CFS2155 – Object-Oriented Programming – week18

A Quick Review

So far we have seen:

- how to use *constructor functions* to create *objects* from *classes*
- how we may solve problems using *event models* to model *behavior*
- how we may implement these behaviors using *methods*

A Game Example

Many games associate *game actions* with *key presses* from a console.

Let's imagine that the following actions may occur in a popular *shoot-then up* game:

- **s**hoot the weapon
- **c**hange the weapon
- **m**ove the player forward
- turn the player to the **r**ight
- turn the player to the **l**eft

For simplicity, a player in this game has unlimited ammunition.

The following code provides a *partial* model for a player within this game:

```
class Player {
public:
  // properties and constructors

  void keyPress(char key) throw (exception) {
    switch(key) {
      case 's': shoot(); break;
      case 'c': change(); break;
      case 'm': move(); break;
      case 'r': right(); break;
      case 'l': left(); break();
      default: throw exception("Invalid key press");
    } // end of switch
  } // end of method keyPress

  // action methods...

}; // end of class Player
```

When a new player is created, we must say what weapons they have access to. So we will supply our constructor function with an vector parameter.

Indexes into this vector will number or enumerate our weapons.

So, by maintaining an additional index into the weapons vector, we will always know which weapon is currently chosen by the player.

Whenever we have to alter the weapon currently chosen, we can simply increment our index to the weapon currently chosen. If we perform addition modulo the weapons vector size, then we can model cycling through a list of weapons.

This suggests that we augment the previous code with the following:

```
class Player {
public:
  vector<string> weapons;
  int currentWeapon;

  Player(vector<string> wlist) {
    weapons = wlist;
    currentWeapon = 0;
  } // end of constructor function

  void change() {
    currentWeapon =
      (currentWeapon + 1) % weapons.size();
  } // end of method change

  // other methods (an exercise!)

}; // end of class Player
```

Notice how this style of programming can be used to write C++ programs that interact directly with a user. We will see more of this latter on in the course.

Accessor Methods

So far, if we wish to see what the *current* state of an object is

eg. after or before we run a method

then we have to display the attributes of the object with *debugging* print statements.

Sometimes we would like to see a *summary* of the object's contents **without** the use of such debugging print statements.

To do this, we need to use methods that can *return* data. The returned data provides us with an alternative or summary *view* of the object.

A Simple Example

For example, with a *vector* class, we might we wish to know how many elements are currently in it.

As a result, our *summary* view of the vector object instance would be an integer representing the *size* of the data contained within the vector.

C++'s vector class provides us with a *size* method for doing just this:

```
class Example {
public:
  vector<string> list;

  Example() {
    list = vector<string>();
  } // end of constructor function

  void add(string msg) {
    list.push_back(msg);
  } // end of method add

  int messageCount() {
    return list.size();
  } // end of method messageCount

}; // end of class Example
```

Notice the following:

- *int* is placed *before* the method name, instead of the keyword *void*. This indicates that this method will return (summary) information.
- the keyword *return* is used whenever we have built the necessary (summary) information to *return* from the method.

Another Example

We want a program to input a list of numbers and then to produce different *views* of those numbers:

```
class NumberEg {
public:
  vector<int> number;

  NumberEg(vector<int> nums) {
    number = nums;
  } // end of constructor function

  // some methods...
```

```
}; // end of class NumberEg
```

Lets add in a method for calculating the sum (view) of the number list:

```
int sum() {
  int sum = 0;
  for(int nos = 0; nos < number.size(); nos++) {
    sum += number[nos];
  } // end of for-loop
  return sum;
} // end of method sum
```

A method for calculating the largest member (view) of the number list:

```
int max() throw (exception) {
  if (number.size() == 0) {
    throw exception("No numbers!");
  } // end of if-then
  int largest = number[0];
  for(int nos = 1; nos < number.size(); nos++) {
    if (largest < number[nos]) {
      largest = number[nos];
    } // end of if-then
  } // end of for-loop
  return largest;
} // end of method max
```

A method for returning numbers that are a multiple of a *given* number:

```
vector<int> multiples(int nos) {
  vector<int> list = vector<int>();
  for (int N = 0; N < number.size(); N++) {
    if (number[N] % nos == 0) {
      list.push_back(number[N]);
    } // end of if-then
  } // end of for-loop
  return list;
} // end of method multiples
```

Using Methods to Structure Code

Now we know about *accessor methods*, we may use them to structure and organise code so that it is easier to read and maintain.

For example, consider the last example. We had a search algorithm that worked its way through a list collecting all list members satisfying a *particular* test.

If we now wanted to locate prime numbers or perfect square numbers, we would need to rewrite this code *each time*.

By using *accessor methods* we may place our test *within* a method, and then simply call that method during our search.

```
class SearchEg {
public:
  vector<int> list;

  SearchEg(vector<int> numbers) {
    list = numbers;
  } // end of constructor function

  boolean test(char testType, int number) throw (exception) {
    switch(testType) {
      case 'e': return number % 2 == 0;
      case 'o': return number % 2 == 1;
      // etc.
      default: throw exception("Invalid test type!");
    } // end of switch
  } // end of method test

  vector<int> search(char testType) throw (exception) {
    if (list.size() == 0) {
      throw exception("List is empty!");
    } // end of if-then
    vector<int> result = vector<int>();
    for (int nos = 0; nos < list.size(); nos++) {
      if (test(testType, list[nos])) {
        result.push_back(list[nos]);
      } // end of if-then
    } // end of for-loop
    return result;
  } // end of method search

}; // end of class SearchEg
```

Notice here the power of using methods:

- by altering the way the code works within the *test* method, we can still *search* through our list, we just look for matching numbers (ie. those that *pass* the test) differently.