CSCI251/CSCI851 Spring-2021 Advanced Programming (**S5d**)

Moving

Outline

- Move constructors.
- Introducing: lvalue and rvalue.
- Moving.

Copying/moving constructor/assignment

- These are to do with initialising an object from another of the same type …

Special Member Function	**For Class X;**
Copy constructor	`X(const X&);`
Copy assignment	`X& operator=(const X&);`
Move constructor (C++11)	`X(X&&);`
Move assignment (C++11)	`X& operator=(X&&);`

Move constructor (C++11): `X(X&&);`

- We use move instead of copy when we want to destroy the thing being copied immediately.
- It's particularly useful if we have resources that cannot be shared, even temporarily and so have to be transferred from one to the other.
  - Move can be thought of as changing ownership.
- Move can be significantly more efficient than a copy and delete and it's a major reason why C++11 STL containers are improved vs. C++98/C++03.

`X(X&&);` ???

- What's with the strange `&&`?
- This corresponds to logical AND, but is also a new kind of reference introduced in C++11.
- It's referred to as an rvalue reference.
- We didn't talk about lvalue and rvalue earlier, we need to know about them now.

Introducing: lvalue and rvalue

- Left value: lvalue.
- Right value: rvalue.
- The names derive from the C use.
  - In C, lvalues could stand on the left-hand side of an assignment, rvalues could not.
- The textbook notes: “In C++, the distinction is less simple.”

- Continuing from the textbook (page 135):

- “In C++, an lvalue expression yields an object or a function.
- However, some lvalues, such as const objects, may not be the left-hand operand of an assignment.
- Moreover, some expressions yield objects but return them as rvalues, not lvalues.
- Roughly speaking, when we use an object as an rvalue, we use the object’s value (its contents).
- When we use an object as an lvalue, we use the object’s identity (its location in memory).”

https://docs.microsoft.com/en-us/cpp/cpp/rvalue-reference-declarator-amp-amp?view=msvc-160

- In the context of `move`, the particularly important characteristic of an rvalue reference is that they can only be bound to objects that are about to be destroyed.
- This includes objects such as literals which as in expressions to be acted with and then removed.

```
int integer = 10;
int &ref = integer;
int &&rref = integer;
```

😐

☺

☹

Error: Rvalue reference 'int&&' cannot cannot be bound to an lvalue.

This is the actual error message, with cannot twice! (-std=c++11)

- So how does the move constructor work?
- It makes use of a standard library function `move` that converts, or casts, from an lvalue to an rvalue reference.
- The function `move` is part of the standard namespace but is found in the header `utility`.

https://en.cppreference.com/w/cpp/utility/move

```
#include <iostream>
#include <utility>
using namespace std;

int main()
{
        int integer=10;
        int &&rref=move(integer);

        cout << rref << endl;
}
```

- When we use `move` we need to be aware the value of the moved-from object is no longer reliable.
- The moved-from object needs to be safe to remove, or destructible.
- We shouldn’t used the moved-from object again other then for deleting, or possibly for assigning to.

Move assignment (C++11): `X& operator=(X&&);`

- There is an example at:
  http://en.cppreference.com/w/cpp/language/move_constructor
- … but it uses some C++14 functionality too.
- Here, as with copy assignment, we need to be careful with self assignment.
- We should check, and if it's a move self assignment we can skip the operations …

```
X &X::operator=(X &&rhs) noexcept
{
        if (this !=&rhs) { …}
        return *this;
}
```

- There is a useful suggestion at the site below, that of getting the move constructor to call the move assignment operator.

https://docs.microsoft.com/en-us/cpp/cpp/move-constructors-and-move-assignment-operators-cpp?view=vs-2019

```
MemoryBlock(MemoryBlock&& other) : _data(nullptr) , _length(0)
{
        *this = std::move(other);
}
```

That site has a full example of a move constructor and move assignment.

Move for vector : `emplace_back`

- You should all be somewhat familiar with populating a vector using `push_back`.
  - There is also the operation `insert`, and for other containers `push_front`.
- A more efficient alternative that is appropriate for a composite relation, the added object being fully committed to the container, is the `emplace_back`.
  - The other replacement operations are `emplace_front` and `emplace`.

- Operations like `push_back` create a local temporary object that is placed into the container.
- With the emplace operations we are creating the object in the vector, so directly in the space managed by the container.