Lecture 8: Encapsulation

Introduction

Encapsulation, a fundamental object-oriented programming concept, protects data by restricting direct access to fields (by assigning private visibility) and using methods—getters (accessors) and setters (mutators)—to access or modify them. This approach ensures data integrity by enforcing controlled interactions with class attributes; thus, providing better code maintainability, flexibility, and security.

Getter Methods

Getter methods, read methods, retrieve the value of private fields. Their general syntax is:

- **Variable**: *return-type identifier*`()` `const` {*body*}
- **Array**: *return-type identifier*`(int` *idx*`)` `const` {*body*}

where *return-type* matches the field's data type, and *idx* represents an array index.

Example:
Each of the following classes represents a 24-hour clock, but they differ internally while behaving similarly externally due to encapsulation.

```
class C1
{
  private:
  //separate variables
  int hr, min, sec;
  public:
  T1() : hr(0), min(0), sec(0) {}
  int hour() const
  {
    return hr;
  }
  int minute() const
  {
    return min;
  }
  int second() const
  {
    return sec;
  }
};
```

```
class C2
{
  private:
  0: hour, 1: minute, 2: second
  int times[3];
  public:
  T2() : times{0,0,0} {}
  int hour() const
  {
    return times[0];
  }
  int minute() const
  {
    return times[1];
  }
  int second() const
  {
    return times[2];
  }
};
```

```
class C3
{
  private:
  //seconds pass midnight
  int spm;
  public:
  T3() : spm(0) {}
  int hour() const
  {
    return (spm / 3600);
  }
  int minute() const
  {
    return (spm / 60 % 60);
  }
  int second() const
  {
    return (spm % 60);
  }
};
```

Setter Method

Setter methods, write methods, modify private fields while ensuring valid values. Their general syntax is:

- **Variable**: `void` *identifier*`(`*data-type identifier*`)` {*body*}
- **Array**: `void` *identifier*`(`*data-type identifier*`,` `int` *idx*`)` {*body*}

where *data-type* matches the field's data type and *idx* represents an index. The setter method assigns a value only if it meets validity conditions.

Example:

```
class C1
{
  public:
  void hour(int val)
  {
    if(val >= 0 && val <= 23)
    {
      hr = val;
    }
  }
  void minute(int val)
  {
    if(val >= 0 && val <= 59)
    {
      min = val;
    }
  }
  void second(int val)
  {
    if(val >= 0 && val <= 59)
    {
      sec = val;
    }
  }
};
```

```
class C2
{
  public:
  void hour(int val)
  {
    if(val >= 0 && val <= 23)
    {
      times[0] = val;
    }
  }
  void minute(int val)
  {
    if(val >= 0 && val <= 59)
    {
      times[1] = val;
    }
  }
  void second(int val)
  {
    if(val >= 0 && val <= 59)
    {
      times[2] = val;
    }
  }
};
```

```
class C3
{
  public:
  void hour(int val)
  {
    if(val >= 0 && val <= 23)
    {
      spm += (val - hour()) * 3600;
    }
  }
  void minute(int val)
  {
    if(val >= 0 && val <= 59)
    {
      spm += (val - minute()) * 60;
    }
  }
  void second(int val)
  {
    if(val >= 0 && val <= 59)
    {
      spm += (val - second());
    }
  }
};
```

Subcript Operator

It is customary to use the *subscript operator* (*indexer*) to access and modify elements of an array instead of a function. It can be overloaded in three ways:

- **Readonly** (Version 1):
  `const` *`data-type`*`& operator[](int` *`idx`*`) const` (returns only variables)
- **Readonly** (Version 2):
  `data-type` `operator[](int` *`idx`*`) const`
- **Read/Write**:
  `data-type``& operator[](int` *`idx`*`)` (use only if no input restrictions apply)

where *data-type* matches the data type of the array.

Example:

```
class C1
{
  public:
  const int& operator[](int idx) const
  {
    int* val[] = {&hr, &min, &sec};
    if(idx >= 0 && idx <= 2)
    {
      return *val[idx];
    }
    throw std::out_of_range("invalid index");
  }
};
```

```
class C2
{
  public:
  const int& operator[](int idx) const
  {
    if(idx >= 0 && idx <= 2)
    {
      return times[idx];
    }
    throw std::out_of_range("invalid index");
  }
};
```

```
class C3
{
  public:
  int operator[](int idx) const
  {
    int val[] = {hour(), minute(), second()};
    if(idx >= 0 && idx <= 2)
    {
      return val[idx];
    }
    throw std::out_of_range("invalid index");
  }
};
```