Ch2: Introduction to Classes

Classes, Objects, Data Members, and Member Functions

Sample Program: A Simple Bank-Account Class

.cpp source-code file

```cpp
// Creating and manipulating an Account object.
#include <iostream>
#include <string>
#include "Account.h"

int main() {
  Account myAccount; // create Account object myAccount

  // show that the initial value of myAccount's name is the empty string
  std::cout << "Initial account name is: " << myAccount.getName();

  // prompt for and read name
  std::cout << "\nPlease enter the account name: ";
  std::string theName;
  std::getline(std::cin, theName); // read a line of text
  myAccount.setName(theName); // put theName in myAccount

  // display the name stored in object myAccount
  std::cout << "Name in object myAccount is: "
    << myAccount.getName() << std::endl;
}
```

Class **Account** defined in the header **Account.h**

```cpp
// Account class that contains a name data member
// and member functions to set and get its value.
#include <string> // enable program to use C++ string data type

class Account {

public:
  // member function that sets the account name in the object
  void setName(std::string accountName) {
    name = accountName; // store the account name
  }

  // member function that retrieves the account name from the object
  std::string getName() const {
    return name; // return name's value to this function's caller
  }

private:
  std::string name; // data member containing account holder's name
}; // end class Account
```

Headers

```
#include "Account.h"
```

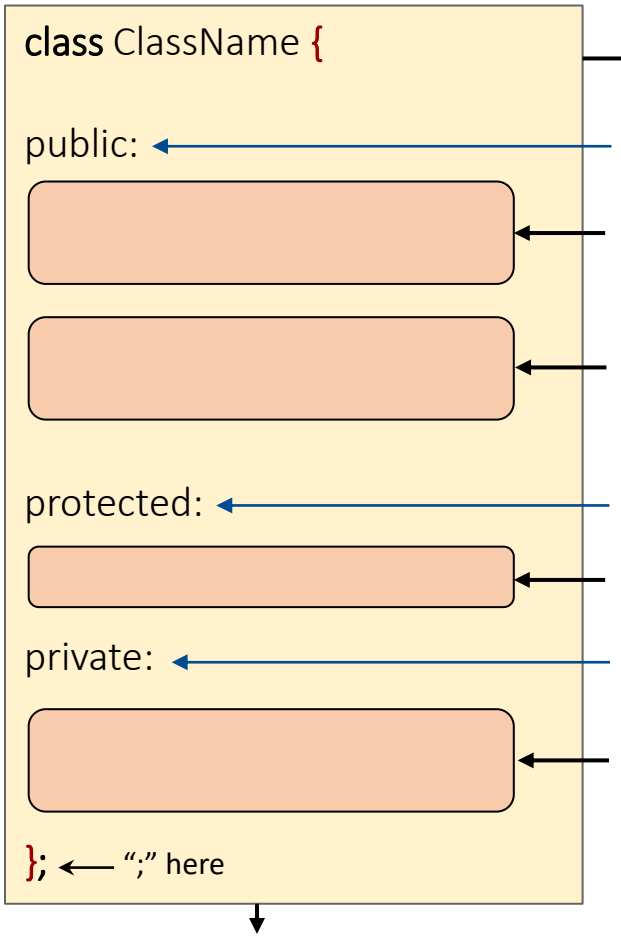
- Headers help to reduce the complexity of code and give you the benefit of reusing the classes and functions that are declared in header files to different .cpp files. File extension of headers is .h and they are included (via #include) wherever (in the source-code file or another headers) needed.

- There are two types of headers: C++ Standard Library headers or User-defined headers.

- In an #include directive, a C++ Standard Library header is placed in angle brackets <> (without .h) and a user-defined header is placed in double quotes "" (with .h). This double quotes tell the compiler that header is in the same folder as .cpp source file, rather than the C++ Standard Library.

- It is a bad programming practice to use “using directives” or “using declarations” in headers.

Structure of a Class



Class should be placed directly here or written in a header file and included here.

Access Specifier

Constructor (optional)
(for custom object initialization)

Member Functions
(Object Operations)

Access Specifier

(This will be discussed later.)

Access Specifier

Data Members
(Object Attributes)

(By convention, place data members last in the class's body.)

For reusability, place each class definition in a separate header (.h) file.

```
int main() {
  ClassName  calssObject1;
  ClassName  classObject2;
   ...
}
```

Classes cannot execute by themselves. A driver program is needed to execute a class by creation of objects of the class (instantiation).

- By convention, the class name begins with an uppercase letter, and names of member functions and data members follows the **camelCase rule**.

Data Type string & Function getline

```
std::string name;
```

```
std::string theName;
```

- This **variable declaration** creates a string variable to hold a string of characters.

- The default value for a string variable is the empty string (i.e., "").

```
std::getline(std::cin, theName);
```

- Global function getline receives a line of text from the user (std::cin), including white-space characters (i.e., a space or a tab, but not a newline), and places it in a string variable (theName).
- Note that std::cin >> theName; cannot be used because when reading a string, std::cin stops at the first white-space character. ▶

- Class string and global function getline are defined in the C++ Standard Library header <string> and belongs to namespace std. Thus, we should use std:: and define:

```
#include <string>
```

Object and Data Member

```
Account myAccount;
```

- To use a class, an **object** of the class should be first created. This process is called **instantiation**. An object is then referred to as an **instance** of its class.
- A class creates a new data type (a user-defined data type).
- You can reuse a class many times to build many objects.

```
std::string name;
```

- **Data Members** are attributes of an object, which are stored in it. The object carries these attributes with it throughout its lifetime.
- If there are many object of a class, each object has its own attributes (copy of the class's data members).
- Member functions can manipulate the attributes of each object separately.

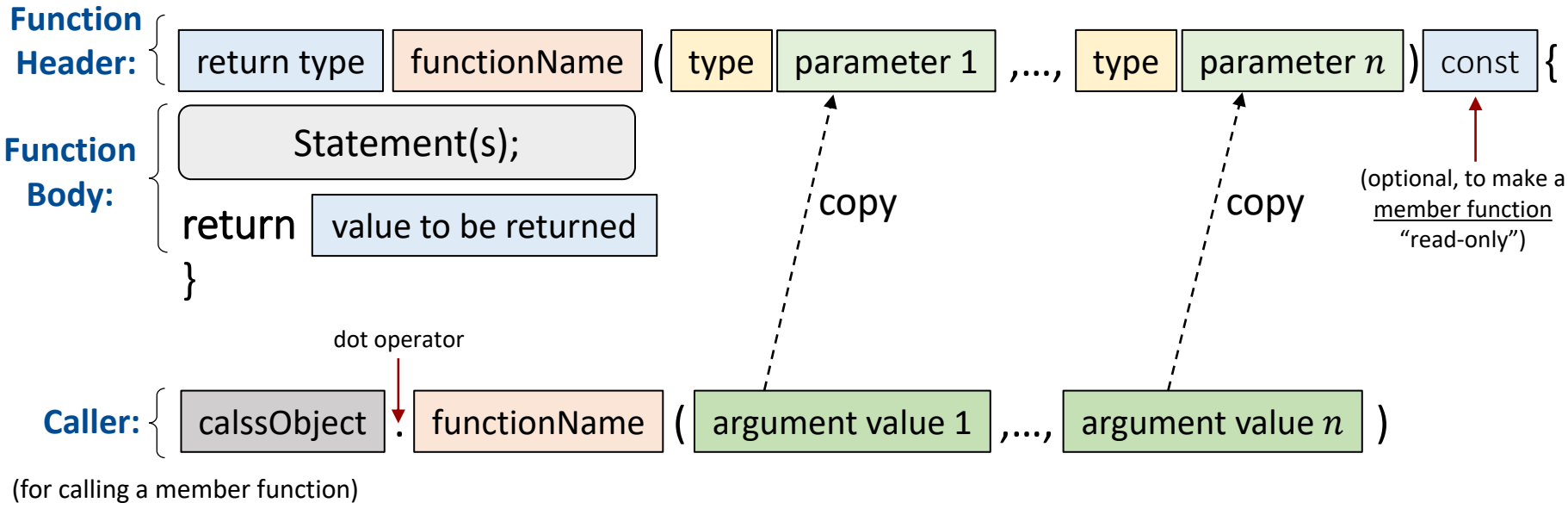
Member Functions and Their Callers

```
void setName(std::string accountName) {
  name = accountName;
}
```

```
std::string getName() const {
  return name;
}
```

```
myAccount.setName(theName);
```

```
myAccount.getName()
```



(for calling a member function)

Member Functions and Their Callers (cont.)

- If a function does not require any parameter to perform a task, its parameter list must be empty as ().
- The **return type** specifies the type of data the member function returns to its caller after performing its task. If a function does not return any information to its caller, its type must be void.
- Declaring a member function with const to the right of the parameter list make the member function "read-only" and force the compiler to issue a compilation error if that function modify the data members. This prevent accidental modification of the data members in some member functions like get member functions.
- The **argument types** in the member function call must be consistent (not necessarily identical) with the types of the corresponding parameters in the member function's definition.
- Typically, you cannot call a member function of a class until you create an object of that class (**static member functions** are an exception, that will be covered later).
- Variables declared in a particular function's body are **Local Variables** which can be used only in that function. When a function terminates, the values of its local variables are lost. Parameters of a function also are local variables of that function.

Access Specifiers: private and public

- There are 3 types of access specifiers: private, public, protected, followed by a colon (:).
- private: Data members or member functions listed after private (and before the next access specifier if there is one) are accessible only to the member functions of that class (or its "friends") and they are encapsulated (hidden) from other functions in the program (such as main()) and member functions of other classes (if there are any).
- public: Data members or member functions listed after public (and before the next access specifier if there is one) are accessible to other functions in the program (such as main()) and member functions of other classes (if there are any).
- By default, everything in a class is private, unless you specify otherwise.
- Once you list an access specifier, everything from that point has that access until you list another access specifier.
- Generally, **data members** should be private and **member functions** public.

Sample Program: Room Class

```
#include <iostream>
class Room {
public:
    int length;
    int width;
    int height;
    int calculateArea() {
      return length * width;
    }
    int calculateVolume() {
      return length * width * height;
    }
};

int main() {
  Room room1;
  room1.length = 42;
  room1.width = 30;
  room1.height = 19;
  std::cout << "Area of Room = "
    << room1.calculateArea() << std::endl;
  std::cout << "Volume of Room =  "
    << room1.calculateVolume() << std::endl;
}
```

(Bad Practice)

```
#include <iostream>
class Room {
public:
  void setData(int len, int wdth, int hgt) {
      length = len;
      width = wdth;
      height = hgt;
    }
 int calculateArea() {
      return length * width;
    }
 int calculateVolume() {
      return length * width * height;
    }
private:
    int length;
    int width;
    int height;
};

int main() {
  Room room1;
  room1.setData(42, 30, 19);
  std::cout << "Area of Room =  " << room1.calculateArea() << std::endl;
  std::cout << "Volume of Room =  " << room1.calculateVolume() << std::endl;
}
```

(Best Practice)

Initializing Objects with Constructors

Sample Program: A Simple Bank-Account Class with a Constructor that Initializes the Account Name

.cpp source-code file

```
// Using the Account constructor to initialize the name data
// member at the time each Account object is created.
#include <iostream>
#include "Account.h"

int main() {
   // create two Account objects
   Account account1{"Jane Green"};
   Account account2{"John Blue"};

   // display initial value of name for each Account
 std::cout << "account1 name is: " << account1.getName() << std::endl;
 std::cout << "account2 name is: " << account2.getName() << std::endl;
}
```

Class **Account** defined in the header **Account.h**

```
// Account class with a constructor that initializes the account name.
#include <string>

class Account {

public:
   // constructor initializes data member name with parameter accountName
   explicit Account(std::string accountName):name{accountName} {
      // empty body
   }

   // function to set the account name
   void setName(std::string accountName) {
      name = accountName;
   }

   // function to retrieve the account name
   std::string getName() const {
      return name;
   }

private:
   std::string name; // account name data member
}; // end class Account
```

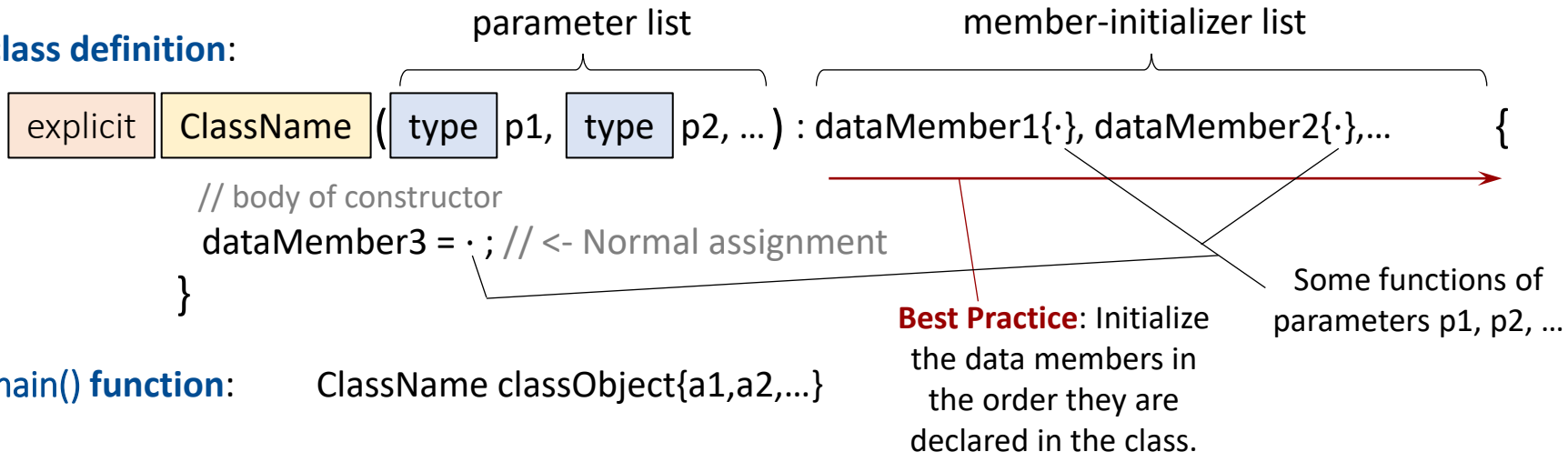
Constructors

```
explicit Account(std::string accountName):name{accountName} {
    // empty body
 }
```

```
Account account1{"Jane Green"};
Account account2{"John Blue"};
```

A special member function called **Constructor** can be defined in a class for initialization of each object of the class once it is created. This is an ideal way to initialize objects' data members.

In class definition:



In main() function: ClassName classObject{a1,a2,…}

Constructors

- Normally, constructors are public.
- The constructor must have the same name as the class.
- Constructors returns nothing; thus, we do not specify a return type (not even void).
- It is a good practice to initialize all data members in constructor, although it is not necessary.
- Constructors cannot be declared const (because initializing an object modifies it).
- Use explicit only for single-parameter constructors and multi-parameter constructors with default values.

- The member-initializer list executes before the constructor's body executes.
- Data members in member-initializer list are initialized in the order they are declared in the class definition. Thus, the **best practice** is to place the arguments in member-initializer list in the order they are declared in the class.

```
class Foo{
public:
  explicit Foo(int x):m1{x * x}, m2{m1 + x} { }
  ...
private:
   int m1;
   int m2;
};
```

```
int main() {
   Foo foo{3};
   ...
}
```

Constructors

- You can perform all initializations in the constructor's body. However, it's more efficient to do it with member initializers as much as possible.

explicit ClassName (type p1, type p2, …) {

```
  // body of constructor
  dataMember1{p1};
  dataMember2{p2};
  dataMember3{p3};
}
```

```
int main() {
   Foo foo{3};
   ...
}
```

```
class Foo{
public:
   explicit Foo(int x){
      m1 = x * x;
      m2 = m1 + x;
   }
private:
   int m1;
   int m2;
};
```

Data Validation

Sample Program: A Simple Bank-Account Class with Data Validation

Class **Account** defined in the header **Account.h**

```
// Account class with name and balance data members, and a
// constructor and deposit function that each perform validation.
#include <string>

class Account {
public:
  // Account constructor with two parameters
  Account(std::string accountName, int initialBalance)
    : name{accountName} { // assign accountName to data member name

    // validate that the initialBalance is greater than 0; if not,
    // data member balance keeps its default initial value of 0
    if (initialBalance > 0) { // if the initialBalance is valid
      balance = initialBalance; // assign it to data member balance
    }
  }

  // function that deposits (adds) only a valid amount to the balance
  void deposit(int depositAmount) {
    if (depositAmount > 0) { // if the depositAmount is valid
      balance = balance + depositAmount; // add it to the balance
    }
  }

  // function returns the account balance
  int getBalance() const {
    return balance;
  }

  // function that sets the name
  void setName(std::string accountName) {
    name = accountName;
  }

  // function that returns the name
  std::string getName() const {
    return name;
  }
private:
  std::string name; // account name data member
  int balance{0}; // data member with default initial value
}; // end class Account
```

Sample Program: A Simple Bank-Account Class with Data Validation (cont.)

.cpp source-code file

```
// Displaying and updating Account balances.
#include <iostream>
#include "Account.h"

int main()
{
  Account account1{"Jane Green", 50};
  Account account2{"John Blue", -7};

  // display initial balance of each object
  std::cout << "account1: " << account1.getName() << " balance is $"
    << account1.getBalance();
  std::cout << "\naccount2: " << account2.getName() << " balance is $"
    << account2.getBalance();

  std::cout << "\n\nEnter deposit amount for account1: "; // prompt
  int depositAmount;
  std::cin >> depositAmount; // obtain user input
  std::cout << "adding " << depositAmount << " to account1 balance";
  account1.deposit(depositAmount); // add to account1's balance

  // display balances
  std::cout << "\n\naccount1: " << account1.getName()
  << " balance is $" << account1.getBalance();
  std::cout << "\naccount2: " << account2.getName()
  << " balance is $" << account2.getBalance();

  std::cout << "\n\nEnter deposit amount for account2: "; // prompt
  std::cin >> depositAmount; // obtain user input
  std::cout << "adding " << depositAmount << " to account2 balance";
  account2.deposit(depositAmount); // add to account2 balance

  // display balances
  std::cout << "\n\naccount1: " << account1.getName()
  << " balance is $" << account1.getBalance();
  std::cout << "\naccount2: " << account2.getName()
  << " balance is $" << account2.getBalance() << std::endl;
}
```

Validation and Presentation Control of private data Using *Set* and *Get* Member Functions

To reduce errors, while increasing the robustness, security and usability of the programs:

- **Set functions** can be programmed to validate their arguments and reject any attempts to modify private data members to invalid values (e.g., a negative body temperature).
- **Get functions** can be programmed to present the private data in a different form, while the actual data representation remains hidden from the user (e.g., presenting a pass/fail instead of raw numeric data).

```
Account(std::string accountName, int initialBalance):name{accountName}{
  if (initialBalance > 0) {
    balance = initialBalance;
  }
}
```

Validation in constructor

```
void deposit(int depositAmount) {
  if (depositAmount > 0) {
    balance = balance + depositAmount;
  }
}
```

Validation in member function

```
int balance{0};
```

This is called **in-class initializer**.

Constructor vs Set Member Functions

- Constructors are called automatically once an object is created whereas calling Set member functions is always optional.
- Initialization by constructors are done only once whereas by using Set member functions you can change a few or all the attributes of the objects anytime.

```
#include <iostream>

int main() {
  MyClass myNumbers{1,2,3};
  std::cout << "Sum is: " << myNumbers.getSum() << std::endl;

  myNumbers.setY(10);
  std::cout << "Sum is: " << myNumbers.getSum() << std::endl;

  myNumbers.setNumbers(5,6,7);
  std::cout << "Sum is: " << myNumbers.getSum() << std::endl;
}
```

```
class MyClass {
public:
  MyClass (int X, int Y, int Z):x{X},y{Y},z{Z}{
  }
  void setY(int Y){
    y = Y;
  }
  void setNumbers(int X, int Y, int Z){
     x = X;
     y = Y;
     z = Z;
  }
  int getSum() {
    return x + y + z;
  }
private:
  int x, y, z;
};
```

UML Class Diagram

UML Class Diagram

UML (**U**nified **M**odeling **L**anguage) class diagrams are used to summarize a class's attributes and operations in concise, graphical, programming-language-independent manner, before implementing in specific programming languages.

In the UML, each class is modeled in a class diagram as a rectangle with three compartments: