63. Constant Method and Self Reference

Objectives

- Understand the `this` pointer
- Understand the uses of the `this` pointer
- Return a reference to an object
- Return a reference to an instance variable
- Understand the dangers of returning references to an instance variable
- Make parameters constant to protect from accidental modification
- Make `*this` constant to protect from accidental modification

Recall

Suppose we have a class `C` with a method `m`. Suppose an object `obj` of `C` invokes `m`. In the body of method `m`, members and member functions without objects are those automatically associated with `obj`

```
// Int.h

class Int
{
public:
    Int(int x) : x_(x) {}

    void print()
    {
        std::cout << x_ << "\n";
    }

private:
    int x_;
};
```

When i invokes `print()`, the `x_` refers to `i.x_`.

```
// main.cpp
#include “Int.h“

int main()
{
    Int i(5);
    i.print();

    Int j(-3);
    j.print();
}
```

Now `x_` refers to `j.x_`.

Again, this applies to methods as well...

Add methods `m1`, `m2` into the `Int` class. For simplicity, I'm inlining the methods into the class header. But remember that you should inline minimally.

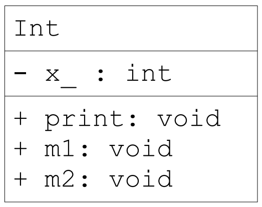
```
// Int.h
...
class Int
{
public:
...
    void m1()
    {
        std::cout << "m1:" << x << "\n";
        m2();
    }

    void m2()
    {
        std::cout << "m2:" << x << "\n";
    }
...
```

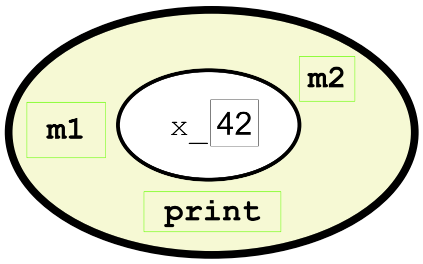
```
// main.cpp
#include "Int.h"

int main()
{
    Int i(42);
    i.m1();

    std::cout << std::endl;

    Int j(0);
    j.m1();

    return 0;
}
```

(Note: This is a review. I already talked about methods calling methods when we were looking at the `Date` class.)

Thinking of Objects

You should think of a class as a rubber stamp for creating objects.
You should think of each object as a computational machine.





You can (and should) think of the “stuff” on the green band (the `m1`, `m2`, `print`) – the `public` things – as things that outsiders can access while the stuff on the inside (the `x_`) – the `private` things – are not available to outsiders. By outsiders, I mean functions not in the class and methods not in this class (i.e. methods of another class).

What is the Invoking Object

What if a method must know the invoking object? Here's such an example:

```
class Int
{
public:
    ...
    Int max(const Int & c)
    {
        if (c.x_ > x_)
        {
            return c;
        }
        else
        {
            return ???;
        }
    }
private:
    int x_;
};
```

```
int main()
{
    Int a(100), b(1);
    Int c = a.max(b);

    return 0;
}
```

For `a.max(b)` we want to return `a`. But what's the name of `a` in the code of `max()`??? In general, what's the name of the object making the call?

There is a secret parameter in every method. You do not pass this parameter into the method. It's done for you automatically.

What is this secret parameter??? ...

The `this` Pointer

Try the program below.

```
// Int.h

#ifndef INT_H
#define INT_H

#include <iostream>

class Int
{
public:
    Int(int x) : x_(x) {}

    void print()
    {
        std::cout << this << x_ << "\n";
    }

private:
    int x_;
};

#endif
```

```
// main.cpp

#include <iostream>
#include “Int.h"

int main()
{
    Int i(42);
    std::cout << &i << "\n";
    i.print();

    Int j(0);
    std::cout << &j << "\n";
    j.print();

    return 0;
}
```

When an object `obj` invokes a method `m`, a special pointer `this` is passed into `m` automatically. The important thing to remember is that:

`this` points to `obj`

If an object `obj` has member variable `x_` and `this` is a pointer to `obj`, then the following are the same:

```
obj.x     (*this).x     this->x
```

This is the same notation used for pointers to `struct` variables. You should ***not*** use `(*this).x`. Write `this->x` instead.

(There is an exception. Static methods do not have the `this` secret parameter discussed here. We will talk about static methods later.)

WARNING: `*p.x` means `*(p.x)` because `.` has higher precedence than `*`. Make sure you try changing the following

```
class Int
{
public:
    ...
    void print()
    {
        std::cout << x_ << "\n";
    }
    ...
private:
    int _x;
};
```

Same!

to this:

```
class Int
{
public:
    ...
    void print()
    {
        std::cout << this->x_ << "\n";
    }
    ...
private:
    int x_;
};
```

OK ... now to use `this` ...

When do you use `this`?

Standard Practice: Don't use `this` in a method just to access members or member functions. In other words, in a method

- `this->x` should be `x`
- `this->m()` should be `m()`

For instance, this does work:

```
...
class Date
{
public:
    …
    int get_year()
    {
        return this->yyyy_;
    }
    void set_year(int y)
    {
        this->yyyy_ = y;
    }
    ...
    void add_m_y(int m, int d)
    {
        this->add_m(m);
        this->add_d(d);
    }
private:
    ...
};
```

but **DON'T DO THAT!!!** Do this:

```
...
class Date
{
public:
    …
    int get_year()
    {
        return yyyy_;
    }
    void set_year(int y)
    {
        yyyy_ = y;
    }
    ...
    void add_m_y(int m, int d)
    {
        add_m(m);
        add_d(d);
    }
private:
    ...
};
```

The pointer `this` is used when it’s used on its own either as `this` or `*this`. In particular `*this` can be used for returning a copy of or a reference to the object making the method invocation and `this` can be used for address comparison (you'll see this later when we need to compare address values when we overload the assignment operator `operator=`.)

(There are exceptions … later. But the above two uses are good enough.)

Returning a copy of `*this`

```
...
class Int
{
    ...
    Int max(const Int & c)
    {
        if (c.x_ > x_)
            return c;
        else
            return *this;
    }
    ...
private:
    int x_;
};
...
```

And now you can try this:

```
#include "Int.h"

int main()
{
    Int a(100), b(1);
    Int c = a.max(b); // a.max(b) should return an
                      // Int object with x_ value of
                      // 100
    c.print();

    return 0;
}
```

Of course, this is better:

```
...
class Int
{
public:
    ...
    Int max(const Int & c)
    {
        return (c.x_ > x_ ? c : *this);
    }
    ...
private:
    int x_;
};
...
```

Get it? In this case, we are returning a copy of the `Int` object that `this` points to, i.e., the object invoking the method.

Notice however that you can also do this:

```
...
class Int
{
    ...
    Int & max(const Int & c)
    {
        return (c.x_ > x_ ? c : *this);
    }
    ...
private:
    int x_;
};
...
```

In this case, a **reference is returned**.

Why is this important? Look at this code:

```
#include "Int.h"

int main()
{
    Int a(100), b(1);
    Int c = a.max(b);
    c.print();

    return 0;
}
```

This means if the return of `Int::max()` is an `Int` and not an `Int &`

```
...
class Int
{
public:
    ...
    Int max(const Int & c)
    {
        ...
    }
    ...
};
...
```

then an `Int` value is returned (it has the same values as `a` or `b`). This `Int` value is used for the invocation of the `Int` copy constructor to initialize `c`:

```
#include "Int.h"

int main()
{
    Int a(100), b(1);
    Int c = a.max(b);
    c.print();

    return 0;
}
```

Make sure you read the last two sentences again (and again and again).

However if a reference is returned:

```
...
class Int
{
public:
    ...
    Int & max(const Int & c)
    {
        ...
    }
    ...
};
...
```

then back in `main()`:

```
#include "Int.h"

int main()
{
    Int a(100), b(1);
    Int c = a.max(b);
    c.print();

    return 0;
}
```

on returning from `Int::max()`, a reference to a or to b is used by the copy constructor to initialize `c`. There's no extra `Int` object created by `Int::max()` that is used to initialize `c`.

Let me repeat again: when you return `Int` (not `Int &`), a copy of the object to be returned is made (by using the copy constructor).

```
...
class Int
{
public:
    ...
    Int max(const Int & c)
    {
        return (c.x > x ? c : *this);
    }
    ...
private:
```

Copy constructor is called using **`c`** or **`*this`**!!! Because return type is **`Int`** (and not **`Int &`**) Remember that!!!

```
    int x;
};
...
```

Whereas if you return `Int &`, no `Int` object is created. The copy constructor used to create `c` will simply reference `a` or `b`.

Returning references

In the previous section, I have shown you an example of returning a reference.

Certain methods and operators update the values of the variables or objects and return the values of the variables or objects. Run this:

```
int x = 5;
x++;
std::cout << x << std::endl;
int y = x++;
std::cout << x << ',' << y << std::endl;
y = (x += 5);
std::cout << x << ',' << y << std::endl;
(x += 5) += 5;
std::cout << x << std::endl;
```

Don't overdo this! Why? Because

```
y = (x += 5);
```

does too many things. A function/method/operator should do ***<u>one</u>*** thing to avoid confusion. This is clearly easier to understand:

```
x += 5;
y = x;
```

Note that the `+=` when called twice:

```
(x += 5) += 5;
```

operates on the same `x`. This means that `+=`, besides incrementing `x`, also returns a reference to x so that the second `+=` operators on the same `x`. If `+=` returns a new `int` value that has the same value as `x` after incrementing, then the second `+=` would have operated on this new `int` value and not on `x`. Therefore the right thing to do is to return a reference to `x` and not a copy of `x`.

Of course returning a reference is faster than returning an object since objects tend to consume more memory. References are implemented using pointers and pointers take up very little memory.

Refer to the `Date` example. Suppose the `add_y` method is to add an integer value to the year value, *<u>and</u>* return the value of the object.

```
// Date.h
...
class Date
{
public:
    ...
    Date & add_y(int);
    ...
private:
    int yyyy_, mm_, dd_;
};
...
```

```
// Date.cpp
...
Date & Date::add_y(int inc)
{
    yyyy_ += inc;
    return *this;
}
...
```

With the above, I can now do this:

```
Date today(1, 1, 1970);
today.add_y(1).add_y(3);
```

The first invocation of `add_y` will add 1 to the year of `today` and then a reference to `today` is returned. Therefore the second `add_y` is called by this reference, which means that the second `add_y` is called by `today`.

However if you do this:

```
// Date.cpp
...
Date Date::add_y(int inc)
{
    y += inc;
    return *this;
}
...
```

i.e., return a `Date` object instead of a reference, then `Date`'s copy constructor is used to create a new `Date` object from `*this`. This object is returned to `main()` and the `add_y` is called by this object so that `today` does not have its year incremented by 3.

We now look at returning references from a function call or method call. There's nothing new here about references.

WARNING: Pay attention to the next programs!!!

Run the following pair of programs. Explain their outputs.

```
#include <iostream>
```

```
int & f(int & x)
{
    x++;
    return x;
}

int main()
{
    int a = 42;
    int & b = f(a);
    std::cout << a << ' ' << b << '\n';
    std::cout << &a << ' ' << &b << '\n';
    b++;
    std::cout << a << ' ' << b << '\n';

    return 0;
}
```

```
#include <iostream>

int & f(int & x)
{
    x++;
    return x;
}

int main()
{
    int a = 42;
    int b = f(a);
    std::cout << a << ' ' << b << '\n';
    std::cout << a << ' ' << b << '\n';
    b++;
    std::cout << a << ' ' << b << '\n';

    return 0;
}
```

The point:

```
int a = 42;
int & b = a;

int & receiver1 = b; // receiver1 references a
int receiver2 = b; // receiver2 gets the value of a

receiver1 = 0; // a is changed
receiver2 = 0; // receiver2 is changed, not a.
```

What about this?

```
int & f()
{
    int x = 42;
    return x;
}
```

This returns a reference to a local variable but you cannot refer to this local variable once it goes out of scope since it will be destroyed at that point. (Note: this is OLD stuff. Yes I have already talked about it.)

Yet more examples on returning `*this` as a reference (DIY)

```
class Point2d
{
public:

    Point2d & set_x(int x) { x_ = x; return *this; }
    Point2d & set_y(int y) { y_ = y; return *this; }

    void print()
    {
        std::cout << x_ << ',' << y_ << '\n';
    }

private:
 int x_, y_;
};

int main()
{
    Point2d p;
    Point2d & q = p.set_x(1);

    q.set_y(2);
    p.print();

    return 0;
}
```

Explain!

```
class Point2d
{
public:
    Point2d & set_x(int x)
    {
        x_ = x;
        return *this;
    }

    Point2d & set_y(int y)
    {
        y_ = y;
        return *this;
    }

    void print()
    {
        std::cout << x_ << ',' << y_ << '\n';
    }

private:
    int x_, y_;
};

int main()
{
    Point2d p;
    p.set_x(1).set_y(2);
    p.print();

    return 0;
}
```

This is similar to the previous program except that I got rid of `q`. Explain!

```
class Point2d
{
public:
    Point2d set_x(int x)
    {
        x_ = x;
        return *this;
    }

    Point2d set_y(int y)
    {
        y_ = y;
        return *this;
    }

    void print()
    {
        std::cout << x_ << ',' << y_ << '\n';
    }

private:
    int x_, y_;
};

int main()
{
    Point2d p;
    p.set_x(1).set_y(2);
    p.print();
    return 0;
}
```

Return type is Point2d and not Point2d&

Explain!

Returning reference to member variable

Of course you can also return a reference to a member (yes I know I've already talked about this before).

```
class Point2d
{
public:
    int & get_x()
    {
        return x_;
    }

    void print()
    {
        std::cout << x_ << ',' << y_ << '\n';
    }

private:
    int x_, y_;
};

int main()
{
    Point2d p;
    p.print();
    int & a = p.get_x();
    a = 0;
    p.print();

    return 0;
}
```

But remember about information hiding. Clients usually should not have direct access to instance variables. The class (and the objects) should provide services to clients through methods and hide internal implementation from outsiders. This makes the software more maintainable.

And you definitely cannot do this in a method:

```
class Point2d
{
public:
    int & get_x()
    {
        int x = x_;

        return x; // ERROR! Returning reference to
                  // local variable!!! Arghh!!!
    }

private:
    int x_, y_;
};
```

Temporary object values

Most of the time we have objects with names:

```
Date date(1970, 1, 1);
date.print();
```

Of course you can create object values not bound to names:

```
Date(1970, 1, 1).print();
```

(I've already talked about this.)

In the second case above, we have a temporary object, or an object value that does not belong to an object name.

In the `Int` class, if you have operator+ returns object values, you might see temporary objects:

```
Int i(5), j(7), k (8);
Int z = i + j + k; // i + j is temporary object
                   // this temporary object is
                   // added with k to produce
                   // another temporary object
```

Of course, just like a method, a function can return an object value

```
Date rand_date()
{
    return Date(rand()%10 + 1,
                rand()%10 + 1, rand());
}
```

Of course this is different from returning a reference:

```
Date & Date::f()
{
    return (*this);
}

int main()
{
    Date date(1, 1, 1970);
    date.f(); // A reference to date
              // is returned
}
```

A temporary object can be destroyed immediately after the expression containing the temporary object is computed.

There is an exception … you can create an object name and bind it to a temporary object value. In this case the temporary object value is not destroyed immediately. For instance

```
int main()
```

```
{
    Date date1 = rand_date();
    ...
}
```

The value of `date1` is actually the same as the value of the object returned by `rand_date()`, i.e., there is no copy constructor call. To verify this add some print statements:

```
Date randdate()
{
   Date ret(rand() % 10 + 1,
            rand() % 10 + 1, rand());
   std::cout << &ret << '\n';
   return ret;
}

int main()
{
   Date date1 = f();
   std::cout << &date1 << '\n';
}
```

You can also insert print statements in the `Date` destructor to verify that there is only one destructor call, not two, i.e., in the above, there is really only one `Date` value.

```
rand_date();          // Date value returned by
                      // rand_date() is not bound to a
                      // name. It's destroyed.
                      // Destructor is called.
Date date1 = rand_date(); // Date value bound to
                          // date1. Destructor not
                          // called yet
```

In the above example:
   Date date1 = rand_date();
the destructor is called only when the name `date1` goes out of scope.

Here are the scenarios you should study and be familiar with:

```
Date Date::f(...) { …; return *this; }
Date & Date::f(...) { …; return *this; }
Date Date::f(...) { …; Date x; return x; }
Date & Date::f(...) { …; Date x; return x; }  (error)
Date Date::f(...) { …; return Date(1,1,1970); }
Date & Date::f(...) { …; return Date(1,1,1970); }
                                             (error)
(date1.f()).print()
Date date2 = date1.f()
Date & date2 = date1.f()
```

Avoid similar names

```
int x_ = 0;

int m2()
{
    return 5;
}

class C
{
public:
    void m1()
    {
        x_ = m2();
    }
    void m2()
    {
        return 0;
    }
private:
    int x_;
};
```

which x_?

which m2?

Arrghh! Global names and members have the same name!!!

Avoid such ambiguities! What's this about …

```
class C
{
public:
    void m(int x_)
    {
        std::cout << x_;
    }
private:
      int x_;
};
```

which x_?

Yikes!!! Instance variable and parameter have the same name!!!

In general here's how C++ search for names in the above cases: first local variables are searched, then instance variables (member variables), and then globals.

Notes

In some programming languages, instead of `this` (in a method) which points to the object invoking the method, in other programming languages there's a `self` or `me` in a method this plays the same role as `this` except that it is a reference to the object invoking the method.

Forcing a reference parameter to be constant

Recall: When writing a function, it’s a good idea to use `const` for arguments that should not be changed. This is the case for both (regular) functions and methods of a class:

```
// function
void f(const C & c) { ... }

// method in class C
void C::g(const C & c) { ... }
```

The following example shows 4 possible ways of pass-by-reference:

- Non-const to non-const &
- Non-const to const &
- Const to non-const &
- Const to const &

Which ones work and which do not? Here's an experiment that will help.

```
void f(const int & i) {}
void g(int & i) {}

int main()
{
    int i = 0;
    const int j = 0;
    f(i); // nonconst passed to const reference
    f(j); // const passed to const reference
    g(i); // nonconst passed to nonconst reference
    g(j); // const passed to nonconst reference

    return 0;
}
```

First figure out on your own which is reasonable. Compile this and read the error message. Correct and check by compiling the program.

Answer:

• Non-const to non-const &	OK
• Non-const to const &	OK
• Const to non-const &	BAD!!!
• Const to const &	OK

Why?

```
void g(int & i) {} // i not constant
                   // g() can change i

int main()
{
    const int j = 0; // j cannot be changed
    g(j);            // oops ... now what?
}
```

The case of object parameters is the same.

The following example shows 4 possible ways of pass-by-reference:

- Non-const to non-const &
- Non-const to const &
- Const to non-const &
- Const to const &

Which ones work and which do not?

```
class C {};

void f(const C & y) {}
void g(C & y) {}

int main()
{
    C a;
    const C b = C();
    f(a);
    f(b);
    g(a);
    g(b);

    return 0;
}
```

Answer:

• Non-const to non-const &	OK
• Non-const to const &	OK
• Const to non-const &	BAD!!!
• Const to const &	OK

Why?

```
class C{};

void h(C & x) {} // h can modify x

int main()
{
    const C y = C(); // y cannot be modified
    h(y); // oops ... now what???
}
```

In summary if `T` is any type (`int` , `char` , …, some `class`, …), and `f()` has the following prototype:

```
      void  f(T & x);
```

and y is a ***constant*** `T` object, then the following is incorrect:

```
const T y;
f(y);
```

```
class C {public: int x};
void f0(const C & y) {}
void f1(C & y) {}
int main()
{
    C a;
    f0(a); f1(a);
    return 0;
}
```

Reminder:
`f0()` cannot change `y`
`f1()` can change `y`

Example:

```
class IntPointer
{
public:
    ...
    IntPointer(const IntPointer & a)
        : p_(new int)
    { *p_ = *(a.p_); }
      ...
private:
    int * p_;
};
```

The member variable of `a` is not changed so we make `a` constant

Example:

```
class Int
{
public:
    Int(int x)
        : x_(x)
    {}

    Int max(Int & a)
    {
        if (x_ < a.x_)
            return a;
        else
            return (*this);
    }

private:
    int x_;
};
```

This method does not change **a**.

So

```
...
class Int
{
public:
    Int(int x)
        : x_(x)
    {}

    Int max(const Int & a)
    {
        if (x_ < a.x_)
            return a;
        else
            return (*this);
    }

public:
      int x_;
};
...
```

Example:

```
class C
{
public:
    void f(int a)
    {
        x_ = a;
    }
    int g(int a)
    {
        return y_ + a;
    }

private:
    int x_, y_;
};

void j(C & c)
{
    c.f(0);
}

void k(C & c)
{
    c.g(0);
}

int main()
{
    C c, d;
    j(c);
```

Which function param can be made const?

```
    k(c);
    return 0;
}
```

Example. Invoking `c.f()` will modify `c`

```
    class C
    {
        ...
        void f(int a){ x_ = a; } // modifies x_
        ...
    };
```

This means that the following will ***not*** work:

```
      void j(const C & c) { c.f(0); }
```

Invoking `c.g()` will not modify `c`

```
class C
{
    ...
    int g(int a)
    {
        return y + a;
    }
    ...
};
```

This means that the following ***will*** work:

```
      void j(const C & c) { c.f(0); }
```

Exercise. Which parameters in `j,k` can be made constant?

```
class C
{
public:
    void f(int a)
    {
        x_ = a;
    }
    int g(int a)
    {
        return y_ + a;
    }
    void h(C & c)
    {
        f(c.x_);
    }
    void i(C & c)
    {
        c.x_ = g(5);
    }

private:
    int x_, y_;
};

void j(C & c, C & d)
{
    c.h(d);
}

void k(C & c, C & d)
{
    c.i(d);
}

int main()
{
    C c, d;
    j(c, d);
    k(c, d);
    return 0;
}
```

Exercise. Which parameter should be made `const`?

```
class Int
{
public:
    Int(int x0)
        : x_(x0)
    {}

    void incrementBy(Int & a)
    {
        x_ += a.x_;
    }

    void increment(Int & a)
    {
        a.x_ += x_;
    }

    int get()
    {
        return x_;
    }

private:
    int x_;
};

int main()
{
      Int a(1), b(5);
      a.incrementBy(b);
      a.increment(b);
      std::cout << a.get() << ' ' << b.get() << '\n';
      return 0;
}
```

Forcing a return reference to be constant

Recall this example from above:

```
class Point2d
{
public:
    int & get_x()
    {
        return x_;
    }

    void print()
    {
        std::cout << x_ << ',' << y_ << '\n';
    }

private:
    int x_, y_;
};

int main()
{
    Point2d p;
    p.print();
    int & a = p.get_x();
    a = 0;
    p.print();

    return 0;
}
```

The `p.get_x()` method returns a reference to `p.x_`. This reference is then given to `a`:

```
    int & a = p.get_x();
```

This means that `a` is a reference to `p.x_`. Therefore changing `a` will mean changing `p.x_`.

But remember what I said earlier about information hiding. Clients usually should not have direct access to instance variables. The class (and the objects) should provide services to clients through methods and hide internal implementation from outsiders. This makes the software more maintainable. So to prevent `main()` from changing `p.x_` directly, you can do this:

```
class Point2d
{
public:
    const int & get_x()
    {
        return x_;
    }
...
```

The reference returned will not be able to change the `x_` of the object. In fact, now the `main()` above cannot be compiled. Why?

```
class Point2d
{
public:
    const int & get_x()
    {
        return x_;
    }
...
};

int main()
{
    Point2d p;
    p.print();
    int & a = p.get_x();
    ...
}
```

Because the `a` in `main()` is a reference that can potentially attempt to the the value of `p.x_`. The issue is not whether `a` will actually change or not. The only way for the above to compile is to make it impossible for a to change, i.e., you would have to do this:

```
class Point2d
{
public:
    const int & get_x()
    {
        return x_;
    }
...
};

int main()
{
    Point2d p;
    p.print();
    const int & a = p.get_x();
    ...
}
```

How to make `*this` constant?

But what about `this`? It's passed in automatically and is not shown in the parameter list! How do you "`const`" `this`?

```
class Int
{
public:

    Int(int x)
        : x_(x)
    {}

    Int max(const Int  & a)
    {
        if (x_ < a.x_)
            return a;
        else
            return (*this);
    }

private:
    int x_;
};
```

`a` is not changed so we made it constant

But ….

```
class Int
{
public:
    Int(int x)
        : x_(x)
    {}

    Int max(const Int & a)
    {
        if (x_ < a.x_)
            return a;
        else
            return (*this);
    }

private:
    int x_;
};
```

But `this` is also not changed. How do we make it constant?

Do this:

```
class Int
{
public:
    Int(int x)
        : x_(x)
    {}
    Int max(const Int & a) const
    {
        if (x_ < a.x_)
            return a;
        else
            return (*this);
    }

private:
    int x_;
};
```

`const` after function header will make `*this` constant in the method

Example:

```
class Date
{
public:
    int get_d() const;

private:
    int yyyy_, mm_, dd_;
};

int Date::get_d() const
{
    return dd_;
}
```

Makes *`this` constant in the body of the function.

Exercise. Correct the error(s)!

```
class Date
{
public:
    int get_d() const
    {
        return dd_;
    }

    void set_d(int dd) const
    {
        dd_ = dd;
    }
private:
    int yyyy_, mm_, dd_;
};
```

WARNING:

```
class C
{
public:
    void f() const
    {}

    void g() const;
};

void C::g() const
{}
```

Note: This is for inlined method.

Note: For non-inlined method. `const` must appear in the method prototype ***and*** the method definition!!!

Go ahead and experiment with this:

```
class C{
public:
    void f() const
    {}

    void g() const;
};

void C::g() ~~const~~
{}

int main()
{}
```

Experiment: Try compiling this without `const`. Read the error message.

Exercise. Which methods should be constant?

```
class Int
{
public:
    Int(int x)
        : x_(x)
    {}

    void incrementBy(Int & a)
    {
        x_ += a.x_;
    }

    void increment(Int & a)
    {
        a.x_ += x_;
    }

    int get()
    {
        return x_;
    }
private:
    int x_;
};

int main()
{
    Int a(1), b(5);
    a.incrementBy(b);
    a.increment(b);
    std::cout << a.get() << ' ' << b.get() << '\n';
    return 0;
}
```

Functions with `const` at the end are called **constant member functions**. In other words, constant member functions are methods that will not modify the object invoking the method.

Exercise. Which statement in `main()` is incorrect? Next, Compile this and read the error message. Correct and check by compiling the program.

```
class C
{
public:
    void m1()
    {}

    void m2() const
    {}
};

int main()
{
    const C a;
    C b;
    a.m1();
    a.m2();
    b.m1();
    b.m2();
    return 0;
}
```

Exercise. Which methods should be constant? Which parameters should be constant?

```
class C
{
public:
    void m(C & a, C & b)
    {
        x_ = a.x_;
        y_ = b.y_;
    }

    void m(C & a, C & b)
    {
        x_ = a.x_;
        b.y_ = y_;
    }

    void m(C & a, C & b)
    {
        a.x_ = x_;
        y_ = b.y_;
    }

    void m(C & a, C & b)
    {
        a.x_ = x_;
        b.y_ = y_;
    }

private:
    int x_, y_;
};
```

Exercise. Go over ***all*** the previous classes you have built (`Date`, `Vehicle`, `WeatherCtrl`, `Int`, `IntPointer`, `IntArray`, `IntDynArr`, etc.) Make all relevant parameters constant whenever possible. Make all methods constant whenever possible.

Exercise. Does this compile?

```
class C
{
public:
    void m()
    {}

    void m() const
    {}
};
```

Exercise. Look at the `IntPointer` class very carefully. In particular look at the de-reference operator, `operator*()`:

```
class IntPointer
{
public:
    ...
    int & operator*()
    {
        return *p_;
    }
    ...
};
```

Should `operator*()` be constant? The question to ask is this: is `p` changed? The important point … There is a difference between asking if `p` should be constant and if `*p` should be constant.

Exercise. Continuing the above idea, look at `IntArray` and `IntDynArr`. There is a similar issue for `operator[](int)`. Should it be a constant operator?

Summary

- When an object `obj` invokes a method `m`, a pointer called `this` is in the scope of `m`. The `this` pointer points to `obj`. In other words, in a method, the `this` pointer always point to the object invoking the method. (See section on static methods for exception.)

- In the body of a method, if an object `obj` has instance variable `x_`, then the following are the same:

  `x_` is the same as `this->x_`

- In the body of a method, if an object `obj` has another method called `m`, then the following are the same:

  `m()` is the same as `this->m()`

- A method can return a copy of the value of the object of class `C`:
  ```
  C C::m()
  {
      ...
      return *this;
  }
  ```
  or a reference to itself:
  ```
  C & C::m()
  {
      ...
      return *this;
  }
  ```

- If a method returns a reference to itself, then you can chain up methods:
  ```
        obj.m().m1()
  ```
  where `m1` is another method and `m1` is also invoked by `obj`.

- WARNING: If `m` does not return a reference of the invoking object, then in the following:
  ```
        obj.m().m1()
  ```
  `m1()` is not invoked by `obj`, but rather by a copy of `obj`.

- A constant method is a method that cannot modify the object invoking the method.

- A method is made constant by putting the word `const` on the right of the method header. This is done in the header file and if the method is defined outside the class definition, then it must also be done in the definition of the method.