C++ Classes and Member Functions

CS3021 Introduction to Data Structures and Intermediate Programming

C struct Revisited

- Recall that C structs allow us to group heterogeneous collections of elements.

```
struct employee {
  char name[25];
  int age;
  float salary;
};
struct employee John, Chester, Bill;
```

- But what if we wanted to provide the ability, from *within* **employee**, to modify John’s **salary** in a way that protects the data value from outside caller functions?

C++ Object-Orientation

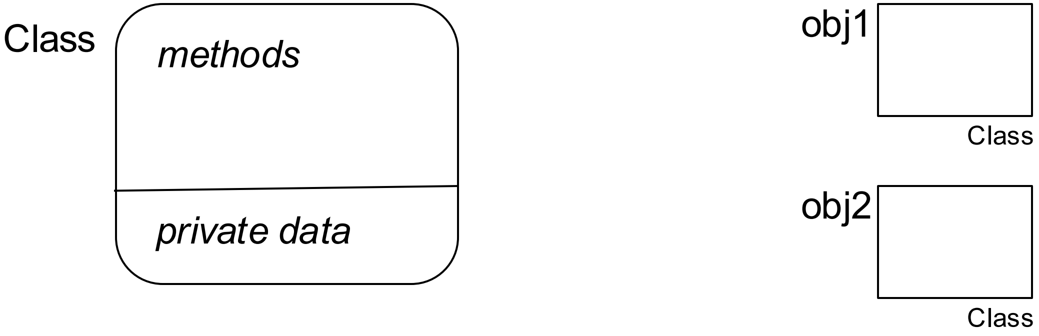
- Object-oriented programs use objects.
- An object is a *thing*, both tangible and intangible, such as an employee, a bank account, a vehicle, etc.
- To create and use an object inside a program, we must first provide a definition for the object, such as *what* kinds of information it holds, and *how* it behaves – this is called a class.

C++ Classes

- A class is an expanded concept of a structure.
- Instead of holding only data (like a struct), it can hold both data and functionality.
  - Data *members* in a class are collections of identically-typed items.
  - Member functions are called *methods.*

C++ Objects

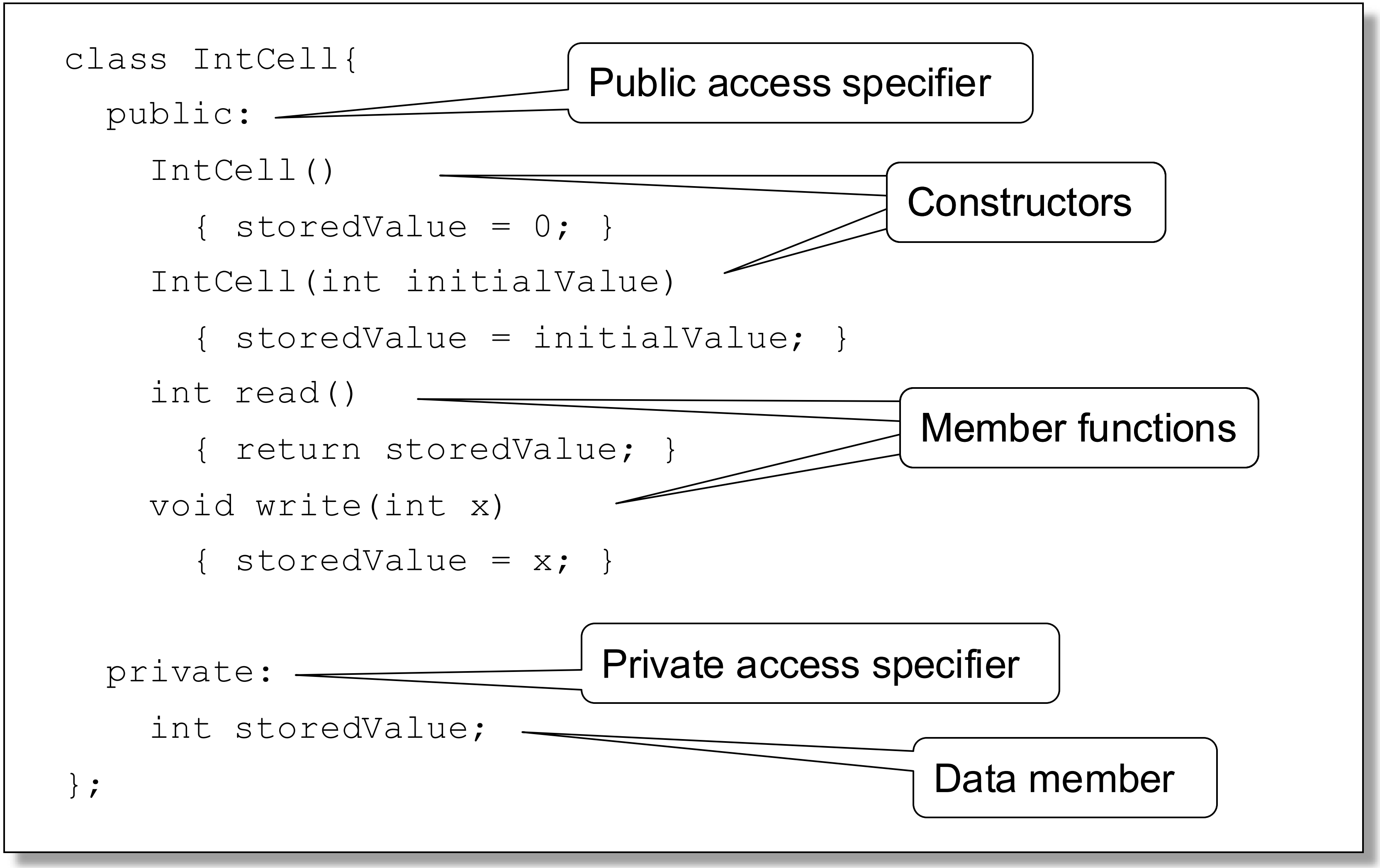
- An object is an *instantiation* of a class.
  - Roughly, a class is the definition, while an object is the declaration of an *instance* of that definition.
- Each object contains all the data components and member functions specified in the class.
- In terms of variables, a class would be a *type*, and an object would be a *variable.*



C++ Class Example

- Suppose we want to create a C++ class that stores a data member (in this case an **`int`**) in a data cell, and provides read/write access to that member.
- Why not just use an **`int`** in the first place?
  - Our example shows how we could build the concept of a class that holds *any* type of data member (another primitive type, a multi-dimensional array, another class, etc.)

C++ Class Definition



```
class IntCell{
 public:
  IntCell()
   { storedValue = 0; }
  IntCell(int initialValue)
   { storedValue = initialValue; }
  int read()
   { return storedValue; }
  void write(int x)
   { storedValue = x; }

 private:
  int storedValue;
};
```

IntCell1.cpp

Access Specifiers (1)

- The **`public`** keyword is an access specifier; **`public`** members are visible to any other object or function.

```
public:
  IntCell()
    { storedValue = 0; }
  IntCell(int initialValue)
    { storedValue = initialValue; }
  int read()
    { return storedValue; }
  void write(int x)
    { storedValue = x; }
```

- By default, all class variables and methods are **`private`**, meaning they are only visible to other member functions (methods) of the same object.

Access Specifiers (2)

- `public`: accessible within the body of the base class, and anywhere the program has a reference (e.g., a pointer) to an object of that base class, or any derived class of the base.
- `protected`: accessible within the body of the base class by friends and members of the base, and by any derived class of the base.
- `private`: accessible only within the body of the base class, and the friends of the base class.

Access Specifiers (simplified…)

- `public`: accessible to all classes, including *derived* classes.
- `protected`: accessible only to the class they belong to (and its *friends*), and any *derived* classes.
- `private`: accessible only to the class they belong to (and its *friends*).

Class Constructors (1)

- Objects generally need to initialize variables or allocate dynamic memory when they are created.
  - For example, what would happen if we called **`read()`** without first setting **`storedValue`**?
- To avoid problems, a class includes a special function (*method*) called a constructor.
  - Automatically called whenever a new object of this class is *instantiated*.

Class Constructors (2)

- A constructor must have the same name as the class, and ***cannot have any return type***, not even void.

- If no constructor is defined in a class, then the C++ compiler will include a *default constructor*.
  - If any constructor is defined, the compiler will not define a default constructor.

Constructor Example

- A class may have multiple constructors for different situations, for example:

```
IntCell()
  { storedValue = 0; }

IntCell(int initialValue)
  { storedValue = initialValue; }
```

Zero-parameter constructor

Defined-parameter constructor

- Use of **`explicit`** in the constructor prevents implicit type conversions we may not want:

```
explicit IntCell(int initialValue)
  { storedValue = initialValue; }
```

Constructor Usage Examples

```
int main () {
   IntCell obj1;
   IntCell obj2(12);
   IntCell obj3 = 37;
   IntCell obj4();
   cout << "obj1 value: " << obj1.read() << endl;
   cout << "obj2 value: " << obj2.read() << endl;
   cout << "obj3 value: " << obj3.read() << endl;
   cout << "obj4 value: " << obj4.read() << endl;
   obj1 = 10;
   cout << "obj1 value: " << obj1.read() << endl;
   return 0;  }
```

Invokes zero-parameter constructor

Invokes defined-parameter constructor

Illegal only if constructor is **explicit**

Error: function declaration!

Illegal only if constructor is **explicit**

IntCell1.cpp

Constructor Initializer Lists

- We can use initializer lists to initialize data members directly:

```
IntCell(int initialValue = 0)
   : storedValue (initialValue){}
```

Default value

Defined value

- The initializer list appears before the constructor body, which may not be needed, e.g., if only members are initialized.

IntCell2.cpp

Class Destructors

- Counterpart to class constructor.
- Called whenever an object goes out of scope (or if `delete` is explicitly called).
- Frees up resources allocated by object instantiation.
- Must have the same name as the class, but preceded with a tilde (~).
- Must return no value (as with a constructor).

Destructor Example

- IntCell destructor:

```
IntCell::~IntCell() { delete storedValue; }
```

- A destructor is not actually needed here, since **`IntCell`** only contains an **`int`** data member, which need not be deallocated.

Separate Interface/Implementation

- In C++, it is common to separate a class interface from its implementation.
- The interface lists class member variable declarations and public method prototypes.
- Methods are defined outside the class interface, in a separate class implementation.
  - The *scoping operator* (::) is used to indicate that a function belongs to a particular class.
  - Shorter functions may be defined inside the class interface.

IntCell Class Interface

```
#ifndef IntCell_h
#define IntCell_h

class IntCell {
  public:
    explicit IntCell(int initialValue);
    int read() const;
    void write(int x);

  private:
    int storedValue;
};

#endif
```

Header guard to prevent header file from being defined more than once.

IntCell.h

IntCell Class Implementation

```
#include “IntCell.h”

IntCell::IntCell(int initialValue = 0)
  : storedValue (initialValue){}

int IntCell::read() const {
  return storedValue;
}

void IntCell::write(int x) {
  storedValue = x;
}
```

scope resolution operator

Constructor w/ Initializer List

Signatures must match interface exactly, e.g., “const” must appear here

IntCell3.cpp

Questions?