Encapsulation

Introduction

Encapsulation, a fundamental object-oriented programming concept, protects data by restricting direct access to fields (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

A getter method retrieves the intended value of a private field. Its general syntax is:

Primitive Data Types

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

Abstract Data Types

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

where *return-type* typically matches th data type of the field, and *idx* represents an array index.

Example:
A 24-hour clock internally can store either three variables (or a single array) representing hour, minute, and second, or a single variable representing seconds passed midnight. Regardless of its storage, due to encapsulation, its external behavior will be identical.

```
class Clock1
{
  private:
  //separate variables
  int hr, min, sec;
  public:
  int hour() const{return hr;}
  int minute() const{return min;}
  int second() const{return sec;}
};
```

```
class Clock2
{
  private:
  //seconds passed midnight
  int spm;
  public:
  int hour() const{return (spm / 3600);}
  int minute() const{return (spm / 60 % 60);}
  int second() const{return (spm % 60);}
};
```

Setter Method

A setter method modifies the value of a private field while maintaining its validity; meaning, it prohibits the assignment of an invalid input to its field. Its general syntax is:

Primitive Data Types

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

Abstract Data Types

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

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

Example:

```
class Clock1
{
 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 Clock2
{
 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

For array objects, it is common to access and modify their elements with the *subscript operator* (*indexer*). In C++, the subscript operator can be overloaded to behave as a getter method or as both a getter and setter method if there are no data constraints. The general syntaxes are:

- **Readonly**:

  `const` ***`data-type`*** `& operator[](int` ***`idx`*** `) const` (returns only variables)

- **Read/Write**:

  `data-type` `& operator[](int` ***`idx`*** `)` (use only if no input restrictions apply)

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

```
class OrderedPair
{
 private:
 double x[2];
 public:
 const double& operator[](int idx) const
 {
  if(idx >= 0 && idx < 2) {return x[idx];}
  throw std::out_of_range("out of bound");
 }
 double& operator[](int idx)
 {
  if(idx >= 0 && idx < 2) {return x[idx];}
  throw std::out_of_range("out of bound");
 }
};
```