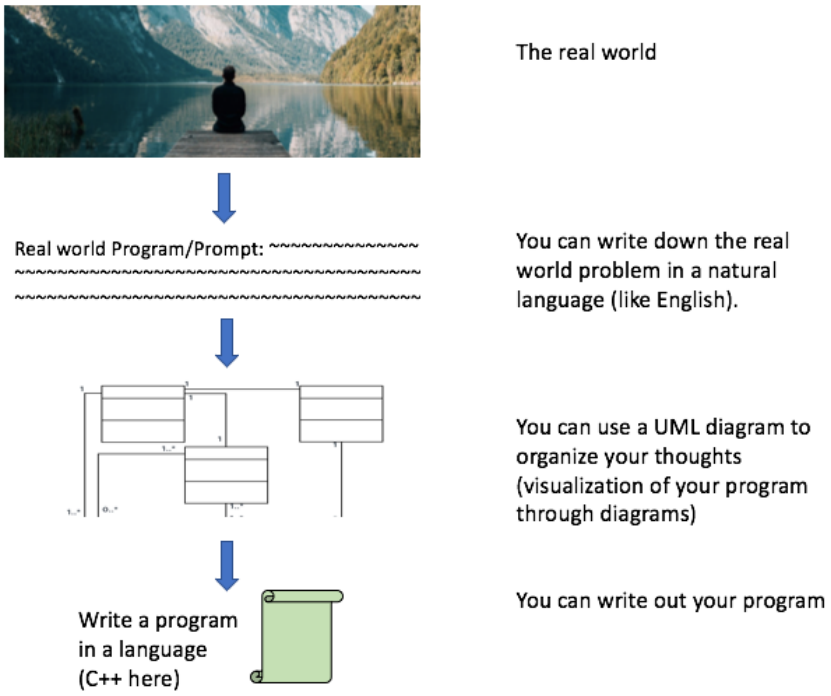
Additional Topics I (class/activity diagrams, enums, makefiles)

Class/Activity Diagrams (Two types of UML diagrams)

Note: in future classes (outside of 1325) you will learn about these more in depth-DO NOT LEAVE THIS CLASS THINKING YOU KNOW ALL ABOUT UML DIAGRAMS. (I will do an example in the last program of both)

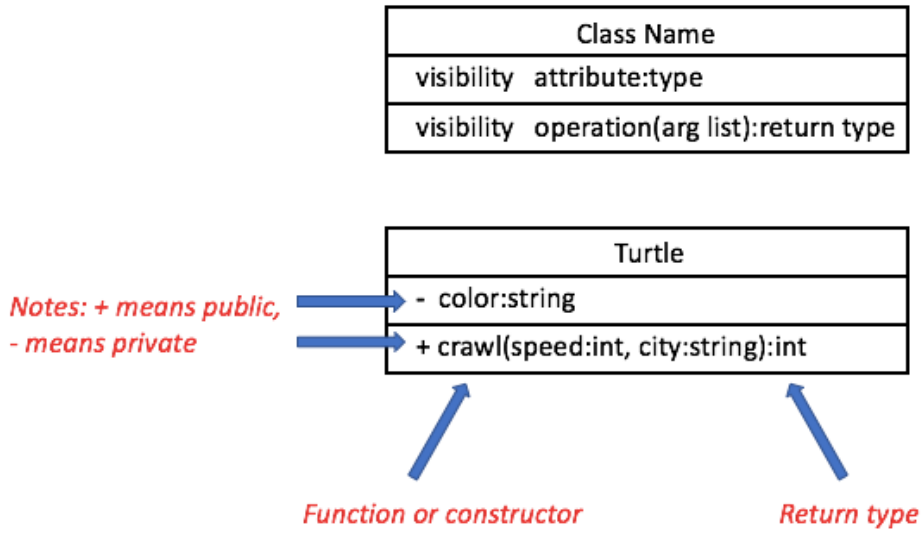
I like to think of it as a visualization (in diagrams) of your problem you are trying to solve and turn into code:



For this semester, we will only be doing **class diagrams and activity diagrams**.

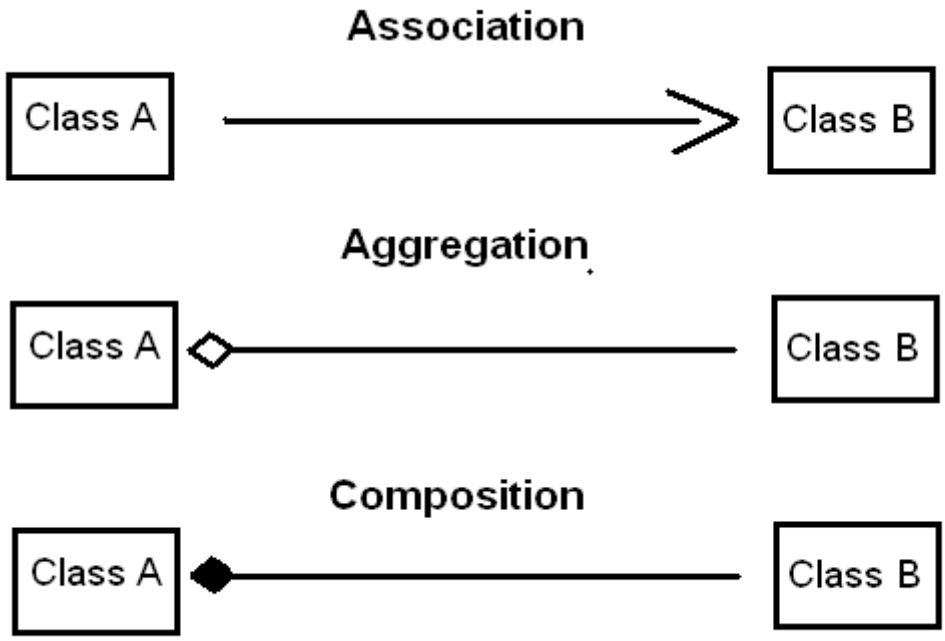
Class diagrams:

These show the classes (and relationships between classes) in our programs:



Note: # means protected

Relationships between classes (association, composition, aggregation):



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 (using an instance of B for example), 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)

NOTE: There is A LOT more detail about class diagrams that we will not cover in 1325

Activity diagrams:

These show the actual flow of your program (see sample programs-I will be drawing these on the board).

Different types of UML diagrams (website shown in class):
https://creately.com/blog/diagrams/uml-diagram-types-examples/ (you can also google UML types)

Websites with good explanations on relationships between classes (you can look up more):
http://aviadezra.blogspot.com/2009/05/uml-association-aggregation-composition.html
https://www.youtube.com/watch?v=UI6lqHOVHic

You can create UML diagrams using free options online (there are many different options). The one I used in class today:
http://www.umlet.com/umletino/umletino.html
NOTE: Virtual Machine has a program called Umbrello

T***HE FOLLOWING ARE JUST TO HELP YOU GET STARTED-I'M NOT SAYING YOU MUST USE THIS SPECIFIC WEBSITE TO MAKE THEM (YOU CAN USE WHATEVER PROGRAM/WEBSITE YOU FIND HELPFUL):***

Creating Activity diagrams:
https://www.lucidchart.com/pages/uml-activity-diagram?a=0

Creating Class diagrams:
https://www.lucidchart.com/pages/uml-class-diagram?a=0

(Show examples)

Enums/enum classes:

Enums:

A user defined type. I find them the most useful when I want to let a value have a meaning in the real world (note that there are other cases where you might find them useful):

```
computer$ g++ -std=c++11 practice.cpp
computer$ ./a.out
It is Monday or Sunday :(
```

```
#include <iostream>


using namespace std;
enum Feelings{Happy, Sad}; /*0 means Happy, 1 means Sad.  Note: I could also do something like:
enum Feelings{Happy=11, Sad=12}; where 11 means Happy and 12 means Sad*/

int get_feelings(string day)
{
 if(day=="Monday"|| "Sunday")
 {
     return 1;
     }

     return 0;
    }

    int main(int args, char ** argv)
    {
     int f=get_feelings("Monday");

     if(f==Happy) /*I am using the integer value returned interchangeably with the word I defined it to
    mean (instead of checking if f==0)*/
     {
          cout<<"It's not Monday or Sunday!"<<endl;
     }

     else
     {
         cout<<"It is Monday or Sunday :("<<endl;
     }

    }
```

Enum classes:

Notice I get the following warning when I compile the next code:

```
computer$ g++ practice.cpp
practice.cpp:21:11: warning: comparison of two values with different enumeration
      types ('Animal' and 'Building') [-Wenum-compare]
    if (a1== s1) // The compiler will as integers (see warning)
          ~~^  ~~
1 warning generated.
```

```
Computers-MacBook-Air:C++ computer$ ./a.out //we can still run the program
Equal!
```

```
#include <iostream>

enum Animal
{
 CAT,
 DOG
};

enum Building
{
 SCHOOL,
 HOUSE
};


int main(int argc, char **argv)
{
   Animal a1 = CAT;
   Building s1 = SCHOOL;

   if (a1== s1) // The compiler will compare as integers even though animals and buildings are not the same (see warning)
 {
      std::cout << "Equal!\n";
 }
   else
 {
      std::cout << "Not equal!\n";
 }

   return 0;
}
```

I can use an enum class instead (example below):

```
#include <iostream>

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

using namespace std;

int main(int argc, char **argv)
{
   Color color = red;
   Color color1=Color::red;
```

```
   int num = color;

   //Animal a1=cat; cant do this
   Animal a = Animal::cat;
}
```

Notice by using enum classes, we are not allowed to compare diff enum types:

```
computer$ g++ -std=c++11 practice.cpp
practice.cpp:21:11: warning: comparison of two values with different enumeration
      types ('Animal' and 'Building') [-Wenum-compare]
    if (a1== s1) // The compiler will as integers (see warning)
         ~~^  ~~
practice.cpp:21:11: error: invalid operands to binary expression
      ('Animal' and 'Building')
    if (a1== s1) // The compiler will as integers (see warning)
         ~~^  ~~
1 warning and 1 error generated.
```

```
#include <iostream>

enum class Animal
{
 CAT,
 DOG
};

enum class Building
{
 SCHOOL,
 HOUSE
};


int main(int argc, char **argv)
{
   Animal a1 = Animal::CAT;
   Building s1 = Building::SCHOOL;

   if (a1== s1) // Won't work
 {
      std::cout << "Equal!\n";
 }
   else
 {
      std::cout << "Not equal!\n";
 }

   return 0;
}
```

Makefiles (simple examples):

I will go over the general idea of a makefile. **Note that the examples I am giving are very SIMPLE examples.**

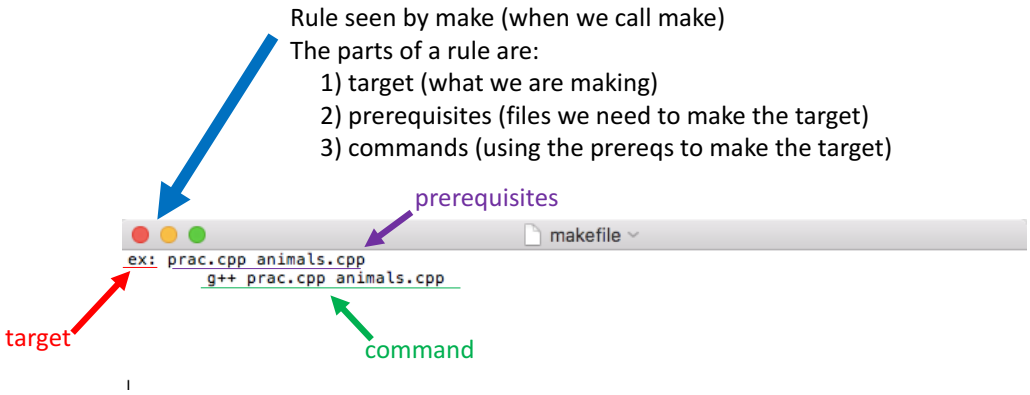
Per the course objectives, in 1325 you should be able to:
-make a makefile to compile code
-declare rules in makefiles
-call certain rules
-know what just calling "make" does

1) *make* is a command (called a utility). Just like *ls* is a command (to list files in a directory) we can use the *make* command to automate processes (like the process of putting together all pieces of an executable program together).

2) When we call *make* (saying we want to execute the *make* command), *make* knows what we want it do by reading the *makefile* that we create.

3) A *makefile* is a special file that contains commands of we want to do. It contains *rules* (it can also include other items) that we create (basically all the steps we want to automate). The basic structure of a rule in a *makefile* is:

Target: prerequisites
<TAB> Commands (recipes)



We can compile the files like we did in the previous class (no makefile):

```
computer$ g++ -std=c++11 main.cpp mouse.cpp
```

or we could create a makefile (# used for comments):

makefile

```
ex:    #TARGET
        g++ -std=c++11 main.cpp mouse.cpp  #COMMAND
```

When I run the makefile by typing make, it looks like this (the command called is shown):

```
mouse computer$ make
g++ -std=c++11 main.cpp mouse.cpp  #COMMAND
mouse computer$ ./a.out
Is this a house or school?
```

If I don't want the command to show, I could add @ at the beginning of my command:

makefile — Edited

```
ex:    #TARGET
        @g++ -std=c++11 main.cpp mouse.cpp  #COMMAND
```

Notice now the command doesn't show up when I type in make:

```
mouse computer$ make
Computers-MacBook-Air:mouse computer$ ./a.out
Is this a house or school?
```

We could also break up the compilation process by directly creating object files and then putting them together to make the executable (no makefile):

```
computer$ g++ -std=c++11 -c mouse.cpp  //making an object file (mouse.o)
computer$ ls
main.cpp  mouse.cpp  mouse.h mouse.o //we see mouse.o
computer$ g++ -std=c++11 -c main.cpp //making an object file (main.o)
computer$ ls
main.cpp  main.o mouse.cpp mouse.h mouse.o //we see main.o
computer$ g++ -std=c++11 -o main1 main.o mouse.o //make an executable
from the object files (main1)
computer$ ls
main.cpp  main.o main1 mouse.cpp mouse.h mouse.o //we see main1
computer$ ./main1 //we can now run main1
```

Above, I created two object files (mouse.o and main.o). I then put them together into an executable called main1 (I could have also just used the default name a.out instead of giving it the name main1).

A *makefile* for the preceding would look like: (notice it's pretty much the same as the steps I took manually, just combined into a single file)

makefile — Edited

```
executable: main.o mouse.o   #TARGET, PREREQs (notice here the prereqs are targets made below)
	g++ -std=c++11 -o main1 main.o mouse.o #COMMAND

main.o: main.cpp #TARGET, PREREQs
	g++ -std=c++11 -c main.cpp #COMMAND

mouse.o: mouse.cpp #TARGET, PREREQs
	g++ -std=c++11 -c mouse.cpp #COMMAND
```

```
computer$ make
g++ -std=c++11 -c main.cpp #COMMAND
g++ -std=c++11 -c mouse.cpp #COMMAND
g++ -std=c++11 -o main1 main.o mouse.o #COMMAND
```

Remember you could put @ before each command so it doesn't show up:

makefile

```
executable: main.o mouse.o   #TARGET, PREREQs (notice here the prereqs are targets made below)
	@g++ -std=c++11 -o main1 main.o mouse.o #COMMAND

main.o: main.cpp #TARGET, PREREQs
	@g++ -std=c++11 -c main.cpp #COMMAND

mouse.o: mouse.cpp #TARGET, PREREQs
	@g++ -std=c++11 -c mouse.cpp #COMMAND
```

```
computer$ make
Computers-MacBook-Air:mouse computer$ ls
a.out      main.cpp   main.o          main1      makefile   mouse.cpp
     mouse.h          mouse.o
```

Above, notice when I type *ls*, the object files I created show up (mouse.o and main.o).

Sometimes we just want to execute a process and not make a target file. These are called *phony targets*. An example is when I want to "clean up" my files created (my object files and the executable). I can make a rule to do this.

Notice I am typing *make* followed by a specific rule (make cleanup). In this case, only the rule *cleanup* is called (I normally call this rule *clean*). *Note: echo is a command to output to screen*

makefile

```
executable: main.o mouse.o   #TARGET, PREREQs (notice here the prereqs are targets made
below)
	@g++ -std=c++11 -o main1 main.o mouse.o #COMMAND

main.o: main.cpp #TARGET, PREREQs
	@g++ -std=c++11 -c main.cpp #COMMAND

mouse.o: mouse.cpp #TARGET, PREREQs
	@g++ -std=c++11 -c mouse.cpp #COMMAND

cleanup:
	echo "Cleaning up object files and executable!" #output to screen
	rm *.o
	rm main1
```

```
computer$ make
Computers-MacBook-Air:mouse computer$ ls
a.out      main.cpp   main.o          main1      makefile   mouse.cpp
     mouse.h          mouse.o
```

```
computer$ make cleanup
echo "Cleaning up object files and executable!" #output to screen
Cleaning up object files and executable!
rm *.o
rm main1
computer$ ls
a.out      main.cpp   makefile   mouse.cpp  mouse.h
```

mouse.h

```
#include <iostream>

enum class smart_mouse{
 SMART, NOT_SMART
};

class Cheese{
 std::string type;

public:
  Cheese(std::string s);

};

class Mouse{
 smart_mouse intelligence;

public:
 Mouse();
};

class Building{

 std::shared_ptr<Cheese> c;
 std::string location;

public:

  Building (std::string s, std::string location );

  virtual void building_statement();

};

class House:public Building{

public:
  using Building::Building;
  virtual void building_statement();
```

```
};

class School:public Building{

public:
  using Building::Building;
  virtual void building_statement();
};

namespace example{

 Building* choose_building();

}
```

mouse.cpp

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

 Cheese::Cheese(std::string s)
 {
    type=s;
 }

 Mouse::Mouse()
 {
   std::string answer;
   std::cout<<"Is this mouse smart?"<<std::endl;
   std::cin>>answer;

  if(answer=="yes")
  {
     intelligence=smart_mouse::SMART;
   }

  else
  {
     intelligence=smart_mouse::NOT_SMART;
   }
 }

 Building::Building (std::string s, std::string location )
 {
  c=std::make_shared<Cheese>(s);
  this->location=location;
 }

 void Building::building_statement()
```

```
 {
  std::cout<<"Building!"<<std::endl;
 }

 void House::building_statement()
 {
  std::cout<<"House!"<<std::endl;
 }

 void School::building_statement()
 {
  std::cout<<"School!"<<std::endl;
 }


namespace example
{

 Building* choose_building()
 {
   std::string answer;

   std::cout<<"Is this a house or school?"<<std::endl; //assume user always enters one or the other
   std::cin>>answer;

   if(answer=="house")
   {
          return new House("House", "Dallas");
   }

  else
  {
          return new School("School","Dallas");
  }
 }
}
```

main.cpp

```
#include "mouse.h"


int main(int argc, char **argv)
{
 Building* b1=example::choose_building();

 b1->building_statement();

 delete(b1);
```

}

Note: If you get the following error, make sure that your *makefile* does not end with .txt AND that you are trying to compile from the same directory with your *makefile*:

```
computer$ make
make: *** No targets specified and no makefile found.  Stop.
```

Program 1:

(In this program, I am trying to use a little of most of the topics we have covered. I won't focus on going over every single line of code in class, just the ideas. I will also draw activity and class diagrams.)

ABC Food stand sells pretzels and salads. Create a program that allows customers to buy from the food stand (there are two cooks in the food stand). The program should allow a customer to keep purchasing until they indicate they are done and then print out all items purchased by the customer. When the program exits, a list of all customers that visited the food stand should be output to screen.

makefile

```
ex: main.o food_stand.o
	@echo "Making food stand program..."
	@g++ -std=c++11 -o main1 main.o food_stand.o

main.o: main.cpp food_stand.h
	@g++ -std=c++11 -c main.cpp

food_stand.o: food_stand.cpp food_stand.h
	@g++ -std=c++11 -c food_stand.cpp

clean:
	@echo "Cleaning up object files and executable!" #output to screen
	@rm *.o
	@rm main1
	@echo "All files cleaned up :)" #output to screen
```

food_stand.h

```
#ifndef FOOD_STAND_H
#define FOOD_STAND_H
#include <string>
#include <vector>
#include <memory>

class Food{

protected:
  std::string type;
  double price;

public:
  explicit Food(std::string type,
  std::string get_type();
  double get_price();

};


class Salad:public Food{

  using Food::Food;//inheriting
};

class Pretzel:public Food{

 using Food::Food; //inheriting
};
```

food_stand.cpp

```
#include <string>
#include <iostream>
#include <vector>
#include "food_stand.h"

Food::Food(std::string type, double price)
 {
   this->price=price;
   this->type=type;
```

main1

```
#include "food_stand.h"
#include <iostream>
#include <string>
#include <vector>
#include <memory>

using namespace std;

int main()
{
  Food_stand f1(1,2, "Cook1", "Cook2");
  string answer="start";

  cout<<"********Food Stand**********"<<endl;
  while(answer!="exit")
  {
    cout<<"\n--Enter customer name:"<<endl;
    cin>>answer;
```

(Sample run- using makefile)

```
computer$ make
Making food stand program...
computer$ make clean
Cleaning up object files and executable!
All files cleaned up :)
```

(Sample run-not using makefile)

```
computer$ g++ -std=c++14 -o foodie main1.cpp food_stand.cpp
computer$ ./foodie
********Food Stand**********


--Enter customer name:
Hansel


Would you like to buy something? We have pretzels and salads.
yes
What would you like to buy?
pretzel
Making pretzel...


Would you like to buy something? We have pretzels and salads.
yes
What would you like to buy?
salad
Making salad...


Would you like to buy something? We have pretzels and salads.
no
Thanks!


***All food purchased by Hansel:
Pretzel for: $2.99
Salad for: $5.99


--Enter customer name:
Gretel


Would you like to buy something? We have pretzels and salads.
yes
What would you like to buy?
salad
Making salad...


Would you like to buy something? We have pretzels and salads.
yes
What would you like to buy?
pretzel
Making pretzel...


Would you like to buy something? We have pretzels and salads.
yes
What would you like to buy?
pretzel
Making pretzel...


Would you like to buy something? We have pretzels and salads.
no
Thanks!


***All food purchased by Gretel:
```

```
Salad for: $5.99
Pretzel for: $2.99
Pretzel for: $2.99

--Enter customer name:
exit
Exiting...

All customers today:
Hansel
Gretel

***Bye!***
```

(show class/activity diagram)

//in my .h file, I give classes, function prototypes, class member variables. I will actually define my functions in the food_stand.cpp file (next file)

food_stand.h

```
#ifndef FOOD_STAND_H  //don't forget your header guards
#define FOOD_STAND_H
#include <string>
#include <vector>
#include <memory>

class Food{ //Food class

protected:
  std::string type;
  double price;

public:
 Food(std::string type, double price);
  std::string get_type();
  double get_price();

};


class Salad:public Food{

  using Food::Food; //using constructor from base class
};

class Pretzel:public Food{

 using Food::Food; //using constructor from base class
};
```

```
class Person{

protected:
  std::string name;

public:
  Person(std::string name);
  std::string get_name();
};


class Employee:public Person{
  int id_num;

public:
  Employee(int id, std::string name);

};

class Cook:public Employee{
  int num_items_made=0;

public:
  using Employee::Employee;
  std::shared_ptr<Salad> operator << (double price); //overloading << operator (definition in .cpp file)
  std::shared_ptr<Pretzel> operator >> (double price); //overloading >> operator (definition in .cpp file)

  int get_num_items();
};

class Customer:public Person{

using Person::Person;

public:
  std::vector<std::shared_ptr<Food>> all_food;
  void buy_food(Cook &c);
  void operator ! (); //overloading ! operator (definition in .cpp file)
};


class Food_stand{

public:
  std::vector<std::shared_ptr<Customer>> all_customers;
  std::shared_ptr<Cook> c1; //making two smart pointers to Cook objects for the Food_stand class
  std::shared_ptr<Cook> c2;

  Food_stand(int id1, int id2, std::string name1, std::string name2);
```

```
  void checkout_customer(std::shared_ptr<Customer> c);
  void close_stand();
};

#endif


//here we have definitions for our functions
food_stand.cpp

#include <string>
#include <iostream>
#include <vector>
#include "food_stand.h"

//remember the scope resolution operator-when we say something like Food::Food, we are saying that
Food (the constructor-right hand side) belongs to Food the class (left hand side)
 Food::Food(std::string type, double price)
  {
    this->price=price;
    this->type=type;
  }

  //here, we saying that get_type() is a function that belongs to the Food class (using the scope
resolution operator)
  std::string Food::get_type()
  {
    return type;
  }

  double Food::get_price()
  {
    return price;
  }



  Person::Person(std::string name)
  {
    this->name=name;
  }

  std::string Person::get_name()
  {
    return name;
  }


  Employee::Employee(int id, std::string name):Person(name)
  {
```

```
  id_num=id;
}


std::shared_ptr<Salad> Cook::operator << (double price)
{
 num_items_made++;
 std::cout<<"Making salad..."<<std::endl;
 std::shared_ptr<Salad> ptr=std::make_shared<Salad>("Salad", price);
 return ptr;
}

std::shared_ptr<Pretzel> Cook::operator >> (double price)
{
 num_items_made++;
 std::cout<<"Making pretzel..."<<std::endl;
 std::shared_ptr<Pretzel> ptr=std::make_shared<Pretzel>("Pretzel", price);
 return ptr;
}

int Cook::get_num_items()
{
 return num_items_made;
}



void Customer::buy_food(Cook &c)
{
 std::string answer;
 std::cout<<"What would you like to buy?"<<std::endl;
 std::cin>>answer;
 if(answer=="pretzel")
 {
  all_food.push_back(c>>(2.99)); //add pretzel-use overloaded operator
 }

 else //salad
 {
  std::shared_ptr<Food> s1=c<<(5.99); //add salad-use overloaded operator
  all_food.push_back(s1);
 }
}

void Customer::operator ! () //show all food
{
 if(all_food.size()!=0)
 {
  std::cout<<"\n***All food purchased by "+name+":"<<std::endl;
  for(int i=0;i<all_food.size();i++)
```

```
    {
      std::cout<<all_food[i]->get_type()<<" for: $"<<all_food[i]->get_price()<<std::endl;
    }
   }

 }


 Food_stand::Food_stand(int id1, int id2, std::string name1, std::string name2)
 {
  c1=std::make_shared<Cook>(id1, name1);
  c2=std::make_shared<Cook>(id2, name2);
 }

 void Food_stand::checkout_customer(std::shared_ptr<Customer> c)
 {
  std::cout<<"Thanks!"<<std::endl;
  all_customers.push_back(c);
 }

 void Food_stand::close_stand()
 {
  std::cout<<"\nAll customers today:"<<std::endl;
  for(int i=0;i<all_customers.size();i++)
  {
    std::cout<<all_customers[i]->get_name()<<std::endl;
  }

  std::cout<<"\n***Bye!***\n"<<std::endl;
 }
```

main1.cpp

```
#include "food_stand.h"
#include <iostream>
#include <string>
#include <vector>
#include <memory>

using namespace std;

int main(int agrc, char **argv)
{
 Food_stand f1(1,2, "Cook1", "Cook2");
 string answer="start";

 cout<<"********Food Stand**********"<<endl;
 while(answer!="exit")
 {
```

```
  cout<<"\n--Enter customer name:"<<endl;
  cin>>answer;

  if(answer=="exit")
  {
   cout<<"Exiting..."<<endl;
  }

  else
  {
   shared_ptr<Customer> cust1=make_shared<Customer>(answer);

   while(answer!="no") //keep buying food
   {
    cout<<"\nWould you like to buy something? We have pretzels and salads."<<endl;
    cin>>answer;
    if(answer=="yes")
    {
     cust1->buy_food(*f1.c1); //access the cook pointer (c1) in the food stand and deref it
    }
   }
   f1.checkout_customer(cust1);
   !(*cust1); //print out customer list using overloaded operator
  }

 }

 f1.close_stand();

}
```