_example.at("Jane") = 20;

    map_example["Jill"]=400;

    map_example.insert({"Bob", 99}); //insert pair into map
    map_example.insert({"Jon", 99}); //won't get inserted because we already have a key Jon

    //check size
    std::cout<<"Size of map:"<<map_example.size()<<std::endl;

    //check if map is empty
    std::cout<<"Map is empty: (1 true, 0 false) "<<map_example.empty()<<std::endl;

    //count-checks if key is present:
    if(map_example.count("Jon"))
    {
        std::cout<<map_example.find("Jon")->first<<std::endl; //if it does exist, use find to print it out-search by key
        std::cout<<map_example.find("Jon")->second<<std::endl;
    }

    //Billy doesn't exist
    if(map_example.count("Billy"))
    {
        std::cout<<map_example.find("Billy")->first<<std::endl; //if it does exist, use find to print it out
        std::cout<<map_example.find("Billy")->second<<std::endl;
    }

    //use iterator on map, search by value
```

```

int counter=0;
int key_search=20;

for (std::map<std::string, int>::iterator it=map_example.begin(); it!=map_example.end(); it++)
{
    if(it->second==key_search)
    {
        std::cout << it->first<<" has the key: " << it->second << std::endl; //first is the key, second is the value
        counter++;
    }
}

std::cout<<"Total found of key: "<<counter<<std::endl;

return 0;
}

```

Streams

Know the difference between an input stream and output stream

Know the standard streams: cout, cin, cerr,

For files: fstreams (i and o), special stream for strings: stringstream (i and o)

Difference between good, bad, fail, eof (state of a stream):

Stream objects keep track of the state of a stream using something called a flag

The states can be:

- **good** - ready to accept data
- **bad**- fatal error in the stream
- **fail** - a non-fatal error
- **eof**- run out of data (we can see this when reading from a file and there is no more data in the file)

How to change output formats (lecture 20):

- *Change of Base*
- *Change of floating point print out (fixed, scientific, default)*
- *Output width*
- *How to use get and getline*

Check to see if a character is a space, digit, letter, etc (1320 concept):

- Use ASCII
- Use functions isdigit() or isalpha()
- Manually have a list of chars (in a vector for example) and check

```

#include <iostream>
#include <ctype.h>

```

```

int main ()

```

```

{
    char c='g';

    if ( std::isdigit(c) )
    {
        std::cout << "Number."<<std::endl;
    }
    if(std::isalpha(c))
    {
        std::cout << "Letter. " << std::endl;
    }
}

```

File Input/Output

Read in from and write to a file. Be able to accept file name from either hard coded or from command line args

Note: Command line arg is a 1320 topic. You should be able to use argc and argv and you should know what it is

Example (read in file):

file: candies.txt:

```

Skittles 200
Snickers 400
Milka 799
Starburst 130
Godiva 500

```

```

#include <iostream>
#include <string>
#include <map>
#include <fstream>

```

//remember argc is number of command line args, argv is what each one is

// ./a.out candies.txt candies1.txt ---> argc==3, argv[0]: ./a.out, argv[1]:candies.txt, argv[2]:candies1.txt

```

int main (int argc, char **argv)
{

```

```

    std::map<std::string, int> candies;

```

//read in file

```

    std::ifstream in_file;

```

```

    std::string line;

```

```

    int calcs;

```

```

    in_file.open(argv[1]); //open whats on the command line-hardcoded would be: //in_file.open("candies.txt")

```

```

    if(!in_file.is_open())
    {

```

```

        std::cout<<"File not open."<<std::endl;
    }

```

```

else
{
    while(!in_file.eof())
    {
        in_file>>line>>cals;
        candies.insert({line,cals});
    }
}

std::cout<<candies["Snickers"]<<std::endl;

}

```

Example (write to file):

```

#include <iostream>
#include <string>
#include <vector>
#include <fstream>

int main (int argc, char **argv)
{
    std::vector<std::string> names={"Paulie", "Pete", "Bill"};

    std::ofstream outf(argv[1]);

    if(outf.is_open())
    {
        for(int i=0;i<names.size();i++)
        {
            outf<<names[i]<<std::endl;
        }
    }

    else
    {
        std::cout<<"Couldn't write to file."<<std::endl;
    }

}

```

Namespace

(class code 2)

Discussion of namespaces

Scope resolution errors

```
fluffy
23
scruffy
2
puffy
```

```
1  #include <iostream>
2
3  namespace example_one
4  {
5      int number = 13;
6      std::string word = "fluffy";
7  }
8
9  namespace example_two
10 {
11     int number = 23;
12     std::string word = "scruffy";
13 }
14
15 int main()
16 {
17     int number=2;
18     std::string word="puffy";
19
20     std::cout << example_one::number << std::endl;
21     std::cout << example_one::word << std::endl;
22
23     std::cout << example_two::number << std::endl;
24     std::cout << example_two::word << std::endl;
25
26     std::cout << number << std::endl;
27     std::cout << word << std::endl;
28
29     return 0;
30 }
```

Classes/Encapsulation

Definition of Encapsulation and Abstraction (check vocab)

Difference between a class and a struct

-struct members are public by default, class members are private by default

Tostring methods:

```
#include <iostream> // std::cout
```

```
#include <string> // std::string, std::to_string
```

```
int main ()
```

```
{
    std::string lucky = "Lucky number: " + std::to_string(13); //turn to string
```



```

std::string ages = "All ages:" + std::to_string(18+21);
std::cout << lucky << std::endl;
std::cout << ages << std::endl;
return 0;
}

```

Enum classes:

```
#include <iostream>
```

```

enum Color {red, white, blue};
enum class Animal {cat, dog}; // this is an enum class

```

```
using namespace std;
```

```
int main() {
```

```

    Color color = red;
    Color color1=Color::red;
    int num = color;

```

```
//Animal a1=cat; cant do this
```

```
Animal a = Animal::cat;
```

```
}
```

Know the difference between public, protected and private:

- public means accessible outside of class (in UML class Diagram: +)
- protected means only accessible to derived classes (in UML class Diagram: #)
- private means only accessible in that class alone (in UML class Diagram: -)

Create an instance of a class:

```

Dog d1;
Dog d1("Spike");
Dog *dl=new Dog(); //this is a pointer to an instance-added to heap, see below
delete(dl); //free memory

```

These are 1320 topics:

- Adding to stack (and knowing when it goes out of scope)
 - char letter='c'; //this goes to the stack*
- Adding to heap (and knowing how to delete it)
 - using the *new* keyword (in C, we used malloc) to add to the heap
 - using *delete* to free memory (in C, we used free)

Know how to code a class (specifically):

Constructor

Default constructor (extra info):

https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.cbclx01/cplr376.htm

Destructor

How to access functions

How to access variables: public ones directly private/protected through:

Modifier Methods (set methods)

Accessor Methods (get methods)

Operator overloading

Friend classes/functions (Lecture 17):

Friend class:

```
#include <iostream>
```

```
#include <string>
```

```
class Popcorn{
```

```
    std::string flavor;
```

```
protected:
```

```
    float price;
```

```
public:
```

friend class Movie_goer; /*Popcorn is now a friend class of Movie_goer so Movie_goer can access private and protected members. Note that we would have to declare Movie_goer as a friend class to Popcorn to have it access private and protected members of Popcorn (see next example). Just because class A is friends with class B does not mean class B is friends with class A-you would need to declare them both as friends.*/

```
    Popcorn(std::string flavor, float price)
```

```
{
```

```
    this->flavor=flavor;
```

```
    this->price=price;
```

```
}
```

```
};
```

```
class Movie_goer{
```

```
    std::string name;
```

```
    bool likes_popcorn;
```

```
public:
```

```
    Movie_goer(std::string name, bool likes_popcorn)
```

```
{
```

```
    this->name=name;
```

```
    this->likes_popcorn=likes_popcorn;
```

```
}
```

```
void see_info(Popcorn p) /*we can access private and protected variables from Popcorn from a function in
Movie_goer since we declared Popcorn as a friend class of Movie_goer*/
```

```
{
    std::cout<<"Flavor is: "<<p.flavor<<std::endl;
    std::cout<<"Price is: $"<<p.price<<std::endl;
}
};
```

```
int main(int argc, char **argv)
{
    Movie_goer m1("Bobby", true);
    Popcorn p1("Butter", 7.99);

    m1.see_info(p1);
}
```

Friend function (methods):

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

```
class Paper {
    int width, height;
public:
    Paper()
    {
    }
    Paper (int x, int y) : width(x), height(y) //different syntax to give values
    {
    }
    int area()
    {
        return width * height;
    }
    friend Paper friendfunction (Paper& p); //friendfunction is a friend of Paper, so it can access private members
};
```

```
Paper friendfunction (Paper& p) //definition of friendfunction
```

```
{
    Paper r;
    r.width = p.width*5;
    r.height = p.height*5;
    return r;
}
```

```
int main () {
    Paper foo;
    Paper p1 (2,3);
    foo = friendfunction (p1);
}
```

```
cout << foo.area() << '\n';
return 0;
}
```

Inheritance

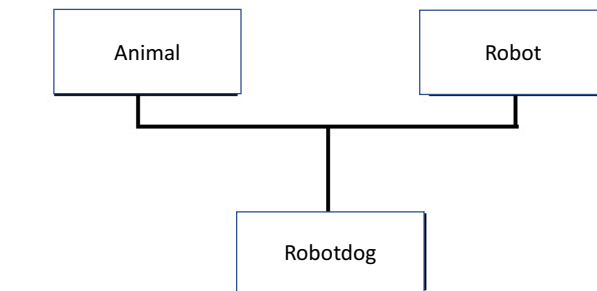
Know the Definition of Inheritance (check vocab)

-Also, Know the definition of base (parent, super) class vs derived (child, extended) classes

Know how to inherit:

- Single Inheritance (check lecture notes), Multiple inheritance
- Different ways to inherit (what is inherited and what is not): public, private protected and what that means
 - *Example: Is a protected variable inherited when we publically inherit? Yes*

Multiple Inheritance:



```
class Animal{
```

```
};
```

```
class Robot{
```

```
};
```

```
//inherits from both Animal and Robot
```

```
class Robotdog:public Animal, public Robot{
```

```
};
```

Know how to call functions from the base class in the class (particularly constructors):

- Delegation: Constructors are NOT inherited, they are delegated (actual term to use-do not call it inheriting constructors) introduced in C++11:

```
class Pretzel:public Food{
```

```
using Food::Food;
```

```
};  
-----
```

```
Employee::Employee(int id, std::string name):Person(name)  
{  
    id_num=id;  
}
```

Function overloading from the base class

Know how to call base class functions from an instance of the derived class

Polymorphism

Know the definition of Polymorphism (check vocab)

Virtual keyword when it applies to functions

Difference when a function is set to virtual and when it is not

Abstract methods (pure virtual functions)

Abstract Classes

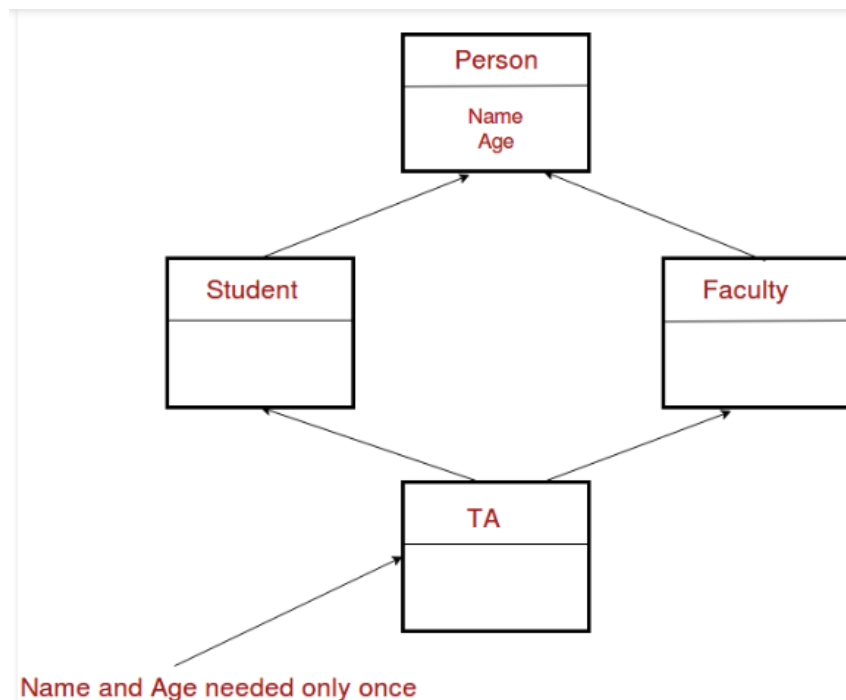
Creating a base class reference to a derived object

Casting

Calling derived object functions from a base class reference

What is the diamond problem and how to solve it

Using a virtual class



```
#include<iostream>  
using namespace std;  
class Person {  
public:  
    Person(int x)
```

```

    {
        cout << "Person::Person(int ) called" << endl;
    }
};

```

```

class Faculty : public Person {
public:
    Faculty(int x):Person(x)
    {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

```

```

class Student : public Person {
public:
    Student(int x):Person(x)
    {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

```

```

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x)
    {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

```

```

int main()
{
    TA ta1(30);
}

```

Make them virtual:

```

#include<iostream>
using namespace std;

```

```

class Person {
public:
    Person(int x)
    {
        cout << "Person::Person(int ) called" << endl;
    }
    Person()
    {
        cout << "Person::Person() called" << endl;
    }
};

```

```

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x)
    {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : virtual public Person {
public:
    Student(int x):Person(x)
    {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

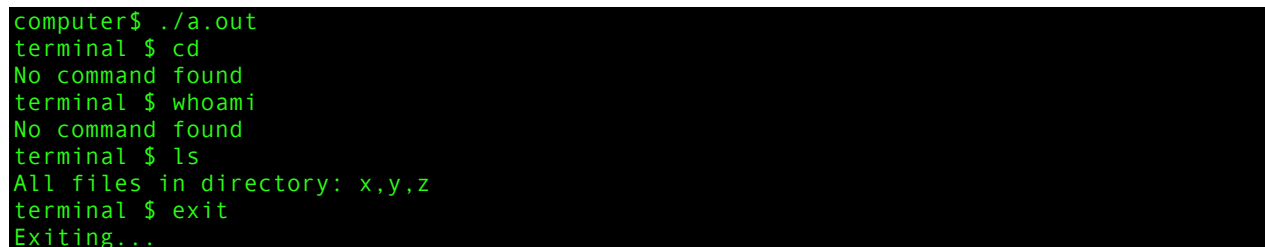
class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x), Person(x)
    {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}

```

Command Line Interfaces/Graphical-User Interfaces

Create a command line interface (you should be able to do this using concepts from 1320 and 1310):



```

computer$ ./a.out
terminal $ cd
No command found
terminal $ whoami
No command found
terminal $ ls
All files in directory: x,y,z
terminal $ exit
Exiting...

```

```

#include <iostream>
#include <string>

```

```
//commands: ls, exit
```

```
int main ()
```

```

{
std::string input="";
std::string currentdirectory="terminal $ ";

while(input!="exit")
{
std::cout<<currentdirectory;
std::cin>>input;

if(input=="ls")
{
std::cout<<"All files in directory: x,y,z"<<std::endl;
}

else if(input!="exit" && input!="ls")
{
std::cout<<"No command found"<<std::endl;
}

else
{
std::cout<<"Exiting..."<<std::endl;
}
}
}

```

Create a GUI using GTKMM from code, not from a GUI editor such as glade

-At a minimum you should be able to create the following (both using pointers and not using pointers): (check class codes)

- Message Dialog
- Dialog
- File open/save as dialog (next lecture)
- Box
- Buttons and callback functions (signal handlers):
<https://developer.gnome.org/gtkmm-tutorial/stable/sec-connecting-signal-handlers.html.en>
- Entries
- Label
- Window
- Menu items
- Menu
- MenuBar

Multi-threading

Definition of processes and threads (check vocab list)

Understand the concept of multi-threading

Be able to create a thread

Be able to sleep a thread

Know what a mutex is

Know how to lock/unlock resources

Know how to join threads back to the main

```
#include <iostream>    // std::cout
#include <thread>       // std::thread, std::this_thread::sleep_for
#include <chrono>       // std::chrono::seconds

void pause_thread(int n, std::string th)
{
    std::this_thread::sleep_for (std::chrono::seconds(n)); //sleep means the execution of the thread pauses
    //for the given time
    std::cout << "For "<<th<<","pause of " << n << " seconds ended\n";
}

int main()
{
    //create threads-each of them has the list of instructions given by pause_thread
    std::thread t1 (pause_thread,3,"t1");
    std::thread t2 (pause_thread,3,"t2");
    std::thread t3 (pause_thread,3,"t3");
    std::cout << "Joining threads....\n";

    t1.join(); //thread executes-function returns when thread is done executing
    t2.join();
    t3.join();
    std::cout << "All threads joined.\n";

    return 0;
}
```

We can “lock” our execution down so we don’t have threads “running” into each other (one executes at a time-for example, using std::cout one at a time) using something called a mutex (*a mutual exclusion object that prevents two properly written threads from concurrently accessing a critical resource*)

```
C++ computer$ ./a.out
Joining threads!
For t1,pause of 3 seconds ended
For t2,pause of 3 seconds ended
For t3,pause of 3 seconds ended
All threads joined.
```

```
#include <iostream>    // std::cout
#include <thread>       // std::thread, std::this_thread::sleep_for
#include <chrono>       // std::chrono::seconds
#include <mutex>
```

```

void pause_thread(int n, std::string th, std::mutex &mtx)
{
    std::lock_guard<std::mutex> lock(mtx);
    std::this_thread::sleep_for (std::chrono::seconds(n));
    std::cout << "For "<<th<<","pause of " << n << " seconds ended\n";
}

```

//You can also manually lock and unlock the mutex (lock_guard handles that for you):

```

//void pause_thread(int n, std::string th, std::mutex &mtx)
//{
//    mtx.lock();
//    std::this_thread::sleep_for (std::chrono::seconds(n));
//    std::cout << "For "<<th<<","pause of " << n << " seconds ended\n";
//    mtx.unlock();
//}

```

```

int main()
{
    std::mutex mtx;
    std::thread t1 (pause_thread,3,"t1", ref(mtx));
    std::thread t2 (pause_thread,3,"t2", ref(mtx));
    std::thread t3 (pause_thread,3,"t3", ref(mtx));
    std::cout << "Joining threads!\n";
    t1.join();
    t2.join();
    t3.join();
    std::cout << "All threads joined.\n";

    return 0;
}

```

UML Class Diagrams (Class/Activity)

Create a class UML from a text description (you may also be given a UML diagram and write code from that)

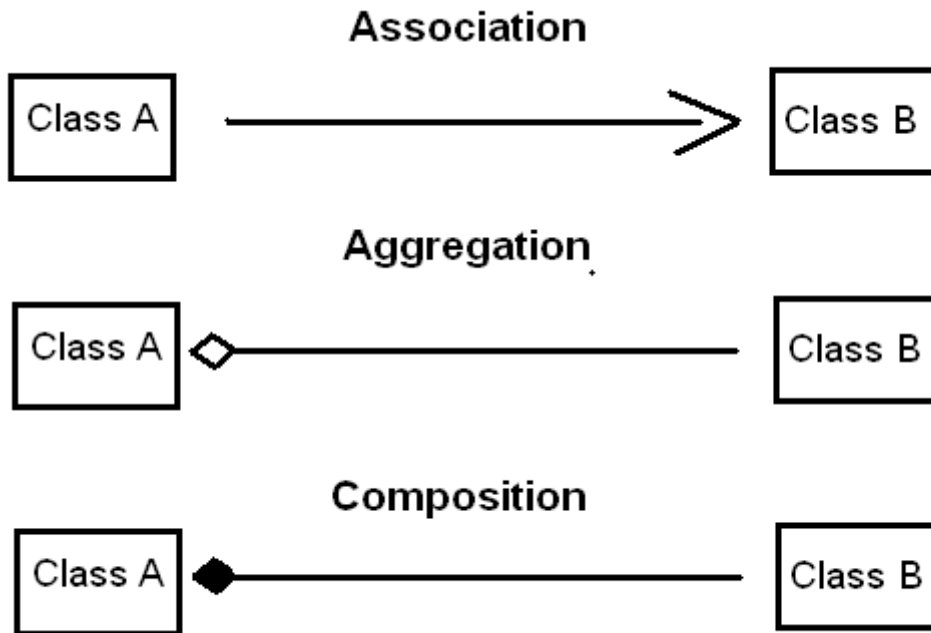
Different notations between public/private/protected:

- Public: +
- Private: -
- Protected: #

Know how to write out variables and functions:

Animal
-type: string -color: string #age: int
+ feed(price: float)

Know how to show relations between classes (association, composition, aggregation, inheritance):



Association: two classes need to show a relationship to each other (In the above example, Class A is using Class B-Class A knows that Class B exists, but Class B does not necessarily know that Class A exists). This is a more general term, the following two terms are more specific if we want to tell more about the relationship of classes.

More specific:

Aggregation: Class B can exist without class A

Example: a Student (Class B) can exist without a Classroom (Class A)

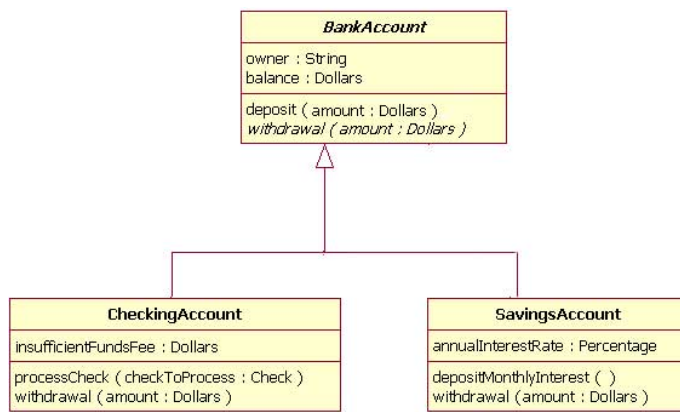
a Family (Class B) can exist without a House (Class A)

Composition: Class B cannot exist (or should not exist) without class A

Example: a Room (class B) cannot exist without a Building (class A)

a Hand (class B) cannot without a Body (class A)

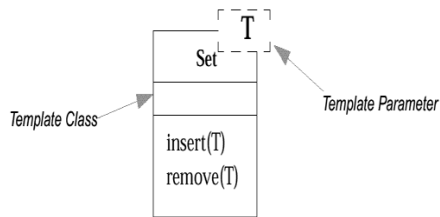
Inheritance: (multiple inheritance included in previous section)



taken from: <https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/index.html>

Note: BankAccount is abstract (class name written in italics)

Template class diagram:



Software Design Pattern

(check vocab list)

Basic definition of a software design pattern

Basic definition of an anti-pattern

Exception Handling

Know how to handle exceptions:

Try Catch Blocks

Create custom exception classes by extending exception:

```
class My_exception: public std::exception{....}
```

Be able to throw exceptions

C++ standard exceptions: (these are inherited from the exception class)

`Bad_alloc`

`Bad_cast`

`Bad_exception`

`Bad_typeid`

`Bad_function_call`

Bad_weak_ptr
Logic_error
runtime_error

Note:

Runtime error happens while your program is running (seg fault)

Compile time error happens when compiling your program (these are caught while compiling, syntax errors etc)

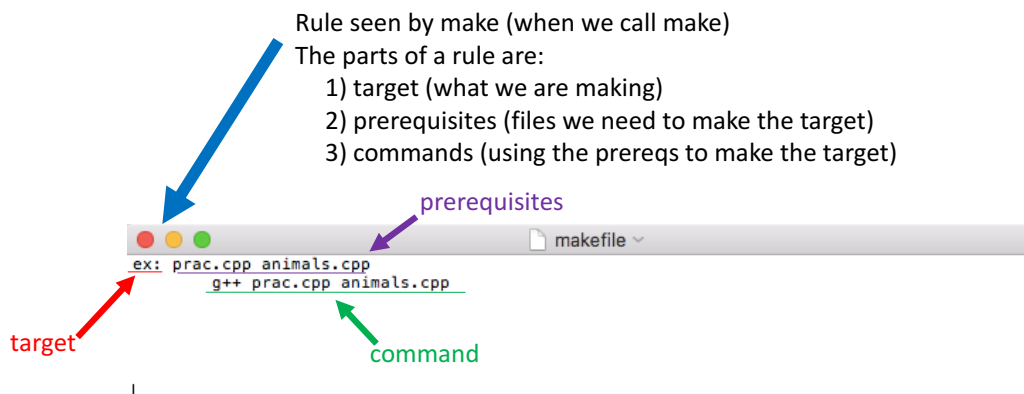
Makefiles

Make makefile to compile code

Should know how to declare rules in makefiles

Should know how to call certain rules, and what just calling “make” does

- Example of calling certain rule: make clean will call the clean rule specifically



Generics/Templates

Know the definition of a generic and a template (generics in c++) (check definitions)

Create a function that uses templates:

```
#include<iostream>
```

```
class Cat{  
public:  
int num=3;
```

```
Cat operator+(Cat c) //overloaded operator  
{  
Cat c1;  
int value;
```

```

    value=this->num+c.num;
    c1.num=value;
    return c1;
}

};

```

//Template function. We can pass in different types for the function (instead of specifying it for one type)
 //Think of the T as a placeholder meaning any type (int or string for example)

```

template <class T>
T plus_sign(T n1, T n2) //takes two parameters of type T and returns something of type T (some type)
{
    T ret;

    ret = n1 + n2;

    return ret;
}

```

```

int main()
{
    int num1=100,num2=200,num3;
    long long1=13,long2=21,long3;
    std::string s1="hello", s2=" world!", s3;
    Cat c1, c2, new_cat; //each cat start with the num 3

```

//Notice we can use the template function with any of the types:

```

num3 = plus_sign(num1,num2);
std::cout<<"\nThe sum of integer values : "<<num3<<std::endl;

```

```

long3 = plus_sign(long1,long2);
std::cout<<"\nThe sum of long values : "<<long3<<std::endl;

```

```

s3=plus_sign(s1,s2);
std::cout<<"\nTwo strings : "<<s3<<std::endl;

```

```

new_cat=plus_sign(c1, c2);

```

std::cout<<"New value in c1: "<<new_cat.num<<std::endl; //remember all Cat objects start with the num 3
 and we defined + to be an overloaded operator that adds two Cat objects together (meaning adding the nums
 together), so it is 6

```

}

```

Create a class that uses templates:

```

#include <iostream>
using namespace std;

```

```

template <class T>
class Temp_example {
    T e1, e2; //type not specified
public:
    Temp_example (T first, T second) //type not specified
    {
        e1=first;
        e2=second;
    }

    //template <class T>
    T larger ()
    {
        T retval;
        retval = e1>e2? e1 : e2; //is e1 greater than e2 (T/F)? Return e1 if true, e2 if false
        return retval;
    }
};

//syntax to define function outside of class:
// template <class T>
// T Temp_example<T>::larger ()
// {
//     T retval;
//     retval = a>b? a : b;
//     return retval;
// }

int main () {
    Temp_example <int> example1 (100, 75); //100 is larger

    Temp_example <char> example2('A','a'); //the letter a is larger (check ASCII-97), A is less (check ASCII-55)

    cout << example1.larger()<<endl;
    cout<< example2.larger()<<endl;
    return 0;
}

```

Vocabulary List (This can change)

- **Object-Oriented Programming (OOP)** – A style of programming focused on the use of classes and class hierarchies. (The “PIE” model of OOP is based on 3 fundamental concepts:)
- **Encapsulation** – Bundling data and code into a restricted container
- **Inheritance** – Reuse and extension of fields and method implementations from another class
- **Polymorphism** – The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods to generate different results

- **Abstraction** – Specifying a general interface while hiding implementation details
- **Class** – A template encapsulating data and code that manipulates it
- **Class Hierarchy** – Defines the inheritance relationships between a set of classes
- **Class Library** – A collection of classes designed to be used together efficiently
- **Base Class** – The class from which members are inherited
- **Derived Class** – The class inheriting members
- **Override** – A derived class replacing its base class' implementation of a method
- **Multiple Inheritance** – A derived class inheriting class members from two or more base classes
- **Primitive type** – A data type that can typically be handled directly by the underlying hardware
- **Method** – A function that manipulates data in a class
- **Getter** - A method that returns the value of a private variable
- **Setter** - A method that changes the value of a private variable
- **Instance** – An encapsulated bundle of data and code
- **Object** – An instance of a class containing a set of encapsulated data and associated methods
- **Variable** – A block of memory associated with a symbolic name that contains an object or a primitive data value
- **Operator** – A short string representing a mathematical, logical, or machine control action
- **Operator Overloading** – Providing a user-defined meaning to a pre-defined operator (e.g., +, ==, <<) for a user-defined type (e.g., a class)
- **Constructor** - A special class member that creates and initializes an object from the class
- **Destructor** – A special class member that cleans up when an object is deleted
- **Friend** – A function that is granted access to its friend class' private members
- **Unified Modeling Language (UML)** - The standard visual modelling language used to describe, specify, design, and document the structure and behavior of software systems, particularly OO
- **Algorithm** – A procedure for solving a specific problem, expressed in terms of an ordered set of actions to execute
- **Software Design Pattern** - A general reusable algorithm solving a common problem
- **Anti-Pattern** – A common error that occurs in software systems and processes
- **Exception** – An object created to represent an error or other unusual occurrence and then propagated via special mechanisms until caught by special handling code
- **Declaration** – A statement that introduces a name with an associated type into a scope
- **Definition** – A declaration that (also) fully specifies the entity declared
- **Namespace** – A named scope
- **Data Validation** - Ensuring that a program operates on clean, correct and useful data
- **Validation Rules** – Algorithmically enforceable constraints on the correctness, meaningfulness, and security of input data
- **Stack** – Scratch memory for a thread
- **Heap** – Memory shared by all threads for dynamic allocation
- **Template** – A C++ construct representing a function or class in terms of generic types
- **Generic Programming** – Writing algorithms in terms of types that are specified as parameters during instantiation or invocation
- **Standard Template Library** – A library of well-implemented algorithms focused on organizing code and data as C++ templates
- **Iterator** – A standard library abstraction for objects referring to elements of a container
- **Concurrency** – Performing 2 or more algorithms (as it were) simultaneously
- **Process** – A self-contained execution environment including its own memory space
- **Thread** – An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space
- **Reentrant** – An algorithm that can be paused while executing, and then safely executed by a different thread
- **Mutex** – A mutual exclusion object that prevents two properly written threads from concurrently accessing a critical resource

