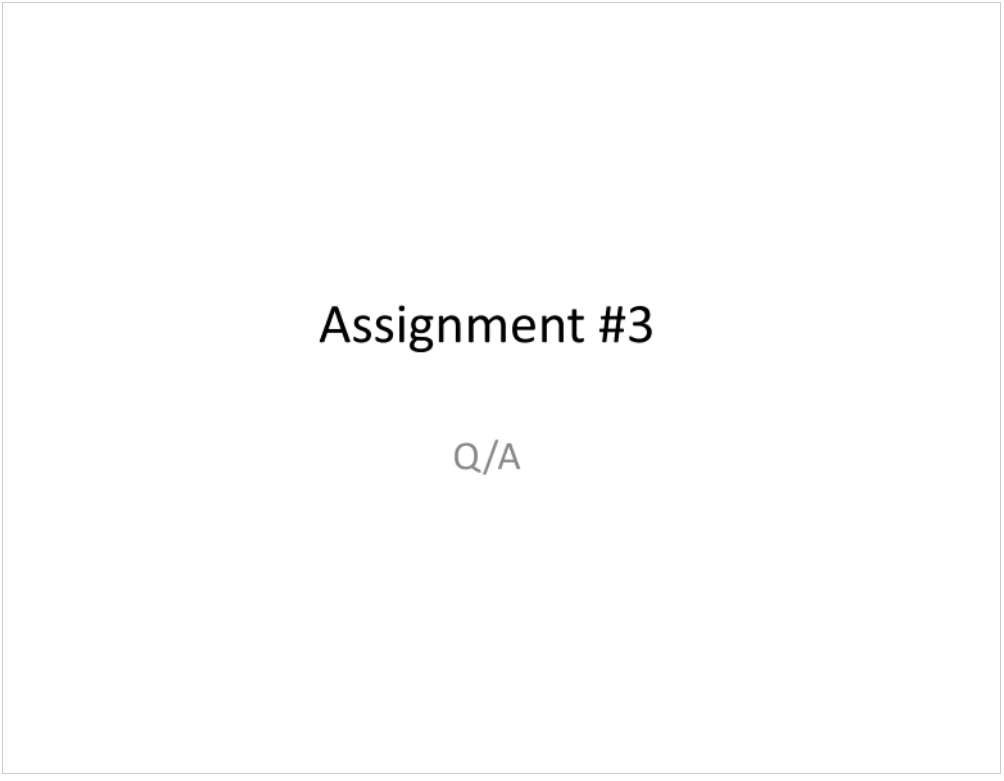
Lecture_15 (3)

Monday, March 28, 2022 9:36 AM



Hints & Tips

- Make sure you run your code – you never know what will happen…
- Try invalid input – make sure you are handling situations like division AND modulus by 0.
- Try unusual combinations of operators and check that your operator precedence (that works).
  - (2 + 4) * 5 / 6
- Use hierarchical Inheritance in your code:
  - Command -> Binary Operator Command -> Add Command

Hints & Tips

```
try
{
     num = std::stoi(token);
}
catch (std::invalid_argument)
{
// Error converting token to a number.
Invalid input.
     return nullptr;
}
```

Assignment #3

- Reminder(s):
  - Test your code on Thomas – please run your code!
    - Remember you are creating the driver this time, so there is no excuse if it doesn't run properly!
  - Use your Valgrind report as a guide – if there is a memory leak attempt to solve the problem.
    - Focus just on the leaks – ignore the errors in the report.
  - Test for both valid and invalid input – make sure you have accounted for common exceptions.
    - Error - Invalid Input
    - Error – Division By 0

Abstract Factory Pattern

Abstract Factory Pattern

- Example:
  - https://sourcemaking.com/design_patterns/abstract_factory/cpp/before-after
- Abstract Factory
  - Concrete Factories
- Products
  - Concrete Products
- Driver

Abstract Factory Pattern

- Let's first start with the Product here:

```
class Widget
{
    public: virtual void draw() = 0;
};
```

- Recall that our Product should be abstract – here we aren't focused on the implementation. That will go in the "concrete" Product.

Abstract Factory Pattern

- Let's now create our Product families – here we are creating "products" for Linux based environments:

```
class LinuxButton : public Widget
{
	public: void draw()
	{
		std::cout << "Linux Button\n";
	}
};

class LinuxMenu : public Widget
{
	public: void draw()
	{
		std::cout << "Linux Menu\n";
	}
};
```

Abstract Factory Pattern

- Let's now create a second Product family – here we are creating "products" for Windows based environments:

```
class WindowsButton : public Widget
{
	public: void draw()
	{
		std::cout << "Windows Button\n";
	}
};

class WindowsMenu : public Widget
{
	public: void draw()
	{
		std::cout << "Windows Menu\n";
	}
};
```

Abstract Factory Pattern

- Now we need to create our Abstract Factory class – here again, we need to make sure this is an interface.
  - Pure Virtual – No Implementation

```
class Factory {
      public:
            virtual Widget * create_button() = 0;
            virtual Widget * create_menu() = 0;
};
```

Abstract Factory Pattern

- Now are ready to create our concrete Factories.

```
class LinuxFactory : public Factory
{
      public:
            Widget * create_button()
            {
                  return new LinuxButton;
            }

            Widget * create_menu()
            {
                  return new LinuxMenu;
            }
};
```

Abstract Factory Pattern

- …and our other concrete Factory:

```
class WindowsFactory : public Factory
{
      public:
            Widget * create_button()
            {
                  return new WindowsButton;
            }

            Widget * create_menu()
            {
                  return new WindowsMenu;
            }
};
```

Abstract Factory Pattern

```
private:
  Factory *factory;

 public:
  Client(Factory *f) {
    factory = f;
  }

  void draw() {
    Widget * w = factory->create_button();
    w->draw();
    display_window_one();
    display_window_two();
  }
```

```
void display_window_one() {
   Widget * w[] = {
       factory->create_button(),
       factory->create_menu()
   };
   w[0]->draw();
   w[1]->draw();
 }

 void display_window_two() {
   Widget * w[] = {
       factory->create_menu(),
       factory->create_button()
   };
   w[0]->draw();
   w[1]->draw();
 }
```

Abstract Factory Pattern

- Now that we have our Client created let's create a small Driver to run our code:

```
int main()
{
	Factory * factory;

	#ifdef LINUX
	factory = new LinuxFactory;

	#else // WINDOWS
	factory = new WindowsFactory;
	#endif

	Client * c = new Client(factory);
	c->draw();
}
```

Composite Pattern

Composite Pattern

- Have you ever been in the following situation:
  - Have a LOT of objects and need to show their hierarchical relations?
  - Have a LOT of objects and objects composed of many other objects, but really do not care about their implementation?
- Analogy:
  - A building typically has many floors and these floors have many rooms.
    - Calculate the area of each room, combine the area of all rooms on each floor, then combine area of each floor.

- We can treat our expr_cmd as a node in the expression tree
- Keep a running total (maintain State while traversing)

Composite Pattern

- Pattern Classification:
  - Structural Pattern
- Problem:
  - We need a way to treat "things" differently even if they appear to be used in the same way.
- Solution:
  - Compose objects into tree structures to represent the part-whole hierarchies

What is a composite pattern?

↳ A way to compose objects into tree structures to represent the "PART-WHOLE" hierarchies

↳ Parent child relationship and trying to maintain it.

Composite Pattern

- Motivation:
  - Our current design for the expression evaluator uses a stack-based implementation that evaluates a postfix expression.
  - It is also possible to represent the expression as an expression tree to better show operator association with operands, and then use post order traversal to evaluate it.
    - How can we convert our calculator to handle such behavior?

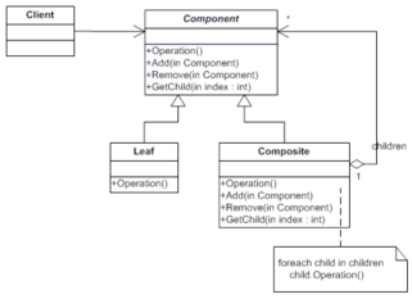
Composite Pattern

- Allows us to build structures of objects in the form of trees that contain both composition of objects and individual objects as nodes in the tree.

- The composite's role is to define behavior of the components having children and to store child components.

Composite Pattern

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and composition of objects uniformly.



- We should have 1 generic calculator function, recursive definition, that can be used at all levels

Composite Pattern

- From our previous design, we know that each entity in the expression can be represented as a command object.
  - These command objects can be represented as a tree-based hierarchy.
    - Command -> Unary/Binary/Number -> Add/Subtract/etc.
- How can we convert this to a represent the same concepts in an expression tree format?

Composite Pattern

- How can we represent the following in a tree-based structure?
  - -5 * (3 + 4)
- What are the different components of this expression?
- How does the role of the Composite pattern fit here?

Composite Pattern