1 When const references should be used and should not be used

1.1 Parameters to constructors

We’ve already seen code like the following:

```cpp
int add10(int x) {
    x+=10;
    return x;
}

int add10ref(const int& x) {
    int y = x + 10;
    return y;
}
```

Here, add10 is more efficient than add10ref because add10 involves one copy constructor call whereas add10ref involves construction of a const reference and a constructor call. The point is that if we want to modify a parameter passed into a function, and if we do not want the variable passed as a parameter from another scope to be modified, then it makes sense to pass by value, because we’re inevitably going to create a copy anyways.

The same principle applies in constructors when we wish to use parameters to construct member data of a class. Indeed, a parameter in the form of a const reference is of course const, meaning that the data to which it is bound cannot be used to move construct a class member variable.

```cpp
struct Efficient {
    Efficient(std::string _str) : str(std::move(_str)) {}
    std::string str;
};

struct Inefficient {
    Inefficient(const std::string& _str) : str(_str) {}
    std::string str;
};
```

In both of the classes above, the goal of the constructor is to initialize the member str to contain the data stored/bound to the parameter _str. To understand which is more efficient, we need to consider the different kinds of parameters a user will pass into the constructor:

1. An std::string l-value.
   
   (a) Efficient: _str is copy constructed from the l-value, then str is move constructed from _str. 1 copy and 1 move.
   
   (b) Inefficient: _str is bound to the l-value, then str is copy constructed from _str. 1 const reference initialization and 1 copy.

2. An std::string r-value.
(a) Efficient: \_str is move constructed from the r-value, then \_str is move constructed from \_str. 2 moves.

(b) Inefficient: \_str is bound to the r-value, then \_str is copy constructed from \_str. 1 const reference initialization and 1 copy.

3. A string literal (type char[])

(a) Efficient: \_str is constructed from the string literal, then \_str is move constructed from \_str. 1 construction and 1 move

(b) Inefficient: a temporary string is constructed from the literal, then \_str is copy constructed from \_str. 1 construction and 1 copy.

Normal construction and copy construction are much more time-consuming than move construction because they involve the allocation of new memory, whereas move construction just requires the reassignment of ownership of memory. If we were to score the constructors by subtracting a point for each normal or copy construction, Efficient would be scored -1,0,-1 and Inefficient would be scored -1,-1,-2. Thus, we see that Efficient is more efficient, as the name would suggest.

1.2 Return value

The return type of a function can be a const reference if the return is not a reference to a local variable of a function. For example,

```cpp
1   const int& fun () {  
2     int x = 10;  
3     return x;  
4   }
```

is invalid code because the variable x local to fun; thus, when fun is finished running, the variable x will be destroyed. Accordingly, the const reference will then be bound to data that is destroyed when the function is finished running. In this case, one should return by value. On the other hand, the following code is valid

```cpp
1   struct Data {  
2     const int& fun () {return dat;}  
3     int dat = 10;  
4   };  
```

Here fun will return a const reference to dat, which is a member variable of a Data object, so it will not be destroyed when fun is finished running. In summary, one should make sure that a returned reference does not refer to data that will be deleted.

2 Polymorphic classes

In the real world, objects are organized into categories and subcategories. Subcategories of media include print media, photography, and cinema. Subcategories of print media include periodicals, books, and outdoor media (billboards, etc.). Subcategories of periodicals include newspapers and
magazines. The objects in each of these categories often have many features in common. All media has a purpose: whether the purpose is entertainment, advertising, education, etc. there is a purpose. All periodicals have a title, a list of authors, an editor, an owner, etc., but they do not have a director or a camera person as in cinema. The point is that objects in a category have features in common that they may not have with objects in other categories.

The following example is less of a real world example and more of an example you may actually see while coding.

```cpp
class IntOperator {
public:
    IntOperator(bool _order_matters) : order_matters(_order_matters) {}
    virtual std::string get_info() const = 0;
    bool isOrdered() {
        return order_matters;
    }
private:
    bool order_matters;
};
class BinaryIntOperator : public IntOperator {
public:
    BinaryIntOperator(bool _order_matters) : IntOperator(_order_matters) {}
    virtual int eval(int x, int y) = 0;
};
class BinaryIntSum final : public BinaryIntOperator {
public:
    BinaryIntSum(bool _order_matters) : BinaryIntOperator(_order_matters) {}
    virtual std::string get_info() const override {
        return "adds two integers";
    }
    virtual int eval(int x, int y) const override {
        return x + y;
    }
};
class BinaryIntDivision final : public BinaryIntOperator {
    BinaryIntDivision(bool _order_matters) : BinaryIntOperator(_order_matters) {}
    virtual std::string get_info() const override {
        return "divides two integers with integer division";
    }
    virtual int eval(int x, int y) const override {
        return x/y;
    }
};
class TernaryIntOperator : public IntOperator {
public:
    TernaryIntOperator(bool _order_matters) : IntOperator(_order_matters) {}
    virtual int eval(int x, int y, int z) = 0;
};
```

Here, we create classes that represent integer operators, where an integer operator is simply a function that takes in some number of values and outputs an integer from those values. We list some features below:
1. The base class, called IntOperator, has a pure virtual function `get_info()` that should return information about the specific operator. We know it is a pure virtual function because of the “=0” after the declaration. Having a pure virtual function makes a class into an “abstract class”—the terminology being due to the fact that we cannot have an instance of an abstract class, so an abstract class cannot exist in memory. Pure virtual functions are used within an (abstract) class when it is desired that every class inheriting from the abstract class implements the pure virtual functions. Indeed, if a class derived from an abstract class does not implement a pure virtual function, the derived class is also abstract. Since BinaryIntOperator and TernaryIntOperator do not implement `get_info`, and since they have pure virtual functions of their own, they are abstract as well. However, BinaryIntSum and BinaryIntDivision are not abstract since they implement all pure virtual functions.

2. We see in the constructors for all classes derived from IntOperator that the constructor of each classes' parent is called. Indeed, a derived class contains all of it’s parent classes; thus, one must construct a class’ parent before specifically constructing the derived class itself. Here, we only need to call the parent constructor, since the derived classes’ do not add any member variables to IntOperator.

3. The BinaryIntSum and BinaryIntDivision classes are marked as final because no other classes can be derived from those classes.

4. The member functions of BinaryIntSum and BinaryIntDivision include the keyword override because they are redefining virtual functions provided by their parent.

5. Each class is marked as having public inheritance (e.g. TernaryIntOperator : public IntOperator). This means that public functions and data are available with public access in the derived class, protected functions and data are available with protected access in the derived class, and private functions and data are not available to the derived class. Thus, the main difference between the public/private dichotomy we’re used to and the current situation is the addition of the keyword protected. Protected data acts like private data but is accessible to derived classes. Private data is not accessible in derived classes.

2.1 Polymorphism

Polymorphism in C++ takes advantage of the fact that objects in the same category are used in the same manner; indeed, the differences are often in the details rather than in the generalities. For example, consider driving a truck vs. driving a boat. Both vehicles have the same core functionality: you can turn them on, turn them off, turn left, turn right, accelerate, and decelerate. Now let’s imagine that we have a remote control that can connect to any vehicle and allow us to drive the vehicle, giving us the listed functionalities. Then we do not even need to know what kind of vehicle we are controlling to drive the vehicle: the remote control knows and will supply the correct implementation details when we attempt to turn right for example (truck: rotate wheels; boat: rotate engine).

To translate the above example into C++, the remote control is like a pointer (or reference) of type Vehicle pointing to objects of type Truck and of type Boat. In the class Vehicle, there would be virtual functions `on()`, `off()`, `left()`, `right()`, `accelerate()`, and `decelerate()`. Additionally, in the class Truck for instance, there would be non-virtual functions that are not in Boat, such as `turnOnAC()`,
loadTruckBed(), toggle4WD(), etc. The remote control would be able to drive the truck with the
basic functionality on(), off(), etc. but would not have be able to do more specific tasks such as
turnOnAC(). For these more specific tasks one would need a more specific remote control (a Truck
remote control); i.e. one would need a pointer of type Truck.

To summarize the preceding paragraphs, consider the following example

```cpp
int main() {
    Vehicle* vh_truck = new Truck();
    Vehicle* vh_boat = new Boat();
    // drive vehicles — OK using basic functionalities
    vh_truck->on();
    vh_truck->accelerate();
    vh_boat->on();
    vh_truck->left();
    vh_boat->right();
    vh_truck->off();
    // etc.
    // however we cannot use the Vehicle ‘‘remote control’’ for the following
    // vh_truck->turnOnAC(); error!
    Truck* truck_ptr = dynamic_cast<Truck*>(vh_truck);
    truck_ptr->loadTruckBed(); // OK! using Truck remote control!
    truck_ptr->toggle4WD();
}
```

Polymorphism is the idea that a single remote control can be used to drive many different vehicles:
it is the provision of a single “interface” to different “types.”

### 2.2 Details on Virtual Functions

Remember, data is stored in RAM as a sequence of 0s and 1s. Simply knowing where data is stored
does allow us to obtain the actual data: we need to know the type of the data in order to interpret
the 0s and 1s. Indeed, even though a std::string and an array of doubles may take up the same
amount of space in memory, and consist of the exact same sequence of 0s and 1s, the data itself is
of course very different because the types are different.

Consider the following code.

```cpp
struct A {
    virtual void f() {std::cout << '"'"hi"'"' << std::endl;}
};
struct B : public A {
    virtual void f() {std::cout << '"'"hello"'"' << std::endl;}
}
void fun(A* ptr) {
    ptr->f();
}
int main() {
    A* aptr = new A;
    A* bptr = new B;
    fun(aptr);
    fun(bptr);
}
```
From the perspective of the function fun, the calls fun(aptr) and fun(bptr) are the same: in both we dereference the ptr to A and call the function f. However, polymorphism means that fun(aptr) will print “hi” and fun(bptr) will print “hello.” We noted above that given a memory location, it is required to know the type of the data stored in memory in order to interpret the sequence of 0s and 1s correctly. Thus, how is this possible when the function fun only knows that the parameter ptr is a pointer to A; in other words, how is the correct version of f called when fun does not know if the object pointed to is an A object or a B object?

The answers to the questions above lie in vtables (virtual member table). Every class in a hierarchy of polymorphic classes has an associated vtable. The vtable of a class is a collection of function pointers to the correct (most derived) implementation of the functions in that class. If we create an object of a particular polymorphic type, a pointer called a vpointer is added to the object that points to the vtable associated with the class. Thus, when a virtual function of an object is called, the vpointer is used to access the vtable, and then the correct implementation of the virtual function is found in the vtable and used.

In the code above, ptr->f() is able to produce either “hi” or “hello” because the vptr of the object ptr points to can be different, which means that different vtables will be searched depending on the polymorphic type of the object to which ptr points, giving different implementations of the function f.

2.3 Destructors

One should make the destructors of a hierarchy of a polymorphic hierarchy of classes virtual. This is so that when one deletes a Base type pointer to a Derived object, the Derived class’ destructor will be called rather than the Base class’ destructor.

2.4 Slicing

Polymorphism only applies to pointers and references. Continuing with the code in the previous section,

```c
// previous code
void foo(A obj) {
    obj.f();
}

int main() {
    // previous code
    foo(*aptr);
    foo(*bptr);
}
```

The code in the main() method will print out “hi” twice. This is because the parameter to the function is no longer a pointer: it is now simply an A object. But why can’t we just follow the vptr to the vtable with the correct code as before? Remember that a derived object contains a base object. When a B type object is passed into foo, a new A object is created from the A object that is contained within B. The result is that the vptr of B is thrown away, and all that is left is the A vptr, which is used inside of the body of foo to call f(). Removing all of the B data from a
B object and retaining only the A object is called slicing.

2.5 Static and Dynamic Types

If we have a Base type pointer to any object, then the static type of that object is Base: the static type is just the type of the pointer or reference referring to the object. But as we have seen it is possible to have a Base pointer to a Derived object, and polymorphism allows us to dereference the Base pointer and obtain an object that acts like a Derived object. In this case, the dynamic type of the object is Derived, while the static type is of course Base. The dynamic type is determined by which vtable the vptr of an object points to.

Continuing the example above,

```c
int main() {
    // previous code
    A obj; // static and dynamic type A
    B obj1; // static and dynamic type B
    A obj2 = obj1; // static and dynamic type A (slicing)
    A* obj_ptr1 = obj; // static type A, dynamic type B
    A* obj_ptr2 = obj1; // static type A, dynamic type B
}
```

2.6 Casting

The main difference between static_cast and dynamic_cast is that static_cast involves no runtime checks on feasibility, whereas dynamic_cast, does. Another way to put this is that static_cast only checks the static types of the objects being casted: as long as the static types of the pointers are in the same class hierarchy, the compiler will not complain. On the other hand, dynamic_cast checks the dynamic types of the objects being casted: the dynamic_type of the object being casted must be at least down in the hierarchy from the object being casted to. This will become clearer in a second.

It is always possible upcast (towards Base) a pointer, since we know that a Derived object contains a copy of all of its parent classes, so casting will simply redirect the pointer to one of these internal parent classes. In this case one can just use static_cast, which doesn’t involve any runtime checks on feasibility, since it is always possible to upcast. On the other hand, it is often that case that one want to cast a Base type pointer to a Derived type pointer—to downcast. However, it is not always possible to do this because a Base type pointer can refer to a Base object or to a Derived object, and if the pointer refers to a Base object than casting will fail. In this case one should always use dynamic_cast, since dynamic_cast will check the dynamic type of an object at runtime to determine whether casting is possible. If casting is not possible, dynamic_cast will return a nullptr.

Continuing the above example,

```c
int main() {
    // previous code
    B* obj_ptr1 = dynamic_cast<B*>(obj_ptr); // ok
```
// B* obj_ptr1 = static_cast<B*>(obj_ptr); bad – no runtime check of feasibility
B* obj_ptr2 = dynamic_cast<B*>(obj_ptr1); // obj_ptr2 set to nullptr because dynamic types cannot be casted
if (obj_ptr2) {
    std::cout << 'B object' << std::endl; // will not print because obj_ptr2 is nullptr
}
}

3 static

A static member variable or member function of a class is a variable or function that that exists regardless of whether the class itself has been instantiated. The point of having a static member function, even though it is not associated memory-wise to a class, is that it is able to access private static members of a class. Consider the code below

struct A {
    static int fun() {return n;}
    static int n;
};
int A::n = 1;
int main() {
    int x = A::fun();
}

This code will compile even though in main we have not actually constructed an A object.

4 Exceptions

Dynamic casts between reference types will throw a std::bad_cast exception if the dynamic cast is not possible. Note that casting between pointers can return a nullptr, but there is no reference analogous to nullptr so exceptions are thrown instead. A thrown exception should be caught by the programmer using a try/catch block if the programmer wants the code to continue running. Uncatch exceptions cause the program to exit.
In the code above, the `dynamic_cast` inside of the `fun` function will throw a `std::bad_cast` exception, which will be caught by the `try` block inside `main`. Note that “boo!” is not printed because the exception occurs before “boo!” is printed, causing `fun` to unwind immediately and return to `main`. Since the error thrown is a `std::bad_cast`, it will be caught inside of the first `catch` block. The `throw` inside of the `catch` block will cause the exception to be further passed on to the last `catch` block, which will print “???” . After this, the program will continue running. Note if `throw` were included in the last `catch` block, the program would quit immediately because the exception will be unhandled.