COMP6771 19T2 (https://webcms3.cse.unsw.edu.au/COMP6771/19T2)

Week 10 ▾ Tutorial ▾ Sample Answers ▾

Advanced C++ Programming (https://webcms3.cse.unsw.edu.au/COMP6771/19T2)

Consider the following code:

```
class Base {
 private:
  int i_;
};

class Derived : public Base {
 public:
  void foo(Derived& o) {
    j_ = o.i_ + o.j_;
  }
 private:
  int j_;
};
```

1. Why does this code not compile?

   You cannot access a private member from a derived class.

   Change the base class to:

   ```
   class Base {
   protected:
     int i_;
   };
   ```

   The code will now compile, but what are the potential disadvantages of this solution?

   It is generally considered bad practice to make data members anything other than private without good reason. Otherwise, any class inheriting off it would be relying off its implementation, rather than its interface.

2. Change the code to:

   ```
   class Base {
    public:
     const int& GetI() const { return i_; }
    private:
     int i_;
   };

   class Derived : public Base {
    public:
     void foo(Derived &o) {
       j_ = o.GetI() + o.j_;
     }
    private:
     int j_;
   };
   ```

   What benefits does this approach give us? Why do we return a const reference to *i_*?

   This allows a subclass to only depend upon the interface of the base class, rather than the implementation. This means that if the base class chooses to refactor or modify the code, the subclass won't be affected. We return a const reference to i because a getter should work on a const object. By returning a const reference, rather than a non-const reference, we both disallow modification, and allow the function to be called on a const object. If we later want to add a setter, this gives us the power to do so, through either a non-const getter, or a manual setter.

Consider the following code:

```
#include <iostream>

class B {
 public:
  virtual void f() {
    std::cout << "B ";
  }
};

class D : public B {
 public:
  void f() override {
    std::cout << "D ";
  }
};

int main() {
  B b;
  D d;
  b = d;
  B& bref = b;
  D& dref = d;
  B& dbref = d;
  b.f();
  d.f();
  bref.f();
  dref.f();
  dbref.f();
}
```

1. For each of these variables, what is the static and dynamic type (for b, consider this both after and before assigning d to it)?

   The static type is the declared type of a variable. Hence, b has static type B, d has static type D, bref and dbref have static type B&, and dref has static type D&.
   The dynamic type is the type a variable actually is at runtime.
   b and d are neither references or pointers. Hence, their dynamic type is **always** the same as their static type. Hence, b and d have dynamic types B and D respectively.
   bref, dref, and dbref are references. Hence, their dynamic type is the dynamic type of the object they refer to. Hence, bref has dynamic type B&, dref has dynamic type D&, and dbref has dynamic type D&.

2. The assignment *b = d;* is legal but is poor style as it causes the object slicing problem. What is the object slicing problem?

   When you try and assign a derived class into a variable of type base class, the base class only has sufficient space for the base class subobject. Hence, it copies over the base class subobject and completely ignores the derived class' data.

3. What is the output of this code?

   Since f is a virtual function, we use the dynamic type of objects to determine which function to call (if it was not, we would use the static types). Hence, we output "B D B D D".

4. How can we correct this code to prevent the object slicing problem?

   Instead of storing an object of fixed size, store an object that can have dynamic size (specifically, a pointer). Use a std::unique_pointer<B> instead of a B for the type of b.

Draw the vtables for classes A, B and C:

```
class A {
 public:
  virtual void f(int) {}
  virtual int g() {}
  void a() {}
  virtual ~A() {}
};

class B : public A {
 public:
  void f(int) override {}
  virtual int h() {}
};

class C : public B {
 public:
  virtual void f(int, int) {}
  virtual void x() {}
  static void b() {}
};
```

A	**B**	**C**
A::f(int)	B::f(int)	B::f(int)
A::g()	A::g()	A::g()
~A()	~B()	~C()
VTABLE END	B::h()	B::h()
	VTABLE END	C::f(int, int)
		C::x()
		VTABLE END

Consider the following code:

```
#include <iostream>

class Animal {
 public:
  Animal() {
    std::cout << "Animal\n";
  }
};

class Amphibian {
 public:
  Amphibian() {
    std::cout << "Amphibian\n";
  }
 private:
  Animal a1_;
};

class Fish {
 public:
  Fish() {
    std::cout << "Fish\n";
  }
 private:
  Animal a1_;
  Amphibian a2_;
};

int main() {
  Fish a3;
}
```

1. Work out on paper/whiteboard what the output of this program is.

    Animal (Fish member)
    Animal (Amphibian member)
    Amphibian (Fish member)
    Fish

2. Change the declaration of class Fish to:

    ```
    class Fish  : public Amphibian {
    ```

    Work out what the new output is.

Animal (Amphibian member)
Amphibian (Fish base class)
Animal (Fish member)
Animal (Amphibian member)
Amphibian (Fish member)
Fish

3. Why are the outputs different?

Each derived class has a base class subobject which must be initialised before initialising anything else in the class.

Consider the following code:

```
#include <iostream>

struct X {
  ~X() { std::cout << "~X()\n"; }
};

struct Y {
  ~Y() { std::cout << "~Y()\n"; }
};

class A {
  X x;
 public:
  ~A() { std::cout << "~A()\n"; }
};

class B : public A {
  Y y;
 public:
  ~B() { std::cout << "~B()\n"; }
};

int main() {
  B b;
}
```

On paper/whiteboard work out the output of this program.

~B()
~Y() (B member)
~A() (base class)
~X() (A member)
Note that this is just the reveres order that you would do a constructor in.

Revision Questions Consider the following code:

```
#include <list>
#include <iostream>

class X {
public:
  X() { std::cout << "ctor "; }
  X(const X&) { std::cout << "copy-ctor "; }
  ~X() { std::cout << "dtor "; }
};

int main() {
  {
    std::cout << "Pointer: "
    std::list<X*> l;
    X x;
    l.push_back(&x);
  }

  {
    std::cout << "\nValue: "
    std::list<X> l;
    X x;
    l.push_back(x);
  }
}
```

On paper/whiteboard work out the output of this program.

Pointer: ctor dtor Value: ctor copy-ctor dtor dtor

A friend function of a class can access:

A. Only the public members of the class
B. Only the public and protected members of the class
C. All members of the class
D. All members of the class and its base classes
E. Only the public and protected members of the class and its base classes

All members of the class.

On paper/whiteboard work out the output of the following program:

```
#include <iostream>
#include <utility>

void e(const int& i, int j, int k) {
  std::cout << "1 ";
}

void e(int& i, int j, int k) {
  std::cout << "2 ";
}

template <typename A, typename B, typename C>
void f(A&& a, B &&b, C&& c) {
  e(std::forward<A>(a), std::forward<B>(b), std::forward<C>(c));
}

int k{1};
int g() { return k; }

int main() {
  f(1,2,3);
  int i{1};
  f(i,2,3);
  const int &j = i;
  f(j,2,3);
  f(g(),2,3);
}
```

```
1 2 1 1
```

1. What are the differences (e.g., types, semantics, memory usage) between a T, a pointer to a T, and a reference to a T?

type	**T**	**T***	**T&**
semantics	A value	An optional reference to a value	A reference to a value
memory	sizeof(T)	sizeof(pointer)	sizeof(pointer)

2. Explain the differences in the meaning and effect of using *const* in the following member function prototypes:

```
const T& getValueAtIndex(int i);
T getValueAtIndex(const int i);
T getValueAtIndex(int i) const;
```

The first returns a const reference. It returns a reference to the value stored in the container, but you can't modify the value through this reference.
The second returns a copy of the value stored in the container. A const int parameter is no different to an int parameter.
The third also returns a copy of the value stored in the container. The const here indicates that this is a const member function, and so does not modify the container.