CPP301-C

Module 5

Additional C++ Language & Library Features

1 Move Semantics

1.1 Lvalues & Rvalues

A l-value can represent the left-sided value in an assignment operation. It is basically a value that has a permanent or semi-permanent location that can be used to access the existing value or modify the value at that location.

Variables are l-values: move1

```
int a;  // variable a is a l-value

int main() {
    printf("a=%ld\n", a);    // access a
    a = 1;                   // modify a
}
```

A function can also be an l-value if it returns the location of the original value rather than just a copy of the value. You can return by reference or by pointer. Both of the following functions are l-values.

Functions can be l-values:

```
int& b() { return a; }      // return reference of a
int *c() { return &a; }     // return pointer to a

int main() {
    b() = 2; printf("b()=%ld\n", a);    // modify a using reference from b()
    *c()= 3; printf("*c()=%ld\n", a);   // modify a using pointer from c()
}
```

Still some l-values cannot be modified if it's marked as a constant. Thus they can only be used on the right-side in an assignment like a r-value rather than on the left-side. The following function cannot be used on the left-side.

Constants cannot be l-values:

```
const int& d() { return a; }    // return a constant reference of a

int main() {
    d() = 4;                    // this statement cannot be compiled
    printf("d()=%ld\n", d());   // this statement can be compiled
}
```

A r-value represents a temporary value which may also be a copy of a l-value. If you pass something by value or return something by value instead of by references or by pointers, it becomes a r-value. Both the following functions are r-values.

Functions that are r-values:

```
int e() { return 123; }     // return a copy of 123
int f() { return a; }       // return a copy of a's value

int main() {
    e() = 5;    // this statement cannot be compiled
    f() = 6;    // this statement cannot be compiled
    printf("e()=%ld", e()); // this statement can be compiled
    printf("f()=%ld", f()); // this statement can be compiled
}
```

1.2 Copy Constructor

To support passing and returning objects by value, every class will have, by default, a copy constructor. A copy constructor is a constructor that accepts a reference to an object of the same type. Usually the reference will also be constant as we only access the original object but we never modify it. Only the copy of the object is modified. The default implementation will copy all the member variables. This can be a problem if a member variable is pointing to dynamically allocated resource/memory. So instead of making a copy of the resource/memory, it only makes a copy of the pointer/handle. So if the destructor releases the dynamic resource/memory, the copy will not work if the original is destroyed and the original will not work if the copy is destroyed. Thus it is necessary for you to implement a custom copy constructor to resolve this problem. In the following example, we will implement a **valueable** template-class that stores a value in dynamic memory. The object keeps a pointer to the memory that stores the value rather than storing the value itself.

A class to create objects that store a value in dynamic memory:

```
template<typename T>
class valueable {
private:
    T *m_pValue;
public:
    valueable() : m_pValue(nullptr) { }
    valueable(T value) { m_pValue = new T; *m_pValue = value; }
    ~valueable() { if (m_pValue) delete m_pValue; }
};
```

We will also add an assignment operator to make it easy to change the value and a type operator to make it very easy to access the value. Together with the assignment constructor, the object is compatible to the type of a value. It is also good to provide a function to check if the object does contain a value before accessing the value since the pointer can be null.

Assignment operator:

```
valueable& operator =(T value) {
    if (m_pValue == nullptr) m_pValue = new T;
    *m_pValue = value; return *this;
}
```

Type operator:

```
operator const T&() const {
    if (m_pValue == nullptr)
        throw std::exception("Value is null.");
    return *m_pValue;
}
```

Function to check if object has a value:

```
bool has_value() const {
    return m_pValue != nullptr;
}


void show_value(const valueable<int>& v) {
    if (v.has_value()) {
        int n = v;  // object is compatible with int
        printf("value = %ld\n", n);
    }
}

int main() {
    valueable<int> v1;          // object without value
    valueable<int> v2(123);     // object with value 123
    valueable<int> v3 = 999;    // int 999 is compatible with object

    show_value(v1); // no value
    show_value(v2); // 123
    show_value(v3); // 999

    return 0;
}
```

Because of assignment operator and type operator, the above objects are compatible to **int** and thus you can perform **int** operations with the objects. The type operator is used to convert the object to **int** and the assignment operator is used to store an **int**. The assignment constructor converts **int** to an object. Anywhere you can use **int**, you can use the object. Anywhere you use the object, you can use an **int**.

Treating objects as though they are plain int l-values

```
v2 = v2 + 17; show_value(v2);
v3 = v3 - 99; show_value(v3);
```

Treating an int as the object:

```
show_value(666);
```

A temporary object will be automatically created to store the value. The object will be destroyed at the end of the function. Even though so far **valueable** class seems to be working, we can easily crash it using the following code.

Crashing valueable class:

```
int main() {
    valueable<int> *v1 = new valueable<int>(999);
    valueable<int> *v2 = new valueable<int>(*v1);

    show_value(*v1); delete v1;
    show_value(*v2); delete v2;

    return 0;
}
```

The issue with the above code is that v2 is a copy of v1. The default copy constructor only copies the **m_pValue** pointer from v1 to v2. Thus both objects are pointing to the same dynamic memory that stores the value. Changing v1 value will also change v2 value and vice-versa. Destroying v1 value destroys v2 value and vice-versa. Easy way to resolve this is to disable or delete the copy constructor since the default copy constructor generated by the compiler does not work. Even if you don't declare it as shown below, the compiler will still generate it.

Default copy constructor:

```
valueable(const valueable<T>& obj) = default;
```

Disable the copy constructor:

```
valueable(const valueable<T>& obj) {
    throw new std::exception("Valueable cannot be copied.");
}
```

Delete the copy constructor:

```
valueable(const valueable<T>& obj) = delete;
```

When the copy constructor is deleted, you cannot even compile the code that uses it because it no longer exists. If we want to copy, we don't need to copy the object but copy the object's value. We don't need the copy constructor to do this.

Copy the value from an existing object into a new object:

```
valueable<int> *v1 = new valueable<int>(999);
valueable<int>* v2 = new valueable<int>((int)(*v1));
```

Alternatively, implement your own custom copy constructor that works by not simply copying the **m_pValue** pointer but allocate new dynamic memory to store the value in the new object. Changing the original will no longer change the copy and vice-versa and destroying the original will not affect the copy and vice-versa.

A working copy constructor for valueable class:

```
valueable(const valueable<T>& obj) : m_pValue(nullptr) {
    if (obj.m_pValue != nullptr) {
        m_pValue = new T;   // allocate dynamic memory for the copy
        *m_pValue = *obj.m_pValue;  // store a copy of the original value
    }
}
```

Changing the stored values:

```
*v1 = *v1 + 1;  // updating v1 value does not affect v2 value
*v2 = *v2 - 1;  // updating v2 value does not affect v1 value
```

Normally dynamic objects/memory are accessed through pointers but pointers can be easily converted to references and references can be converted to pointers. No copy is made when you perform these conversions, you are always accessing the original object either by reference or by pointer.

Converting pointers to references

```
valueable<int>& r1 = *v1; show_value(r1);
valueable<int>& r2 = *v2; show_value(r2);
```

Converting references to pointers

```
valueable<int> *p1 = &r1; show_value(*p1);
valueable<int>* p2 = &r2; show_value(*p2);
```

1.3 Move Constructor

A problem with the copy constructor is that it is not efficient for making a copy of a temporary object that only lasts for a short amount of time. For example, the function belows return a local object by value. You cannot return by reference or by pointer a local object because that object no longer exists at the end of the function so a copy has to be returned instead. This is a r-value function.

Return by value:

```
valueable<int> get_valueable(int n) {
    valueable<int> v(n);
    return v;
}
```

```
int main() {
    auto v = get_valueable(666);    // getting a copy
    show_value(v);                  // accessing the copy
}
```

To prove that the copy constructor is called, add a statement to display a message in the copy constructor. The message will appear when the constructor is called. This will provide that the **get_valueable** function returns a copy.

Testing to see when copy constructor called:

```
valueable(const valueable<T>& obj) : m_pValue(nullptr) {
    if (obj.m_pValue != nullptr) {
        m_pValue = new T;
        *m_pValue = *obj.m_pValue;
    }
#ifdef _DEBUG
    puts("Copy constructor called.");
#endif
}
```

If we know the original is going to be destroyed, why not simply move the pointer in the original object over to the copy so the copy doesn't need to allocate new dynamic memory. We will then invalidate the pointer in the original object so destroying it will not affect the copy. This can be done by implementing a move constructor. Following is the signature of a move constructor, notice that we use (&&) double referencing to indicate a move constructor and the original object must not be passed as constant because we need to modify the original object in a move constructor.

Implementing a move constructor:

```
valueable(valueable<T>&& obj) : m_pValue(obj.m_pValue) {
    if (m_pValue) obj.m_pValue = nullptr;
#ifdef _DEBUG
    puts("Move constructor called.");
#endif
}
```

The C++ compiler is intelligent to know when an object is to be destroyed as long as it is not a dynamic object. Thus when you make a copy of the object, it would use the move constructor instead of the copy constructor. If you run the program again, C++ will now use the move constructor instead so the dynamic memory is only allocated and released one time. The original object allocates the memory but the memory will be released by the copy. Move semantics makes usage of r-values more efficient and more performant. Copy and move constructors are not used for l-values.

1.4 C++ Library Move Support

The standard C++ library provides some functions that support move semantics. You can use **swap** function to exchange the pointers between the original and the copy. The pointer to the dynamic value will now in the copy, and the nullptr in the copy will now be in the original.

Swap function from the C++ standard library:

```
valueable(valueable<T>&& obj) : m_pValue(nullptr) {
    std::swap(m_pValue, obj.m_pValue);
            :
```

Sometimes the compiler cannot determine that you no longer want to use the original object after you made a copy so it calls the copy constructor instead of move. Below is an example where v2 is a copy of v1 so both objects have the same value.

Copy instead of move:

```
valueable<int> v1(66);
valueable<int> v2(v1);
if (v1.has_value()) printf("v1=%d\n", (int)v1);
if (v2.has_value()) printf("v2=%d\n", (int)v2);
```

Use the **move** function to cast the original to a double-reference (&&). This will then trigger the move constructor instead. The example below shows v1 is moved to v2 so v2 has the value and v1 no longer has the value.

Move instead of copy:

```
valueable<int> v2(std::move(v1));
```

1.5 No Exception

When you throw an exception, it is attached to the function call stack and the current function terminates. This allow the exception to propagate up the call stack so even if no function catches and handles the exception, eventually it reaches the C++ runtime startup code which will popup a message box to show the error. If you know that no exception will occur when a certain function is called, attach a **noexcept** attribute to the method. Calling the function will be more efficient and faster as the caller function doesn't have to keep checking the call stack everytime the function returns to look for an exception. The move constructor must not result in an exception. A failed moved process may lead to memory leaks causing the program to crash.

Move constructor should not cause exceptions:

```
valueable(valueable<T>&& obj) noexcept : m_pValue(nullptr) {
            :
```

2 Function Callbacks

2.1 Event System

.NET has an event subscription system where multiple methods can subscribe to an event. When the event is triggered, all methods will be executed in the subscription order. We can implement a system similar to it in C++ using object-oriented features. First we implement an **EventListener** that contains a virtual **Notify** function that will be triggered by an event. We have to implement the < operator so that listeners can be stored in a C++ container.

Class for objects that subscribe to events: event1

```
#include <cstdio>
#include <set>

template<typename T>
class EventListener {
public:
    virtual void Notify(T event) = 0;
    friend bool operator < (EventListener<T>& obj1, EventListener<T>& obj2) {
        return true;    // function order is not important
    }
};
```

We now implement the **Event** where you can add or remove any number of listeners. When you **Invoke** the event, all the listeners will be notified.

Class for events to subscriber to:

```
template<typename T>
class Event {
private:
    std::set<EventListener<T>*> m_listeners;
public:
    void Add(EventListener<T>& listener) { m_listeners.insert(&listener); }
    void Remove(EventListener<T>& listener) { m_listeners.erase(&listener); }
    void Invoke(T value) {
        for (auto listener : m_listeners)
            listener->Notify(value);
    }
};
```

We will now implement an event subscriber that can listen for both a **const char *** and **int** event. We can then create the events, add the subscriber and then trigger the event. The appropriate **Notify** function in alll the subscribers will be called.

Class for event subscribers:

```
class EventSubscriber :
    public EventListener<const char *>,
    public EventListener<int> {

public:
    void Notify(const char* event) {
        printf("%s event received.\n", event);
    }
    void Notify(int event) {
        printf("int (%d) event received.\n", event);
    }
};
```

Testing the event system:

```
int main() {
    Event<const char*> e1;
    Event<int> e2;

    EventSubscriber s1;
    EventSubscriber s2;
    e1.Add(s1);
    e2.Add(s1);
    e1.Add(s2);
    e2.Add(s2);


    e1.Invoke("Hello!");
    e1.Invoke(123);
}
```

2.2 C++ Function

Rather than implement custom classes as subscriber objects, subscribers can just be function objects. C++ provides a **function** feature that encapsulates a function as an object so when the object is activated using the call operator (), the function is called. To support adding function objects to a C++ container, you will need to implement a comparer class rather than overload the < operator.

Header file to include for function objects:

```
#include <functional>
```

Comparer for function objects:

```
template<typename T>
class FuncComparer {
public:
    bool operator()(
        const std::function<void(T)>& obj1,
        const std::function<void(T)>& obj2) const {
        return true;    // function order is not important
    }
};
```

We now replace EventListener with function objects:

```
template<typename T>
class Event {
private:
    std::set<std::function<void(T)>,FuncComparer<T>> m_listeners;
public:
    void Add(std::function<void(T)> listener) { m_listeners.insert(listener); }
    void Remove(std::function<void(T)> listener) { m_listeners.erase(listener);
}
    void Invoke(T value) {
        for (auto listener : m_listeners)
            listener(value);
    }
};
```

The following is the test code to subscribe functions to events. Lambda functions can be also used as demonstrated below. When the event is triggered, the functions will be called.

Subscribing functions to events:

```
std::function<void(const char*)> f1 = [](const char* event) {
    printf("%s event received.\n", event);
};
std::function<void(int)> f2 = [](int event) {
    printf("int (%d) event received.\n", event);
};
e1.Add(f1);
e2.Add(f2);
```

2.3 C++ Bind

The above demonstrates calling global and static functions. It is possible to call object functions as well as long as the function is binded to the member functions and object as shown. However you cannot call overloaded functions so each member name has to be unique.

The following show adding of unique member functions to **EventSubscriber** class. Of course they can call overloaded member functions as well.

Adding unique member functions to subscriber class:

```
class EventSubscriber :
    public EventListener<const char *>,
    public EventListener<int> {

public:
    void Notify(const char* event) {
        printf("%s event received.\n", event);
    }
    void Notify(int event) {
        printf("int (%d) event received.\n", event);
    }
    void Notify1(const char* event) { Notify(event); }
    void Notify2(int event) { Notify(event); }
};
```

You can call **bind** function in the C++ library to create a function object that is bound to an object member function. The event data can binded together with the function so you do not need to pass the data when the function is called.

Binding and calling object member functions:

```
EventSubscriber s1;
auto f3 = std::bind(&EventSubscriber::Notify1, s1, "Goodbye!");
auto f4 = std::bind(&EventSubscriber::Notify2, s1, 999);
f3();
f4();
```

If you wish to pass the event data only during the function call, you can replace the data with placeholders, **_1**, **_2**, **_3** etc from the **std::placeholders** namespace. You can import the namespace to directly refer to the placeholders.

Namespace to import for placeholders:

```
using namespace std::placeholders;
```

Passing data during function calls:

```
auto f3 = std::bind(&EventSubscriber::Notify1, s1, _1);
auto f4 = std::bind(&EventSubscriber::Notify2, s1, _1);
f3("Goodbye!");
f4(999);
```

Binded functions can also be added to C++ containers so they can also be subscribed to events. When the events is triggered, the object member functions are called. The only issue is you cannot mix static functions and object functions into the same C++ container.

Add either static functions or object functions:

```
// e1.Add(f1);
// e2.Add(f2);

e1.Add(f3);
e2.Add(f4);

e1.Invoke("Hello!");
e2.Invoke(123);
```

This chapter demonstrates how to form dynamic connections between functions that calling one function can trigger other static or object functions without directly calling the function by name. We have already seen how to call functions by using pointers but this example demonstrates how to call virtual functions and through C++ function objects.