WEEK 9

Templates, Type-Casting, Polymorphism

OVERVIEW

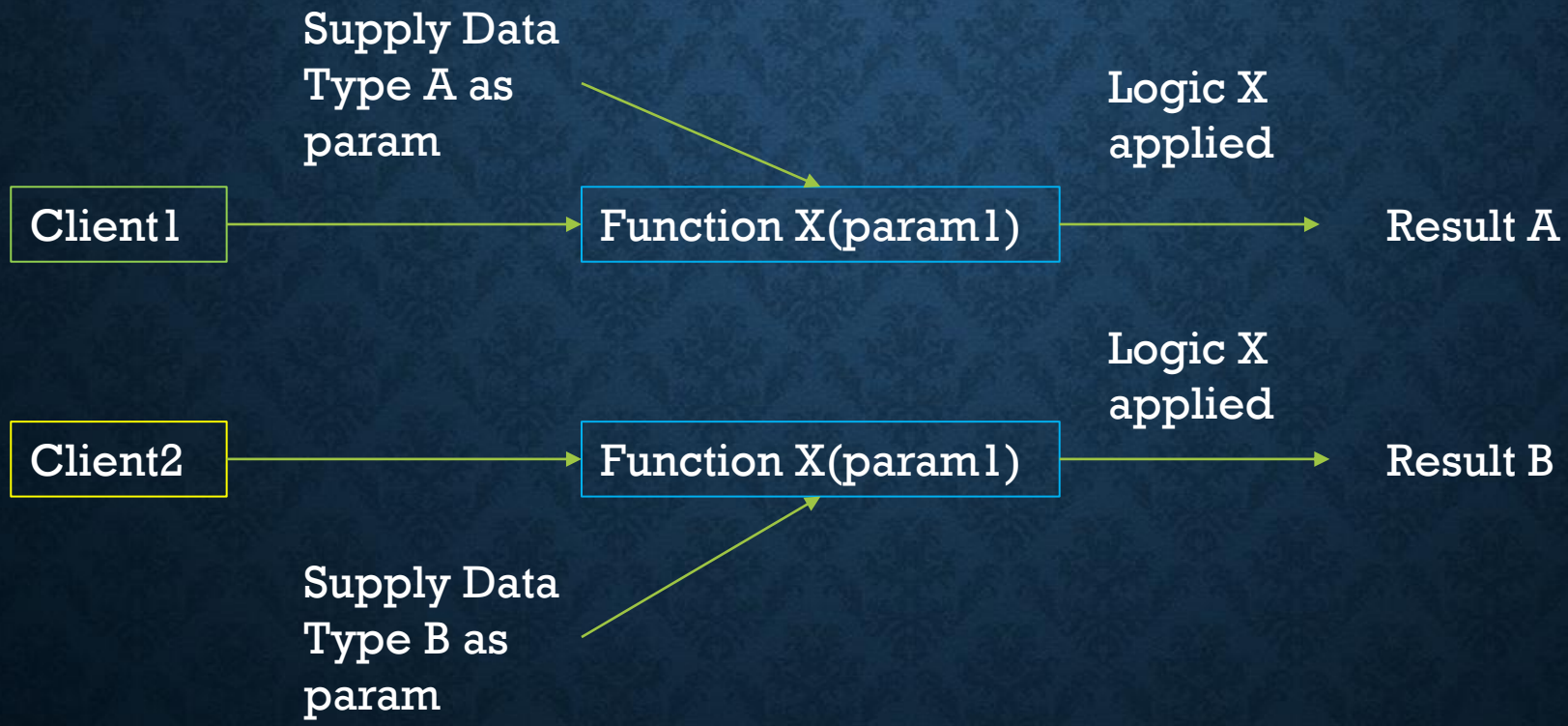
- Week 9-1
  - Templates - Functions, Classes
  - Type Casting – Dynamic Casts
- Week 9-2
  - Polymorphism Overview
    - AD-HOC - Coercion, Overloading
    - Universal - Inclusion, Parametric

WEEK 9-1

POLYMORPHISM

- As our coverage of Polymorphism continues, let's consider another form of it: Parametric Polymorphism
- Parametric Polymorphism refers to the context where a type and the logic executed on that type are independent of one another
- In other words, you can have a client access the same logic on unrelated / different types

POLYMORPHISM



POLYMORPHISM

- C++ offers a language feature to support this kind of situation where we may want to allow for a single function to be able to accept many different types as parameters but to have the function operate the exact same regardless of the supplied types.
- This feature is known as templates

TEMPLATE SYNTAX

- Keyword: **template**
- Definition:
  - template <Type identifier[,…]>
  - Type can be one of the following:
    - typename/**class** – to identify a type, these two keywords are mostly interchangeable
    - int, long, short, char – to identify a non floating-point fundamental type
    - A template parameter – for nesting templates

TEMPLATE SYNTAX

- Sample Usage:
  - template <typename T> - specifies template that uses type T
  - template <class T>
  - template <typename T, int N> - specifies a template that uses a type T and an integer N

TEMPLATES

- Templates can be applied to both functions and classes
- Templated functions or classes are also typically (recommended) placed in header files

FUNCTION TEMPLATE
SWAPPING FUNCTION EXAMPLE

Non-templated

```
// swap.cpp
void swap(int& a, int& b){

        int c;
        c = a;
        a = b;
        b = c;
}
```

templated

```
// swap.h
template<typename T>
void swap(T& a, T&b){

        T c;
        c = a;
        a = b;
        b = c;
}
```

```
#include <iostream>
#include "swap.h" // template definition
int main() {
    double a = 2.3;
    double b = 4.5;
    long   d = 78;
    long   e = 567;
    swap(a, b); // compiler generates
                // swap(double, double)
    std::cout << "Swapped values are " <<
        a << " and " << b << std::endl;
    swap(d, e); // compiler generates
                // swap(long, long)
    std::cout << "Swapped values are " <<
        d << " and " << e << std::endl;
 }
```

```
Swapped values are 4.5 and 2.3
```

```
Swapped values are 567 and 78
```

CLASS TEMPLATE (ARRAY EXAMPLE)

Template

```
// Template for Array Classes
// Array.h
template <class T, int N>
class Array {
        T a[N];
public:
        T& operator[](int i) { return a[i];}
}
```

Implementation

```
// Template.cpp
#include <iostream>
#include "Array.h"
int main() {
        Array<int, 5> a;
        for (int i = 0; i < 5; i++)
                a[i] = i * I;
                std::cout << a[i] << std::endl;
}
```

DO TEMPLATES NEED TO ALWAYS BE IN HEADER FILES???

You might be thinking what about the separation of the declarations and definitions (.h and .cpp) we've had so far.

TYPE CASTING

OVERVIEW

- C++ is a strongly typed language ie it heavily uses the type system to evaluate the legality of things to squash issues during compile time.
- Types can be implicitly converted to other types
  - Example: `int a = 3; double b = a;`
- Types can be explicitly converted (casting) to other types
  - Example: `double x = 10.3; int y; y = int(x); // y = 10`

OVERVIEW

- Type casting circumvents the type system's ability to check if types match (their legality). It is generally recommended to avoid casting unless it is unavoidable and keep them localized.
- If casting is required, make use of C++'s constrained casting functions that have some (limited) type checking.

CONSTRAINED CASTS

STATIC_CAST<TYPE>(EXPRESSION)

- Converts the expression from its evaluated type to the specified type. Works with related types (eg, int and double)
- Most common type of constrained cast
- Limited type checking
  - Rejects conversion between pointers and non-pointer types
- Does not perform runtime type checks

REINTERPRET_CAST<TYPE>(EXPRESSION)

- Converts the expression from its evaluated type to an unrelated type
- Platform dependent (eg. hardware)
- Limited type checking
  - Rejects conversions between related types
- Does not perform runtime type checks

CONST_CAST<TYPE>(EXPRESSION)

- Used mainly to manipulate the const status of an expression
- Limited type checking
  - Rejects conversions between different types
- Does not perform runtime type checks

DYNAMIC_CAST<TYPE>(EXPRESSION)

- Converts the value of the expression from its type to another type within the same class hierarchy
- Performs some type checking
  - Rejects conversions from a base class to a derived class if the object isn't polymorphic
- Can perform upcasts (derived => base) and downcasts (base => derived)

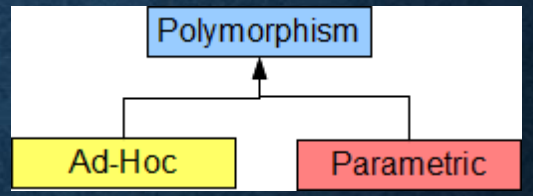
RECOMMENDED READINGS

- http://www.cplusplus.com/doc/tutorial/typecasting/
- https://en.cppreference.com/w/cpp/language/implicit_conversion
- https://en.cppreference.com/w/cpp/language/static_cast
- https://en.cppreference.com/w/cpp/language/reinterpret_cast
- https://en.cppreference.com/w/cpp/language/const_cast
- https://en.cppreference.com/w/cpp/language/dynamic_cast

WEEK 9-2

POLYMORPHISM CATEGORIES

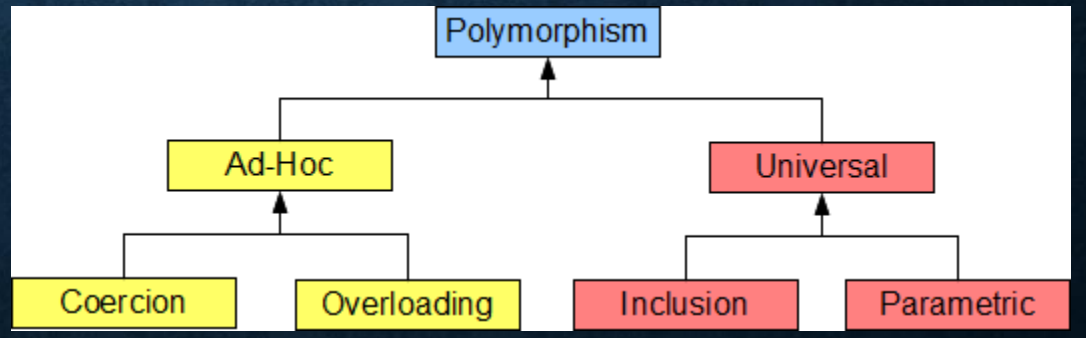
- Christopher Strachey (1967) introduced the concept of polymorphism into procedural programming languages by distinguishing functions in two ways:
- Works differently on different types
  - (AD-HOC)
- Works the same on different types
  - (Parametric)
- These two categories were then further broken down and refined into four categories by Cardelli & Wegner (1985)



POLYMORPHISM CATEGORIES (CONT'D)

- The four categories are aligned like so:
- AD-HOC (Works on a finite set of different & potentially unrelated types)
  - Coercion
  - Overloading

- UNIVERSAL (Works on a potentially infinite number of types across some common structure)
  - Inclusion
  - Parametric



AD-HOC - COERCION

- Coercion addresses the differences between argument types in a function call and the parameter types in the functions definition.
- It allows convertible type changes in the argument's type to match the parameter.
- It's an operation (implemented at compile time in C++, compiler inserts conversion code before function call) that allows for the avoidance of a type error.

AD-HOC - COERCION

- Two possible variations of coercion:
  - Narrow the argument type
  - Widen the argument type

COERCION EXAMPLE

```
//  coercion.cpp
#include <iostream>
void display(int a){ std::cout << "Arg is:" << a << std::endl; }
int main() {
        display(10); // Normal use
        display(12.6); // Narrows the argument from double to int – Prints 12
        display('A'); // Widens/Promotes the argument from char to int – Prints 65
}
```

AD-HOC - OVERLOADING

- Overloading addresses the variations of a function's signature.
- This feature allows for binding of function calls with the same identifier but different arguments with the matching definitions (IE function overloading).
- Overloading eliminates multiple function definition errors
- C++ implements overloading at compile type by renaming each identically named function with a distinct identifier.

UNIVERSAL - INCLUSION

- Inclusion allows for multiple member functions of a similar identifier by selecting the definition from a set of definitions based the object's type. Recall the idea of dynamic binding and the virtual keyword.
- The term inclusion refers to the base type including the derived types in a hierarchy and the ability to select the correct function for an object in that hierarchy based on function binding.

UNIVERSAL – PARAMETRIC (GENERIC)

- Parametric addresses the differences between argument types in the function call and the parameter types in the function's definition.
- This allows for function definitions that share identical logic independent of type.
- Recall that this is implemented in C++ at compile time through the use of templates.