CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 17
October 31, 2018

Overview of PS5

Challenges

PS5 is to add a second agent type to the simulated population. This creates several challenges.

1. Make `Agent` a pure abstract base class for new derived classes `Fickle` and `Crowd`.
2. Create a `Population` class to manage populations with two kinds of agents as well as two possible initial values.
3. Remove population code from `Simulator`, leaving only the code to simulate random communication steps until consensus is reached.
4. Rework `main.cpp` to accommodate the above changes.

Experiments and Observations

Once your code is running, use it to get some understanding for how the number of steps to reach consensus depends on the parameters.

Particularly interesting is to see the effect of adding a small percentage of `Crowd` agents to a population consisting primarily of `Fickle` agents. The difference should become obvious in a population of size 10,000 or so.

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 inhibit 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 17-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

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

This initializes `x` using a ctor. `a` is initialized using the initializer `= new int[3]` defined in the class. The keyword `explicit` inhibits it from being used implicitly to convert an `int` to a `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 `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 `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 *before* the move?
3. Why is `rhs.a` set to `nullptr` *after* the move?

Invoking the special functions

The main program in demo 17-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 diffent. 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.

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

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.

```
class MyPack {
private:
  int count;
public:
  int size() const { return count; }
...
};
```

Operator extensions

Operators are shorthand for functions.

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

Operators can be overloaded just like functions.

```
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`.