Advanced Typing & Polymorphism

COSC1076
Semester 1 2019
Week 10

RMIT UNIVERSITY

Admin

- Assignment 2
  - Due Next Week, **Wednesday May 22**
  - Everything due - code, tests and report
- Quiz Feedback Out
- Assignment 1 marks, by tomorrow

Type Systems

RMIT UNIVERSITY

Types

- For a high-level language, the type system underpins the formal mathematics of how the language:
  - Is compiled
  - Is evaluated
- Effectively, everything is a type including:
  - Variables
  - Structs
  - Classes
  - Functions
  - Methods
  - Enumerations
- Strictly, namespaces are not a “type” but affect how a type is defined.

Strong vs Weak Typing

- In a ***Strongly*** typed language, the type of an entity cannot be changed, once the types of the entity has been instantiated. Languages include:
  - C++
  - Java
  - Python
- In a ***Weakly*** typed language, the type of an entity can be changed. Languages include:
  - Perl
  - Javascript

Static vs Dynamic Typing

- In a ***Statically*** typed language, the type of an entity is determined at compilation. Languages include:
  - C++ **
  - Java **
- In a ***Dynamically*** typed language, the type of an entity is determine at runtime. Languages include:
  - Python
  - Perl
  - Javascript
- ** Except for some forms of typecasting

Forms of Typing

C++	Strong	Static
Java	Strong	Static
Python	Strong	Dynamic
Perl	Weak	Dynamic
Javascript	Weak	Dynamic

Forms of Typing

- The rest of this discussion assumes:
  - Strong Typing
  - Static Typing
- If a language has different typing rules, the following concepts may change

Auto-type

RMIT UNIVERSITY

Auto-type

- C++11 introduced the `auto` keyword for when specifying the type of an entity (typically a variable/field)
- This keyword allows the compiler to automatically deduce the type of the entity using the entities initialisation

```
auto x = 1;
cout << x/2 << endl;
```

- Recall that the type is static and strong
  - The auto-type is determined at compile time
  - The type cannot be changed

Auto-type

```
auto x = 1;
cout << x/2 << endl;
```

- The auto keyword can create significant confusion:
  - It is not clear what the type of an entity may be - especially if the keyword is used prolifically
  - In general, the use of auto should be limited to where the automatically deduced type is not important to the static understanding of the code

Enumerations

RMIT UNIVERSITY

Enumeration

- An ***enumeration*** creates a new type that has a fixed set of values
  - These values are named
  - Only these values may be used
- Enumeration provides compile time guarantees about the set of permitted types

```
enum TileCode {
    SQUARE,
    DIAMOND,
    NINJA_STAR
};
```

```
int main(void) {
    TileCode tilec = SQUARE;

    ...
}
```

Function/Method Overloading

RMIT UNIVERSITY

Function/Method Overloading

- ***Overloading*** is where the same name is used for multiple functions/methods
  - The overloaded functions/methods must have different types**
    - In C++ this means the parameters must be different
    - Be careful about auto-type casting (see later in the lecture)
  - The overload method that is called is determined by the method's type
- Differs to shadowing, since all versions of the function/method can be called

- ** Some languages (such as Haskell) allow the functions to have the same parameters, but return different types.

Function/Method Overloading

- The overloaded method that is called is determined by the method's type

```
int size();
int size(Node* node);

void addBack(const int data);
Node* addBack(Node* node, int data);
```

Constructor Overloading

- We have previously discussed constructor overloading, which allows multiple constructors to be created on a class:

```
Node::Node(int data) :
   Node(data, nullptr)
{}

Node::Node(int data, Node* next) :
   data(data),
   next(next)
{}

Node::Node(const Node& other) :
   Node(other.data, other.next)
{}
```

Class Inheritance

RMIT UNIVERSITY

Class Inheritance

- As with other object orientated languages, C++ classes can ***inherit*** from other classes:
  - Similarities to Java:
    - Public, protected, private scope
    - Method overriding
    - Polymorphism
  - C++ allows for multiple inheritance, that is, a class can have multiple parents

C++ Class Inheritance

- Details to highlight:
  - Defining inheritance
  - Method virtualisation
  - Field names
  - Constructors
  - Deconstructors
  - Parent non-overridden methods
  - Parent overridden methods
  - Code re-use
  - Multiple Inheritance

Defining Inheritance

- Classes may derive from a ***base-class*** or ***super-class*** (not extend - slightly different terminology) another class, using the syntax below.
  - The public keyword is a technical necessity (for our purposes)
  - This ensures all public, protected, and private entities of the base class keep the same scope in the derived class

Derived Class
Sub-class

Base Class
Super-class

```
class LinkedListSize : public LinkedList {
    ...
}
```

Method Virtualisation

- For a derived class to override a method in the base class:
  - The method must be `virtual` (in both classes)
  - The method must have the same scope
  - The method must have the same type
- A derived class can overload a method in the base class without virtualisation

```
class LinkedList {
public:

   ...

   virtual int size();
   virtual void clear();
   virtual int get(int i);
```

```
class LinkedListSize : ... {
public:

   ...

   virtual int size();
   virtual void clear();
```

Field Names

- Fields for public and protected must have unique names
  - Otherwise shadowing occurs
- Private fields may share names

Constructors

- The first operation of the constructor of a derived class must be to call:
  - Another constructor of itself or,
  - One constructor on it base class(es)**
    - ** For multiple Inheritance

```
LinkedList::LinkedList() {
   head = nullptr;
}

LinkedListSize::LinkedListSize() :
   LinkedList(),
   listSize(0)
{}
```

Deconstructors

- Decontructors must be virtual
  - Otherwise they cannot be properly called
    - Constructors don’t have to be virtual
  - C++ will automatically call the deconstructor of call base class(es)
    - Including all classes in an extended class hierarchy

```
class LinkedList {
public:

   LinkedList();
   virtual ~LinkedList();
```

```
class LinkedListSize : ... {
public:

   LinkedListSize();
   virtual ~LinkedListSize();
```

Non-Overridden Parent Methods

- A non-overridden method of a base class may be called by the derived class directly, without any additional syntax.

Overridden Parent Methods

- An overridden method of a base class may be called by the derived class directly, by explicitly referring to the namespace in which that method exists
  - That is, by using the qualified name of the method
  - The qualified name uses the base class name, and optionally the namespace of the base class

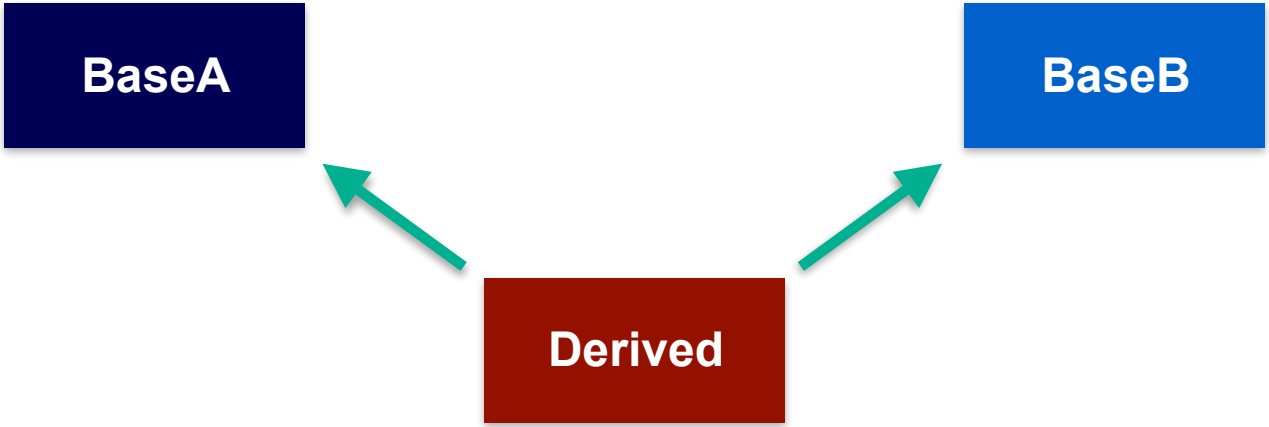
```
void LinkedListSize::clear() {
   LinkedList::clear();
   listSize = 0;
}
```

Code Re-use

- Where possible, a derived class should re-use as much code as possible from it's base class(es), rather than overriding and re-implementing methods
- Things to consider include:
  - If a method requires no changes, it should not be overridden
  - An overriden method should call the base class version for common code
  - If multiple derived classes use similar logic, that should be placed as a protected method in the base class

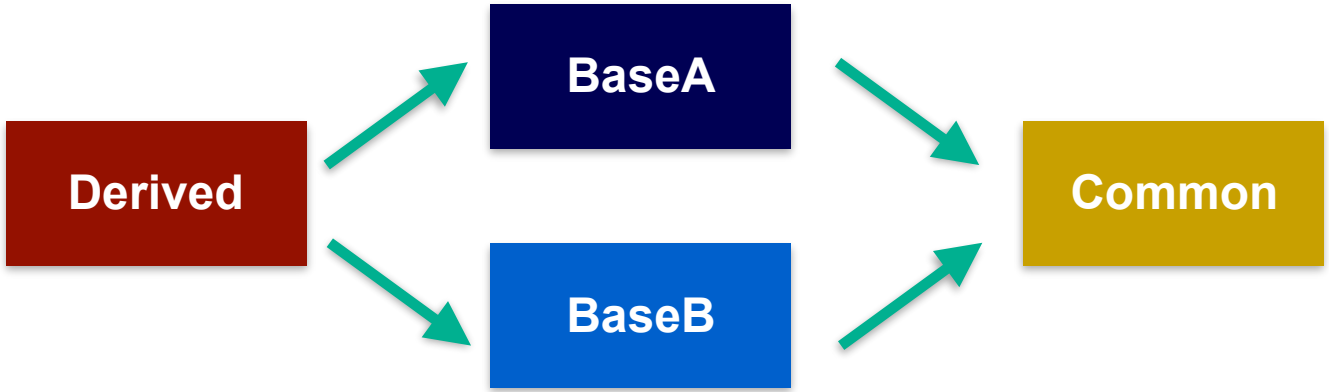
Multiple Inheritance

- A class may derive multiple other classes provided that this does not generate a circular inheritance loop
  - The derived class inherits all methods and fields of all base classes
  - If the base classes have the same methods and fields, qualified version of calling the fields and methods must be used



Multiple Inheritance - Diamond Problem

- Multiple inheritance can result in diamonds
  - A derived class inherits from a base class multiple times
  - This results in duplicates of the same methods and fields
  - Can be handle, but take a lot of properly qualified methods/fields



Polymorphism

RMIT UNIVERSITY

Polymorphism

- ***Polymorphism*** is re-interpreting:
  - An object of a derived class,
  - As an object of a base class
- The object retains is original type
  - The polymorphic type is purely an interpretation of the object
  - Any methods the derived class overrides are what get called

Polymorphism Not Typecasting

- Polymorphism is not typecasting
  - The object retains its original type
  - Typecasting, changes the type of an entity
- In C++, polymorphism only occurs with classes and objects
  - Primitive types are not polymorphic
  - “Compatible” primitive types, (such as integers, floats and doubles) use a combination of implicit type casting and type promotion

Code Re-use

- The most appropriate class should be used in the code
- Things to consider include:
  - A good principle is to use the most “general” base class for any block of code
    - This is determined by what methods need to be called
    - What other code may be passed the object
  - Typecasting should be avoided

Generics

RMIT UNIVERSITY

Genreics

- ***Generics*** is the concept of writing code in such a way that the type(s) of named entities do not need to be specified in the code that is written
  - The types are instantiated at compile or runtime
    - depending on whether the language is statically or dynamically typed
  - The generic type acts as a placeholder
- Generics differs from Polymorphism
  - (Though many will incorrectly refer to uses of polymorphism as “generics” or “generic code”
  - In polymorphism, the type of an entity is always known
  - The code may be flexible, or “general”, but it is not generic

Templates

- C++ provides generics through ***templates***
- A template, as the name suggests, is like a scaffold of code that “isn’t real”
  - A template uses placeholders for where typenames are used in code
  - A template cannot be directly compiled until the typename is instantiated
  - A template is instantiated when some other piece of code “fills in” the typename

Templates - Instantiating

- To instantiate a template:
  - The template “scaffold” is copied!!
  - All of the placeholder types are filled with the instantiated copies
    - Similar to a `#define`
  - The copied instantiated template is what is compiled
  - The placeholder typename can be instantiated to any valid type, including primitive types, arrays, pointers, references, enumerations and/or classes.
- This is very different to other languages, such as Java.

Templates - Implementing

- To implement generic code with a template:
  - The complete definition (not just declaration) of the template must be provided before the location the template is instantiated
  - Typically templates are placed in header files
- The same templates may be instantiated, copied, and compiled multiple times
  - This is why some commentators call templates a “hack”
  - For efficiency, you can manually provide the instantiation of a template for a given type
    - The C++ STL does this in a few locations

Function Templates

```
template<typename T>
T powGeneric(T base, int power) {
  T retVal = 1;
   for (int i = 0; i != power; ++i) {
      retVal *= base;
   }
   return retVal;
}
```

Class (declaration) Templates

```
template<typename T>
class Node {
public:

   Node(T data, Node<T>* next);
   Node(const Node<T>& other);
   ~Node();

   T   data;
   Node<T>* next;
};
```

```
template<typename T>
class LinkedList {
public:

   LinkedList();
   ~LinkedList();

   int size() const;
   void clear();
   T get(int i) const;

   void addFront(const T data);
   void addBack(const T data);

public:
   Node<T>* head;
};
```

Class (definition) Templates

```
template<typename T>
Node<T>::Node(T data, Node<T>* next) :
   data(data),
   next(next)
{}
```

```
template<typename T>
T LinkedList<T>::get(int i) const {
   Node<T>* node = this->head;
   for (int count = 1; count <= i; ++count) {
      node = node->next;
   }
   return node->data;
}
```

Templates - Issues

- If a type isn't known when writing a template, then:
  - How can operators be used? (Such as +, -, ==, etc)
  - How can assignment be used? (ie, x = y)
    - What does assignment mean, between a primitive type, array, pointer, reference, enumeration, and class?
  - What functions or methods can the generic entity be passed to?
  - Can a method be called on the generic entity?

Templates - Issues

- These issues are all resolved at compile time by the copying process
  - The copied and fully-instantiated template is compiles as if it the code was originally written that way
  - Normal C++ rules then take over
- There are “standard” C++ ways of doing things
  - Copy constructors
  - Move semantics
  - Operator overloading
- You have to assume that a user of the template will implement the necessary code features to work with the generic code.
  - C++20 is introducing ways of specifying what is required in the type system

Typecasting

RMIT UNIVERSITY

Typecasting

- ***Typecasting*** changes the type of an entity into another, compatible, type.
- Forms of typecasting:
  - Explicit typecasting is where the programmer writes a cast
  - Implicit typecasting is where the compiler inserts a typecast

Implicit Typecasting

- ***Implicit typecasting*** is where the compiler inserts a typecast
  - Also termed type coercion
- Implicit typecasting can be defined within a language using promotion rules
  - Types may be defined in a hierarchy
  - Types may be implicitly promoted (or demoted) along the hierarchy
  - In C++ any lossless type conversation is done through promotion
- A language may also define “compatible” types
  - This may be lossy, such as double to integer, and vice-versa

Explicit Typecasting

- ***Explicit typecasting*** is where the programmer writes a cast
- C++ Supports 4 types of type conversion
  - `(T)U` - C-style cast. Forces casting even if types are incompatible
    - At a minimum, the raw binary will be forced into being interpreted as if it is of the original type
  - `static_cast<T>(U)` - compile time cast with check for compatible types
  - `dynamic_cast<T>(U)` - runtime cast with runtime check for compatibility
  - `reinterpret_cast<T>(U)` - C++ version of C-style casting
- For pointers:
  - Static and dynamic cast check the pointer type
  - Dynamic cast returns a `nullptr` if the objects are actually incompatible at runtime, whereas static cast does not.

Operator Overloading

RMIT UNIVERSITY

Operator Overloading

- Permits classes to make use of the 38 C++ operators, including:
  - Comparison operators: ==, !=, <, <=, >, >=
  - Arithmetic operators: +, -, *, /, %, ^, +=, -=, etc.
  - Increment/Decrement: ++, --
  - Assignment operator: =
  - I/O operators: <<, >>
  - Access: [], *, ->, &
- Generally operators can be divided into:
  - Operators that do not modify the class
  - Operators that modify the class, and return it
  - Operators that return a new class
- https://en.cppreference.com/w/cpp/language/operators

Operator Overloading

- Operators can be overloaded through:
  - Methods (member functions) on a class
  - Functions external to the class
- For this course we will stick to member functions, though the concept for function versions is very similar

Operator Overloading

- In an expression:

```
lhs <operator> rhs
```

- The overloaded operator method is called on the object on the left-hand side
  - The object on the right-hand side is passed as a parameter

Comparison Operators

- All comparison operators take the form:

```
bool operator???(const Class& rhs) const;
```

  - They return a bool
  - Take the RHS as a constant reference
  - Are a const method
- You have to implement any comparison operator you wish to use:
  - But, only really need to implement two:

```
bool operator==(const Class& rhs) const;
bool operator<(const Class& rhs) const;
```

  - The other comparison operators are implemented from these

RMIT
UNIVERSITY