Week 3

Inheritance & Templates

Agenda

- Week 3-1 – Inheritance
  - Concrete and Abstract Classes
  - Inclusion Polymorphism – Copying, Specializing, Type Discovery
  - Liksov Substitution Principle
- Week 3-2 - Templates

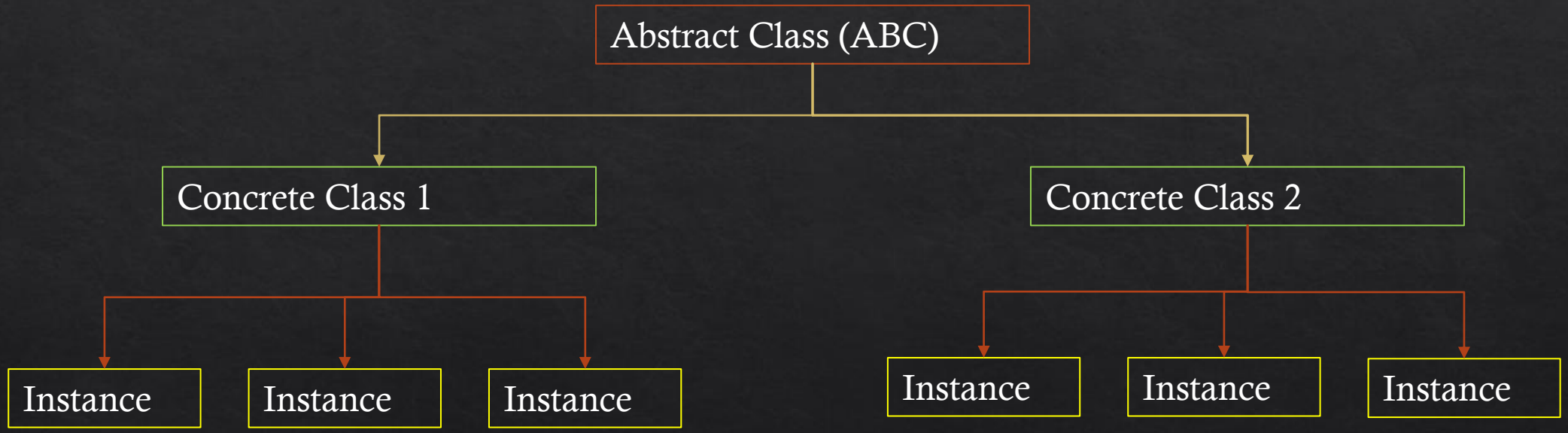
Week 3-1

Inheritance

- Inheritance is a relationship between user-defined types (classes) and from there we have two categories:
  - Concrete classes – Their representation is part of their definition and is known. These are effectively just normal classes.
  - Abstract classes (ABC) – Their representation is not part of their definition and is unknown. These are classes which have or inherit a pure virtual member function. Typically used as an interface to be derived from. The derived classed (a concrete class) will fill in the blanks of the implementation.

Inheritance

- A typical visual of inheritance with at work could be the following:



Compound ABC

Header guards to prevent double header inclusion. Can be replaced with #pragma once

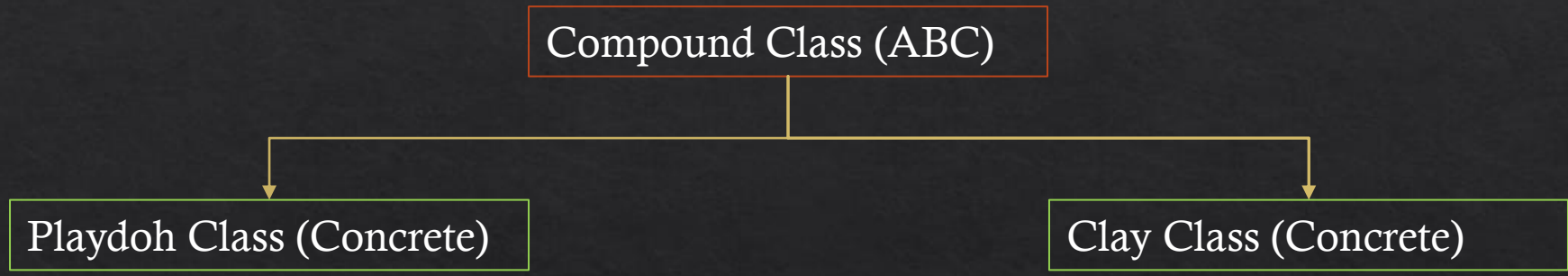
```
#ifndef COMPOUND_H
#define COMPOUND_H

#include <iostream>

class Compound {
public:
  virtual std::ostream& display(std::ostream& os) const = 0;
  virtual std::istream& write(std::istream& is) = 0;
  virtual int value() const = 0;
};
```

Pure Virtual Functions Signifies an ABC

Compound Hierarchy



Inclusion Polymorphism

- Objects are considered polymorphic if they can have different dynamic types throughout its lifetime. These are the types that are known during run time
- When dealing with such objects, in order to call the correct function associated with that object's dynamic type, there needs to be a mechanism to determine that type (dynamic dispatch via the virtual keyword)
- With that in mind there are three additional considerations that come up with dynamic types:
  - How to copy a polymorphic object to another polymorphic object
  - How to specialize an operation for a specific dynamic type
  - How to exclude a type from the default selection of a function

Copying Polymorphic Objects

- In order to copy an object we typically need to know the type of that object:

```
Compound* copy(const Compound& src){
  Compound* copy = new ???? (src)
  return copy;
}
```

Copying Polymorphic Objects

- To allow for this piece we can introduce a virtual clone function that will allow us to make copies of objects within this Compound hierarchy and we'll determine the types again through dynamic dispatch

```
virtual Compound* clone() const = 0;
```

Compound.h

```
Compound* clone() const;
```

Playdoh.h

```
Compound* Playdoh::clone() const {
  return new Playdoh(*this);
}
```

Playdoh.cpp

Specialize an Operation for Polymorphic Types

- If we want to have certain operators to work with polymorphic types then those same operators need to again know what those types are:

```
Compound *c = new Playdoh();
Compound* c2 = new Playdoh();
if (*c == *c2) …
```

???

Specialize an Operation for Polymorphic Types

- Let us then introduce a virtual operator== to our Compound hierarchy:

```
virtual bool operator==(Compound& c) const = 0
```

Compound.h

```
bool operator==(Compound& c) const
```

Playdoh.h

```
bool Playdoh::operator==(Compound& c) const {
  const Playdoh* play = dynamic_cast<Playdoh*>(&c);
  return play ? play->colour == colour : false;
}
```

Playdoh.cpp

Dynamic Type Identification

- C++ includes a <typeinfo> standard library that grants access to functions that can determine the type of objects during run-time.
- The main function of note is typeid() which will return objects of type_info type
- type_info objects contain some information about the type queried including a cstring description that is implementation dependent

Dynamic Type Identification

```
int i;
int * ptr;
cout << "i is: " << typeid(i).name() << endl;
cout << "ptr is: " << typeid(ptr).name() << endl;
```

Prints out int and int* respectively for each

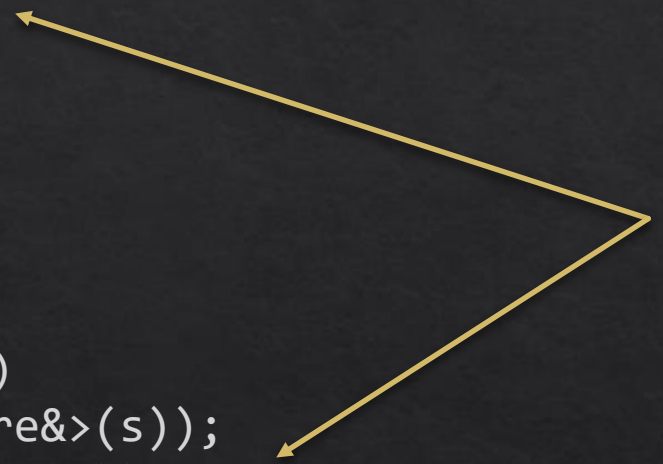
Liksov Substitution Principle

- The basis of this principle is effectively described as:
  - “a function that uses pointers or references to base classes must be able to use objects of derived classes without knowing it”
- More simply put if we were to exchange a derived class in place of a base in a place where we only make use of the base functions, there shouldn’t be any different or incorrect behavior
- Designing your types and their types based on this principle will lead to less faulty code

LSP Violation

```
class Shape{...} // Abstract
class Square : Shape{...}
class Circle : Shape{...}
```

```
void DrawShape(const Shape& s){
    if (typeid(s) == typeid(Square))
        DrawSquare(static_cast<Square&>(s));
    else if (typeid(s) == typeid(Circle))
        DrawCircle(static_cast<Circle&>(s));
}
```



For each new shape we'd need to update the drawShape function

Week 3-2

Templates

- Templates are the form of parametric polymorphism in C++. It allows for the reuse of a single structure or function to be used across a multitude of types in such a way that they all operate the same. In order words it allows for the use of generic types.
- Templates can be applied to both classes and functions alike. The compiler will take our templates and generate the classes and functions using the types we specify

Template Syntax

Template Function

```
template < template-parameter-list > // a function template header
 return-type function-name( ... ) { // a function template body
    // ...
 }
```

Template Class

```
 template < template-parameter-list > // a class template header
 class-key Class-name { // a class template body
    // ...
 };
```

Template Parameters

template < template-parameter-list >

The parameter list can be any combination of the following things:

- A type template parameter (typename / class keywords) – identifies a generic type
- A non-type template parameter (integral types, pointers, Lvalue…) – non generic types
- A template template parameter – another template (nesting)

Function Templates

```
template <typename T>
void temp_swap(T& a, T& b)
{
  T c;
  c = a;
  a = b;
  b = c;
}
```

```
int main(){

  int a = 1, b = 2;
  temp_swap(a,b);

  char x = 'A', y = 'B';
  temp_swap(a,b);
}
```

Compiler generates two temp_swap functions based on each of our uses of swap in the main function

The template parameter in the list can be used to serve as place holders till a type is specified

Specialization

- Template specialization allots for an exception to a template's definition
- This is useful for when we have a case that the general behavior of our templates does not cover well
- This can apply to both functions and classes

```
template <class T>
T maximum(T a, T b) {
    return a > b ? a : b;
}
```

Maximum will work differently when working with character pointers

```
template <> // denotes specialization
char* maximum<char*>(char* a, char* b) {
    return std::strcmp(a, b) > 0 ? a : b;
}
```

Class Templates

```
template <typename T, int N >
class Calculator {

  T result[N]{};
  …
}
```

Class templates are effectively the same as function templates except that the template header line affects the entire class scope

Note that the data member here is of the generic type T

N, here is a integral type and will be used like a constant and will be setting the size of the result array

Class Templates

```
template <typename T, int N >
class Calculator {

  T result[N]{};
  …
}
```

```
int main() {

  int iarr1[]{ 1,2,3 };
  int iarr2[]{ 3,4,5 };

  Calculator<int, 3> c1;
```

When we declare an instance of the templated class, the compiler will generate a class with the respective types. In this case we'll get a 'integer' Calculator with a result array size of 3

Template Default Values

```
template <typename T = int, int N = 3>
class Calculator {

  T result[N]{};
  …
}
```

```
int main() {

  int iarr1[]{ 1,2,3 };
  int iarr2[]{ 3,4,5 };

  Calculator<> c1;
```

Like with functions you can have default values for the template parameters. If you do then you can also declare an instance of that template class without any parameters

Variadic Templates

- Variadic Templates are templates that can accept an arbitrary number of arguments in its parameter list
- A variadic template has a parameter-pack as one of its parameters, there are two forms:
  - A template parameter pack
  - A function parameter pack
- The parameter-pack is denoted by a parameter name preceded by a set of ellipses (…):

```
template <typename T, typename... parameter-pack>
```

Template parameter pack

```
template <typename T, typename... parameter-pack>
void foo(TT... args) {  }
```

Function parameter pack

Variadic Templates

```
template <typename T>
void print(const T& t) {
    std::cout << t << std::endl;
}
```

```
template <typename T, typename... etc>
void print(const T& t,const etc&... pp) {
    std::cout << t << " | ";
    print(pp...);
}
```