WEEK 6

Classes and Resources, I/O Operators

AGENDA

- Week 6-1
  - Classes with resource / Resource pointers
  - Deep vs Shallow copy / assignment
  - Copy constructor / Copy assignment operator
- Week 6-2
  - Standard I/O operators
  - File I/O operators

WEEK 6-1

Resources, Shallow vs Deep Copy, Copy Constructor/Assignment

CLASSES WITH RESOURCES

- Up to this point we've worked with classes with fundamental types as their data members as well as used statically allocated arrays
- Both those kinds of resources have known amounts of memory to be allocated for at compile time
  - In the case of fundamental types, based on those types is the amount of memory
  - In the case of static arrays it is specified how large they are via usually a constant

CLASSES WITH RESOURCES

- How about if we needed the use of dynamic allocation for a class' data members? If we didn't know how much memory we needed till run time?
- We have worked with these kinds of data members as well.
- From here on out we'll begin to call class with these kinds of members Classes with Resources (dynamic resources)

CLASSES WITH RESOURCES

- Recall that with dynamic memory allocation we make use of a pointer syntax and the new/delete keywords to do so.
- Eg. `playdoh* p = new playdoh[3];`

CLASSES WITH RESOURCES

- When we are working with dynamic resources in our class' data members it will be using those same keywords and syntax
- We call these pointers within a class/object a resource instance pointer
- These pointers hold the address to the resources that are outside our object's static memory

CLASS WITH RESOURCES

- Let's consider our previous playdoh class which had colour and weight data members. So far it doesn't have any dynamic resources attributed to it. Let's create a new class that will use our playdoh class and have an array of playdoh that is allocated dynamically.

```
// playdoh
class playdoh{

       char colour;
       int weight;
....
}
```

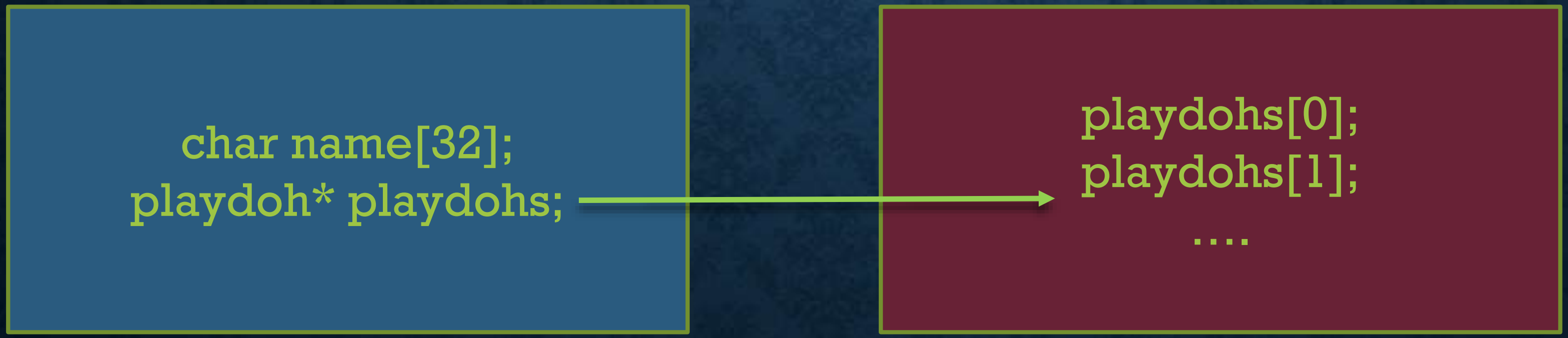
```
// playdohpack
class playdohpack{

       char name[32];
       playdoh * playdohs;
....
}
```

CLASSES WITH RESOURCES

Effectively the arrangement of the static and dynamic resources in our playdohpack instance would look like the below. The name which is statically allocated is within the static memory whereas the dynamically allocated playdohs are in a different memory space.

playdohpack p;



PLAYDOHPACK EXAMPLE

DEALLOCATION

- One minor note about deallocation of dynamic memory:
- The delete keyword has no effect on nullptr's
- Thus when the destructor is called on a object in an empty state it won't have any effect
  - Ie it won't attempt to delete / deallocate memory that wasn't ever initialized or is otherwise pointing to nullptr

SHALLOW COPY

- A shallow copy is a copy that does not do the independent resource allocation for dynamic resources
- Shallow copying should only be applied to non dynamic resources
  - In essence shallow copying is the simple use of the assignment operator (=)
    - Eg. P1.weight = P2.weight;

SHALLOW COPY

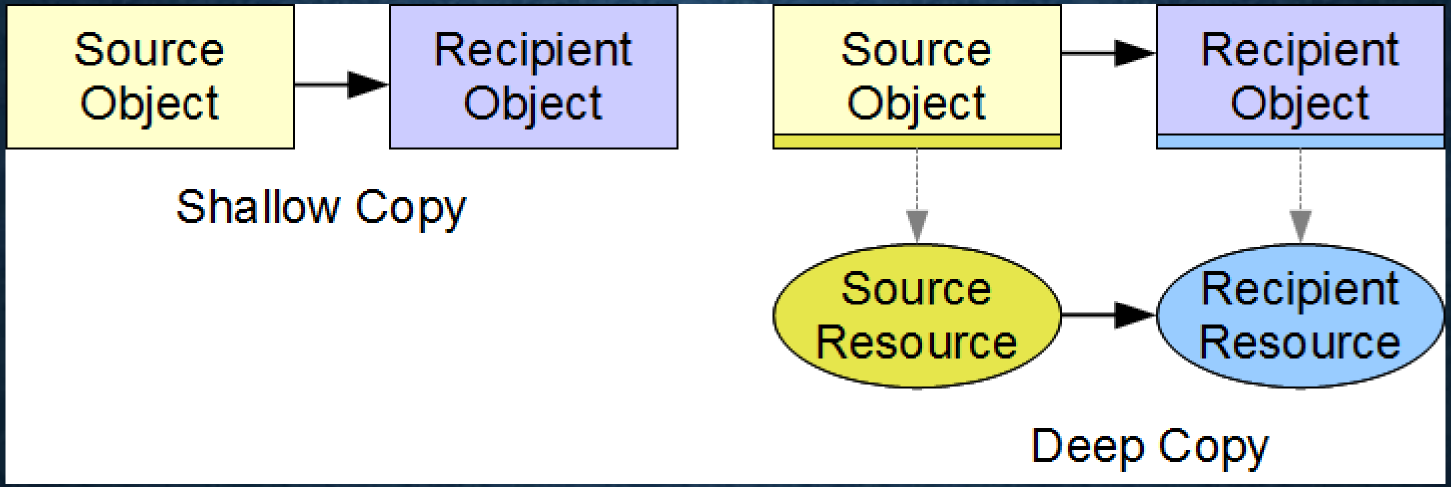
- Shallow copying has one main problem if applied to classes with resources. We will end up with two objects with the exact same dynamic resources (ie affecting object a's dynamic resource also affects object a_copy's resource)

SHALLOW COPY EXAMPLE

DEEP COPIES AND ASSIGNMENT

- When dealing with classes with resources, much like with regular non dynamically allocated data members there is an expectation that the data stored is specific to an instance of the class
- In other words the resources of an object while perhaps holding the same values should be independent of other resources (changing the resources in one object won't have any effect on another)
- When we are talking about copying classes with resources we have to keep the above in mind.

DEEP VS SHALLOW



DEEP COPYING AND ASSIGNMENT

- To facilitate the copying and assignment of classes with resources, we have a couple of special member functions to help out:
  - Copy Constructors
  - Copy Assignment Operators
- Similar to the default constructor, if we don't implement these two special functions the compiler will insert empty ones for you.
- These empty ones will only do shallow copying

COPY CONSTRUCTOR

- The copy constr is a constructor that contains logic for copying from a source object to a newly created object of the same type
- The copy constr is called in the following contexts:
  1. Creates an object by initializing it to an existing object
  2. Copies an object by value in a function call
  3. Returns an object by value from a function

- Declaration example:
  - Type(const Type&)
  - PlaydohPack(const PlaydohPack&)

```cpp
// Copy Constr
PlaydohPack::PlaydohPack(const PlaydohPack& src){

  // Shallow copy non dynamic resources
  strncpy(name, src.name, max_name);
  playdohCount = src.playdohCount;

  // Allocate memory for the playdohs
  if (src.playdohs != nullptr){
    playdohs = new playdoh[playdohCount];
    // copy data from the source resource
    // to the newly allocated source
    for (int i = 0; i < playdohCount; i++){
      playdohs[i] = src.playdohs[i];
    }
  }
  else{
    playdohs = nullptr; // set it to null if we can't copy
  }
}
```

An example of when the copy constr is called:

PlaydohPack p1;
PlaydohPack p2 = p1;

Important steps:

1. Shallow copy
2. Allocate memory for deep copy
3. Deep copy
4. Set to nullptr

COPY ASSIGNMENT

- The copy assignment operator contains logic for copying data from an existing object to an existing object
- The compiler calls this function when it sees client code like:
  - identifier = identifier;
  - p1 = p2;
- Declaration example:
  - Type& operator=(const Type&)
  - PlaydohPack& operator=(const PlaydohPack&)

```cpp
// Assignment operator
PlaydohPack& PlaydohPack::operator=(const PlaydohPack& src){
  // Check for self assignment
  if (this != &src) {
    // shallow copy non dynamic variables
    playdohCount = src.playdohCount;
    strncpy(name, src.name, max_name);

    // deallocate previous allocated memory
    delete [] playdohs;
    // allocate new memory if needed
    if (src.playdohs != nullptr){
      playdohs = new playdoh[playdohCount];
     // deep copy
     for (int i = 0; i < playdohCount; i++){
       playdohs[i] = src.playdohs[i];
     }
    }
    else {
      playdohs = nullptr; // set to null
    }
  }
  return *this;
}
```

Examples of when the assignment operator is called:

PlaydohPack p1;
PlaydohPack p2;
p1 = p2;

Important steps:

1. Check for self assignment
2. Shallow copy
3. Deallocate previous memory
4. Allocate new memory
5. Deep copy
6. Set to nullptr
7. Return the current object

COPY CONSTR & ASSIGNMENT

- At this point we may notice that the copy constr and the assignment operator, the logic present is very similar.
- We could make some adjustments to it so that we reduce the redundancy of the code
- There are two approaches we could take here:
  - A private member function that handles the duplicate copying code
  - A direct call of the assignment operator in the copy constr

DELETE COPY CONSTR & ASSIGNMENT

- If we want to disallow the copying of objects via the copy constr and assignment operator (perhaps we ant to have any objects created through other functions) we can do so with the delete keyword.
- In the declaration of those functions append a = delete:
  - PlaydohPack& operator=(const PlaydohPack&) = delete;
  - PlaydohPack(const PlaydohPack&) = delete;
- These are referred to as deleted functions
- Deleted functions can't be defined/given implementation nor called in client code

WEEK 6-2

I/O Operators

I/O OPERATORS

- The library in which we get the main I/O functionality of C++ is the <iostream> library
- The iostream library's definitions are also contained in the std namespace
- In that library we have the cin and cout objects representing the standard input and output
- The cin object is of type ostream and cin is of type istream

I/O OPERATORS

- To review there are two operators we typically use in reference to I/O operations (ie with the cin & cout objects).

- These are the extraction and insertion operators:
  - << insertion operator – inserts into a output stream
  - >> extraction operator – extracts from the input stream

I/O OPERATORS

- I/O operators are in much like the operators in the previous week, things we can overload to depict new behavior or meaning with our user defined / custom types
- The I/O operators are helper operators

I/O OPERATORS

- For example, so far we've seen display functions called in perhaps this manner:
  - P1.display();
  - Internally this makes use of a line in the shape of: cout << info << endl;

I/O OPERATORS

- What if we wanted to have a similar ability to output our playdoh object straight into the output stream object cout:
  - cout << p1 << endl;
  - How would we do this? Well we'd overload the operator and perhaps also create some member functions to allow the operator to do its work

I/O **OPERATORS**

- Consider first however an updated display function:

Look at this param

```
// Updated display function
void playdoh::display(std::ostream& os) const{

    os << "Playdoh, colour: " << colour << " weight " << weight << std::endl;
}
```

I/O OPERATORS

- The form of the operator overload for the << and >> operators to work with our playdoh will look like the following:

```
std::istream& operator>>(std::istream&, Type&);
std::ostream& operator<<(std::ostream&, const Type&);
```

```
std::istream& operator>>(std::istream&, Playdoh&);
std::ostream& operator<<(std::ostream&, const Playdoh&);
```

Playdoh that can work with << and >> operators

I/O OPERATORS

- Notice the types of the return values as well as the first parameter.
- Recall that in this case the first parameter is also the left operand of the operator
- So if the left operand is an ostream object for the << operator and we know that cout is an instance of an ostream class… using this operator may look like:

```
cout << p1 << endl;
```

I/O OPERATORS

- The overloaded operator then makes use of our previously updated display function

```
// << op overload
std::ostream& operator<<(std::ostream& os, const playdoh& p){

  p.display(os);
  return os;
}
```

- Using this helper operator we're now able to do things like: cout << p1;

I/O OPERATORS

- Let's also consider the case for using operators for input ie:
  `cin >> p1;`
- Similar to the overloading the << operator we will need to make use of a public member function to allow for the extraction of data from the standard input to into a playdoh object. Perhaps a 'read' function that reads from the user.

```cpp
// New read function
void playdoh::read(std::istream& is) {

    char col;
    int wei;

    std::cout << "Enter the colour of the playdoh" << std::endl;
    is >> col;
    std::cout << "Enter the weight of the playdoh" << std::endl;
    is >> wei;

    playdoh temp(col, wei);
    if (temp.weight != 0) // If valid playdoh set as it the current obj
        *this = temp;
    else
        *this = playdoh(); // Else set it as an empty playdoh

}
```

I/O OPERATORS

Our >> operator overload may now look like this:

```
// >> op overload
std::istream& operator>>(std::istream& is,  playdoh& p){
    p.read(is);
    return is;
}
```

Using this helper operator we're now able to do things like: cin >> p1;

I/O OPERATORS

- A question may have popped in your minds: Why do these operator overloads and their respective public member functions that they rely on return a reference to either istream or ostream type?

I/O OPERATORS

- Let’s consider the following:
  - `cout << p1 << p2 << p3;`
  - This could be broken down into:
    - `cout << p1;`
    - `cout << p2;`
    - `cout << p3;`

I/O OPERATORS

This particular behavior is called cascading. And to enable it we need to design our I/O operator overloads in this way.

- cout << p1 << p2 << p3;

This portion is essentially one call of our operator overload: cout << p1;

The whole of cout << p1 << p2 could be written like this:

(cout << p1) << p2.

The return of cout << p1 is a reference to the ostream object cout.

So if [cout << p1] = cout

The rest of the line then becomes [cout] << p2;

FILE OPERATORS

- To do actions that related to input and output of files in C++ we make use of the <fstream> library. Notice that it is somewhat similarly named to the <iostream> library and to little surprise they work similarly as well.

- Like the iostream library which uses objects cin and cout to represent input and output, the fstream library also uses objects to represent files and interact with them in similar ways using the << and >> operators

FILE OPERATORS

- Notable functions with file objects:
  - open(); - This function is used to open a connection to a file. Ie it creates a link between the file object and the file it is meant to represent in our code
  - bool is_open(); - This function checks if our connection to the file was successful. If unsuccessful it could mean that the file we're trying to connect to doesn't exist or has other issues. It returns a Boolean value.

FILE OPERATORS

- Let's look at creating a input file object (input as we we're reading from the file)

```
#include <fstream> // library for doing file
related things

int main(){

std::ifstream fin;
fin.open("mytxt.txt");
//std::ifstream fin("mytxt.txt");
return 0;
}
```

Create the object then open a file with it, thus creating the connection to the existing file. Notice the ifstream type – input file stream

Alternative method using a constructor

- Let’s then trying reading from the file with a loop:

```
std::ifstream fin;
fin.open("mytxt.txt");

int x = 0;

if (fin.is_open()){

  while(fin){
    fin >> x;
    if (fin)
      std::cout << "X is: " << x << std::endl;
    else
      std::cout << "End of file" << std::endl;
    }
  }
else{
  std::cout << "Bad file, can't open" << std::endl;
}
```

Open file

Check if opening file was successful

Check if the file is not at the end of file

Read from file Looks like working with cin

- Here is an example of working with an output file:

```
/ file-output.cpp
//

#include <iostream>
#include <fstream> // library for doing file
related things

int main(){

std::ofstream fin;
fin.open("out.txt");

if (fin.is_open()){

fin << "Line 1" << std::endl;
fin << "Line 2" << std::endl;
fin << "Line 3" << std::endl;
}
return 0;
}
```

Notice the ofstream type – output file stream

Looks like we're working with cout