Move Demo

Bells and Whistles

The Many Uses of Classes
Move Demo
Special member functions demo

Recall the six so-called *special member functions*:

- Default constructor.
- Destructor.
- Copy constructor.
- Copy assignment.
- Move constructor.
- Move assignment.

These are automatically defined if you do nothing, but defining some of them inhibits the automatic definition of others.

Automatic definitions can be enabled by explicitly writing `=default` or disabled by writing `=delete`. 
Special member functions demo

The demo 13-SpecialMbrFcns defines all six special functions and shows how they can be invoked.

It defines a class T with two private data members: an integer x and an integer pointer a.

class T {
private:
    int x;
    int* a = new int[3];
public:
    ...
};
Default constructor and destructor

    // Default constructor
    T() : x(0), a(nullptr) {
        cout << " Null constructor" << endl;
    }

This uses a ctor to initialize the two data members to 0 and nullptr, respectively. It then announces itself.

    // Destructor
    ~T() {
        delete[] a;
        cout << " Destructor" << endl;
    }

This deleted the dynamic extension a and announces itself.
Additional constructor

```cpp
// Constructor from an int
explicit T(int x) : x(x) {
    cout << " Explicit constructor T(" << x << ")" << endl;
}
```

This initializes \texttt{x} using a ctor. \texttt{a} is initialized using the initializer \texttt{= new int[3]} defined in the class. The keyword \texttt{explicit} inhibits it from being used implicitly to convert an \texttt{int} to a \texttt{T}.
Copy constructor and move constructor

// Copy constructor
T(const T& rhs) : x(rhs.x), a(rhs.a) {
    cout << " Copy constructor" << endl;
}

Uses ctor to initialize \( x \) and \( a \) from corresponding members of \( rhs \).

// Move constructor
T(T&& rhs) : x(rhs.x), a(rhs.a) {
    if (this != &rhs) rhs.a = nullptr;
    cout << " Move constructor" << endl;
}

Same as copy constructor but prevents automatic deletion of the dynamic extension in \( rhs \) by setting \( a \) to \texttt{nullptr}.
Copy assignment

    // Copy assignment
    T& operator=( const T& rhs ) {
        x = rhs.x;
        a = rhs.a;
        cout << " Copy assignment" << endl;
        return *this;
    }

Uses operator=() to assign \( x \) and \( a \) from the corresponding members of \( \text{rhs} \). Returns a reference to the left-hand side in keeping with other assignment operators.

Why wasn’t a ctor used here?
Move assignment

```
T& operator=( T&& rhs ) {
    if (this != &rhs) {
        x = rhs.x;
        delete[] a;
        a = rhs.a;
        rhs.a = nullptr;
    }
    cout << " Move assignment" << endl;
    return *this;
}
```

Similar to copy assignment, but:

1. What is the `if`-statement for?
2. Why is `a` deleted \textit{before} the move?
3. Why is `rhs.a` set to `nullptr` \textit{after} the move?
Invoking the special functions

The main program in demo 13-SpecialMbrFcns prints a C++ statement along with output showing what happened.

[T a;]
    Null constructor
    a=(0, 0)

[T b(17);]
    Explicit constructor T(17)
    b=(17, 0x1e94030)

[T d( move(b) );]
    Move constructor
    d=(17, 0x1e94030), b=(17, 0)
Invoking the special functions

[T e;]
Null constructor

[T f;]
Null constructor

[f = move(d);]
Move assignment
f=(17, 0x1e94030), d=(17, 0)

[T g = T(41);]
Explicit constructor T(41)
g=(41, 0x1e94050)
Invoking the special functions

[T h;]
Null constructor

[h = T(89);]
Explicit constructor T(89)
Move assignment
Destructor
h = (89, 0x1e94070)

Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Bells and Whistles
Optional parameters

The same name can be used to name several different member functions if the signatures (types and/or number of parameters) are different. This is called overloading.

Optional parameters are a shorthand way to declare overloading.

Example

```
int myfun( double x, int n=1 ) { ... }
```

This in effect declares and defines two methods:
```
int myfun( double x ) {int n=1; ...}
int myfun( double x, int n ) {...}
```

The body of the definition of both is the same.
If called with one argument, the second parameter is set to 1.
**const**

`const` declares a variable (L-value) to be readonly.

```plaintext
const int x;
int y;
const int* p;
int* q;
```

```plaintext
p = &x;    // okay
p = &y;    // okay
q = &x;    // not okay -- discards const
q = &y;    // okay
```
**const implicit argument**

`const` should be used for member functions that do not change data members.

```cpp
class MyPack {
private:
  int count;
public:
  // a get function
  int getCount() const { return count; }
...}
```
Operator extensions

Operators are shorthand for functions.

Example: <= refers to the function operator <=().

Operators can be overloaded just like functions.

```cpp
class MyObj {
    int count;
    ...
    
    bool operator <=( MyObj& other ) const {
        return count <= other.count;
    }
}
```

Now can write if (a <= b) ... where a and b are of type MyObj.
The Many Uses of Classes
What is a class?

- A collection of things that belong together.
- A struct with associated functions.
- A way to encapsulate behavior: public interface, private implementation.
- A way to protect data integrity, providing world with functions that provide a read-only view of the data.
- A data type from which objects (instances) can be formed. We say the instances belong to the class.
- A way to organize and automate allocation, initialization, and deallocation of storage.
- A way to break a complex problem into manageable, semi-independent pieces, each with a defined interface.
- A reusable module.