Week 4

Relationships & Expressions

Agenda

- Week 4-1 – Relationships
  - Composition, Aggregation, Association
- Week 4-2 - Expressions

Week 4-1

Relationships Between Classes

- Relationships between classes in an Object Oriented context can exhibit different degrees of ownership
- Besides the relationships shown through inheritance and parametric polymorphism there are three particular relationships that occur often in OOP:
  - Composition (Strongest)
  - Aggregation
  - Association (Weakest)
- These three forms will often shown their faces when working with classes with resources

Composition

- Composition is described as the strongest of these three relationships
- It refers to the relationship where a class (the composer) is composed in part of other classes and that the other classes are effectively owned by the composing class – in other words the those component classes are contained within the composer and the composer will determine the lifetime of the componets
- It can be described as an "has a" sort of the relationship.
- In terms of classes with resources if a class is responsible for copying and destroying its resource then the relationship is composition

Composition

Subobject

```
Class PlaydohPack {
        Playdoh playdohs;
        …
}
```

Composing class

Pointer

```
Class PlaydohPack {
        Playdoh* playdohs;
        …
}
```

```
Class Playdoh {
        char* colour;
        int weight;
        …
}
```

Component class

The pointer variant requires the presence of copy mechanisms and a destructor

Composition

- When the component classes are updated the composer isn't affected
- However in order for the composer to access the composed classes, the latter will need to offer some public members to do so

Aggregation

- An aggregation is a like a composition but it doesn't manage the creation of the destruction of the class/objects that it uses thus it is like a weaker version of composition in terms of ownership
- The class which is the aggregator is then considered to be complete with or without the class/objects it uses – compared to composition where these are needed to be complete
- The responsibility for that creation and destruction will instead be left on the class that is being aggregated – in turn that means the destruction of the aggregating class will not affected the aggregated class and vice versa

Aggregation

Subobject

```
Class PlaydohPackAgg {
        Playdoh& playdoh;
        …
}
```

Aggregating class

```
Class Playdoh {
        char* colour;
        int weight;
        …
}
```

Aggregate class

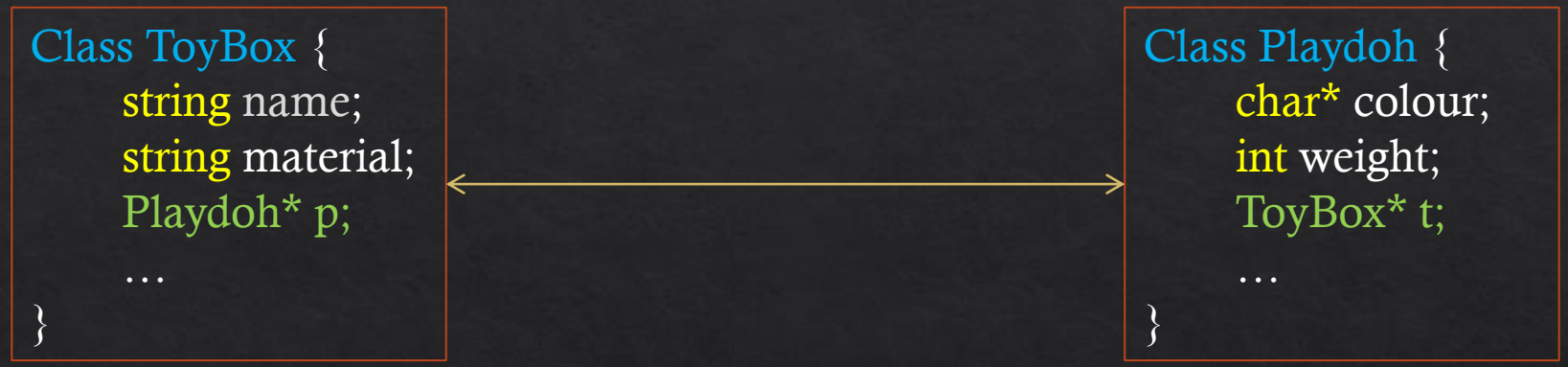
Pointer

```
Class PlaydohPackAgg {
        const Playdoh* playdohs[size];
        …
}
```

Association

- Lastly association is the weakest of the three relationships and thus also the most loosely coupled
- Unlike both aggregation and composition, there isn't a 'has a' relationship in play, classes that are associated with each other can be fully complete with or without each other – in other words they are fully independent classes

Association



Week 4-2

Expressions

- Expressions describe computations and they consist of sequences of operators and operands to possibly produce some result
- In C++ an expression has a non reference type and belongs to a value category
- Expressions can be primary or compound where the latter is a combination of primary expressions connected by operators
- The order in which an expression is evaluated is dependent on an order of precedence

Expressions

- A primary expression can be one of the following:
  - A literal (a number – 100, a floating point number – 1.234, a string – “abc”, boolean – true/false)
  - A name (variable, object)
- Some compound expressions:
  - A = 1 + 2;
  - var++;
  - D = A + B – C

Value Categories

- Any C++ expression is in one of three value categories:
  - Prvalue – value that does not occupy a location in storage
  - Xvalue – an expiring value that does occupy a location in storage (an object near the end of its lifetime)
  - Lvalue – a locator value that occupies allocation in storage

Value Categories

- P(ure) rvalue
  - Literals – 42, true, nullptr
  - Arithmetic expressions – a + b, a % b
  - Comparison expressions – a < b, a == b, a >= b
  - Logical expressions – a && b, a || b, !a
  - …

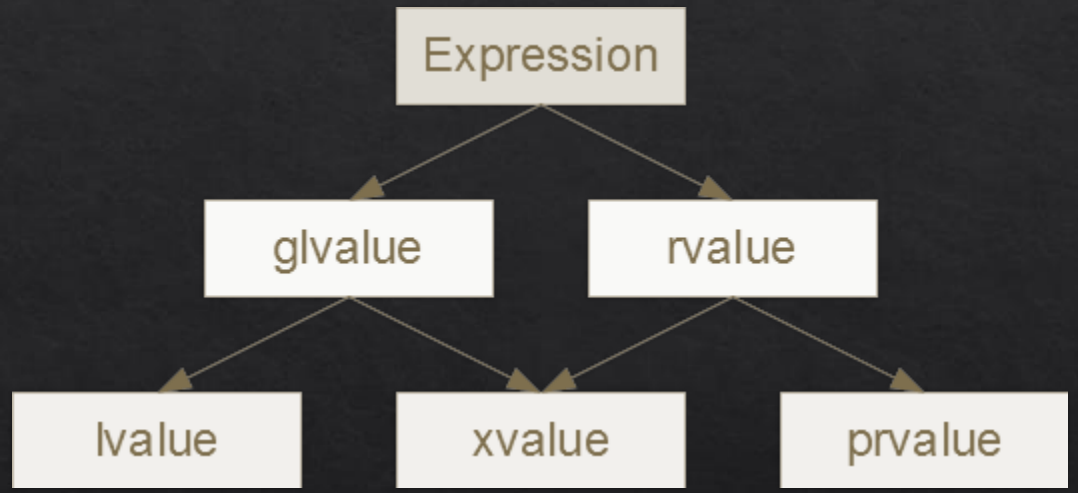
Value Categories

- Xvalue
  - Functions/expressions that return an rvalue reference
    - std::move(a)
    - static_cast<char&&>(a)
  - Any expression that designates a temporary object

Value Categories

- Lvalue
  - The name of a variable, function, or data member
  - Expressions that return an Lvalue reference

Value Categories



Operator Precedence / Order of Evaluation

- Expressions are divided up into six operand aligned categories:
  - Primary, Postfix, Prefix, Unary, Binary, Ternary
- While Primary expressions are simply just a single operand without any operator, the other five categories denote different kinds of operators where the category implies how they're used
- A expression can again be formed from other expressions to be a compound one
- When determining how to evaluate an expression to its result, the order evaluation is determined by what level of precedence a given expression (or part of one in the case of a compound expression) has
- The higher the precedence the earlier that expression will be evaluated

Order of Evaluation

```
struct Box {
  int weight{  };
  int items[5]{1, 2, 3, 4, 5};

    ...

  void display() const {
    cout << "Box Weight: " << weight<< endl;

    for (int i = 0; i < 5; ++i) {
      int temp = *this.items[i];
      cout << i + 1 << ": " << temp << endl;
    }
  }
};
```

Operator Associativity

- If two expressions make use of operators of the same precedence how is it determined which expression is evaluated first?
- For example consider some arithmetic:
  - 7 – 5 + 4
  - Is this (7 – 5) + 4 = 6 OR is it 7 – (5 + 4) = -2
- There where associativity comes into play, where certain operators associate from left to right or right to left (ie we start reading from which side)

Lvalue Operands

- Some operators require its (left) operand to be a Lvalue, these include:
  - & (address of), ++ (increment), -- (decrement)
  - =, +=, -=, *=, /=, %= (various assignment forms)
- As such some examples of Lvalue operands are:
  - A name that isn't an array name
  - An array element – a[i]
  - Expressions that result in Lvalues
  - Dereferenced addresses
  - A string literal ("Hello World")

Postfix / Prefix Increment & Decrement

```cpp
#include <iostream>
int main() {
    int i = 10;

    std::cout << i++ << std::endl;
    (i++)++;            // ERROR (i++) is a prvalue, not an lvalue
    std::cout << i << std::endl;
}
```

```cpp
#include <iostream>
int main() {
    int i = 10;
    std::cout << ++i << std::endl;
    ++(++i);            // OK (++i) is an lvalue
    std::cout << i << std::endl;
}
```

sizeof(), sizeof

- The sizeof() and sizeof opearators are used to determine the size of the operands
- The former is used to evaluate the size of its operand from its type
  - Eg sizeof(int), sizeof(double)…
- The latter is used to evaluate the size of a variable, object or expression
  - Eg sizeof x, sizeof (x + y)…
- Both will return an unsigned integral type representing the size in bytes

decltype()

- The decltype operator evaluates to the type and value category of its argument
- The operator is used to declare a type based on a expression
- In a sense it's similar to the auto keyword that uses inference to determine the type of a variable
- Examples:

```
int x = 10;
double y = 5.5;
double& z  = y;
decltype(x) var1 = 1; // type is int
decltype(y) var2 = 2.2; // type is double
decltype(x + y) var3 = 3; // type is int
decltype(z) var4 = var2; // type is double&
```