0.1 Dynamic Memory Management with Operators new and delete

- Dynamic memory management
  - Control allocation and deallocation of memory
  - Operators **new** and **delete**
    - Include standard header <**new**>; Access to standard version of **new**
- **new**
  - Consider
    - **Time *timePtr;**
    - **timePtr = new Time;**
  - **new** operator
    - Creates object of proper size for type **Time**; Error if no space in memory for object
    - Calls default constructor for object
    - Returns pointer of specified type
  - Providing initializers
    - **double *ptr = new double(3.14159);**
    - **Time *timePtr = new Time(12, 0, 0);**
  - Allocating arrays; **int *gradesArray = new int[10];**
- **delete**
  - Destroy dynamically allocated object and free space
  - Consider; **delete timePtr;**
  - Operator **delete**
    - Calls destructor for object
    - Deallocates memory associated with object; Memory can be reused to allocate other objects
  - Deallocating arrays
    - **delete [] gradesArray;** ; Deallocates array to which **gradesArray** points
    - If pointer to array of objects
      - First calls destructor for each object in array
      - Then deallocates memory

0.2 static Class Members

Each object of a class has its own copy of all the **data members** of the class. in certain cases, only one copy of a variable should be shared by all objects of a class.

- **static** class variable
  - "Class-wide" data; Property of class, not specific object of class
  - Efficient when single copy of data is enough; Only the **static** variable has to be updated
  - May seem like global variables, but have class scope; Only accessible to objects of same class
  - Initialized exactly once at file scope
  - Exist even if no objects of class exist
  - Can be **public**, **private** or **protected**
- Accessing static class variables
  - Accessible through any object of class
  - **public static** variables
    - Can also be accessed using binary scope resolution operator(**::**)
    - **Employee::count**
- **private static** variables
  - When no class member objects exist
    - Can only be accessed via **public static** member function
    - To call **public static** member function combine class name, binary scope resolution operator (**::**) and function name; **Employee::getCount()**
- static member functions
  - Cannot access non-**static** data or functions
  - No **this** pointer for **static** functions; **static** data members and **static** member functions exist independent of objects

The programs of Figs. 1-4 demonstrates a **private static** data member called **count** and a **public static** member function called **getCount**. Figure 4 uses function **getCount** to determine the number of **Employee** objects currently instantiated.

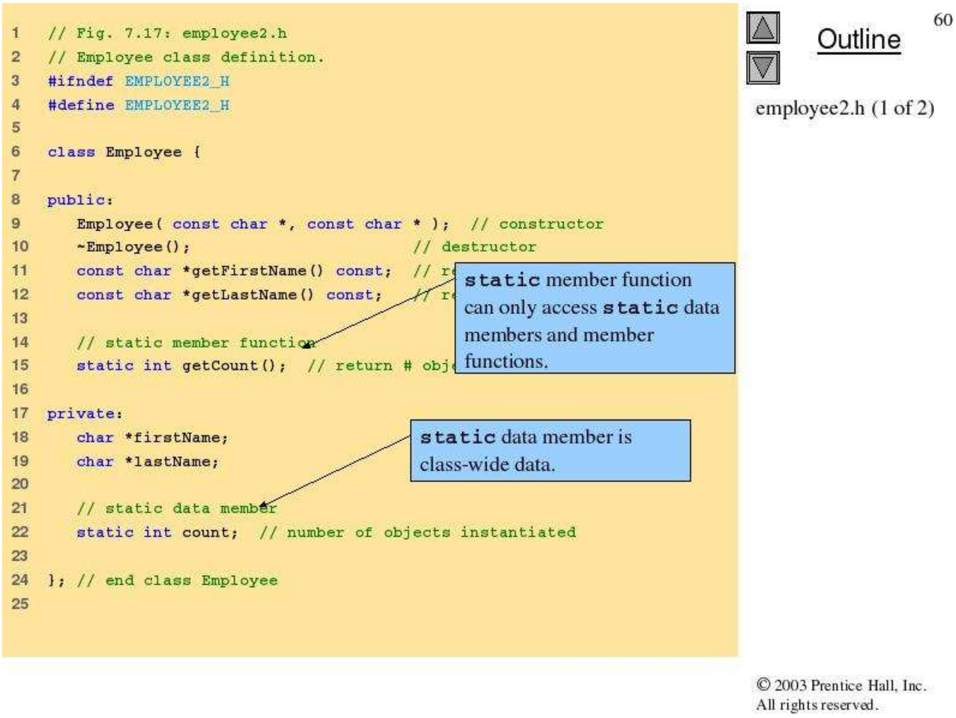


Figure 1: **Employee** class definition with a **static** data member to track the number **Employee** objects in memory.

61

```
#endif
```

Outline

employee2.h (2 of 2)

employee2.cpp
(1 of 3)

```
// Fig. 7.18: employee2.cpp
// Member-function definitions for class Employee.
#include <iostream>

using std::cout;
using std::endl;

#include <new>           // C++ standard new operator
#include <cstring>       // strcpy and strlen prototypes

#include "employee2.h"   // Employee class definition

// define and initialize static data member
int Employee::count = 0;

// define static member function that returns number of
// Employee objects instantiated
int Employee::getCount()
{
   return count;

} // end static function getCount
```

Initialize **static** data member exactly once at file scope.

static member function accesses **static** data member **count**.

62

Outline

employee2.cpp

```

// constructor dynamically allocates space for
// first and last name and uses strcpy to copy
// first and last names into the object
Employee::Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   strcpy( lastName, last );

   ++count;  // increment static count of employees

   cout << "Employee constructor for " << firstName
        << ' ' << lastName << " called." << endl;

} // end Employee constructor

// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
        << ' ' << lastName << endl;

```

new operator dynamically allocates space.

Use **static** data member to store total **count** of employees.

Figure 2: **Employee** class member-function definitions. (part 1 of 2)

63

Outline

employee2.cpp
(3 of 3)

```cpp
   delete [] firstName;  // recapture memory
   delete [] lastName;   // recapture memory

   --count;  // decrement static count of employees

} // end destructor ~Emp

// return first name of 
const char *Employee::getFirstName() const
{
   // const before return type prevents client from modifying
   // private data; client should copy returned string before
   // destructor deletes storage to prevent undefined pointer
   return firstName;

} // end function getFirstName

// return last name of employee
const char *Employee::getLastName() const
{
   // const before return type prevents client from modifying
   // private data; client should copy returned string before
   // destructor deletes storage to prevent undefined pointer
   return lastName;

} // end function getLastName
```

llocates

Use **static** data member to store total **count** of employees.

64

Outline

fig07_19.cpp
(1 of 2)

```cpp
// Fig. 7.19: fig07_19.cpp
// Driver to test class Employee.
#include <iostream>

using std::cout;
using std::endl;

#include <new>          // C++ standard new operator

#include "employee2.h"  // Employee class definition

int main()
{
   cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;   // use class name

   Employee *e1Ptr = new Employee(
   Employee *e2Ptr = new Employee(

   cout << "Number of employees after instantiation is "
        << e1Ptr->getCount();

```

new operator dynamically allocates space.

static member function can be invoked on any object of class.

Figure 3: **Employee** class member-function definitions. (part 2 of 2) and **static** data member tracking the number of objects of a class. (part 1 of 2)

```
    cout << "\n\nEmployee 1: "
         << e1Ptr->getFirstName()
         << " " << e1Ptr->getLastName()
         << "\nEmployee 2: "
         << e2Ptr->getFirstName()
         << " " << e2Ptr->getLastName() << "\n\n";

    delete e1Ptr;  // recapture memory
    e1Ptr = 0;     // disconnect pointer from free-store space
    delete e2Ptr;  // recapture memory
    e2Ptr = 0;     // disconnect pointer f

    cout << "Number of employees a
         << Employee::getCount() << endl;

    return 0;

} // end main
```

Operato
memory

static member function invoked using binary scope resolution operator (no existing class objects).

Outline

fig07_19.cpp
(2 of 2)


```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

Outline

fig07_19.cpp
output (1 of 1)

Figure 4: **static** data member tracking the number of objects of a class. (part 2 of 2)

0.3 Data Abstraction and Information Hiding

- **Information hiding**
  - Classes hide implementation details from clients
  - Example: stack data structure
    * Data elements added (pushed) onto top
    * Data elements removed (popped) from top
    * Last-in, first-out (LIFO) data structure
    * Client only wants LIFO data structure; Does not care how stack implemented
- **Data abstraction**; Describe functionality of class independent of implementation
- Abstract data types (ADTs)
  - Approximations/models of real-world concepts and behaviors; **int**, **float** are models for a numbers
  - Data representation
  - Operations allowed on those data
  - ADTs receive as much as attention today as structured programming did over the last two decades. (ADTs do not replace structured programming. rather, they provide an additional formalization that can further improve the program-development process.)
- C++ extensible; Standard data types cannot be changed, but new data types can be created

The job of high-level languages is to create a view convenient for programmers to use. There is no single accepted standard view-that is one reason why there are so many programming languages. Object-oriented programming in C++ presents yet another view.

The primary activity in C++ is creating new types (i.e., classes) and expressing the interactions among objects of those types.

0.3.1 Example: Array Abstract Data Type

An array is not much more than a pointer and some space in memory. Primitive capabilities! There are many operations that would be nice to perform with arrays, but there are not **built-in** C++. With C++ classes, the programmer can develop an array ADT is preferable to 'raw' arrays. Although the language is easy to extend with these new types, the base language itself is not changeable.

- ADT array
  - Subscript range checking
  - Arbitrary range of subscripts; Instead of having to start with 0
  - Array assignment
  - Array comparison
  - Array input/output
  - Arrays that know their sizes
  - Arrays that expand dynamically to accommodate more elements

0.3.2 Example: String Abstract Data Type

- Strings in C++
  - C++ does not provide built-in string data type; Maximizes performance
  - Provides mechanisms for creating and implementing string abstract data type; String ADT (Chapter 8)
  - ANSI/ISO standard **string** class (Chapter 19)

0.3.3 Example: Queue Abstract Data Type

A waiting line is also called a *queue*.

- Queue
  - FIFO; First in, first out
  - Enqueue; Put items in queue one at a time
  - Dequeue; Remove items from queue one at a time
- Queue ADT
  - Implementation hidden from clients; Clients may not manipulate data structure directly
  - Only queue member functions can access internal data

- Queue ADT (Chapter 15)
- Standard library **queue** class (Chapter 20)

The queue ADT guarantees the integrity of its internal data structure. Clients may not manipulate this data structure directly. Only the queue member functions have access to its internal data.

0.4 Container Classes and Iterators

- Container classes (collection classes)
  - Designed to hold collections of objects
  - Common services; Insertion, deletion, searching, sorting, or testing an item
  - Examples; Arrays, stacks, queues, trees and linked lists
- Iterator objects (iterators)
  - Returns next item of collection; Or performs some action on next item
  - Can have several iterators per container; Book with multiple bookmarks
  - Each iterator maintains own "position"
  - Discussed further in Chapter 20

0.5 Proxy Classes

Sometimes, it is desirable to hide the implementation details of a class to prevent access to proprietary information (including private data) and proprietary program login in a class. Providing clients of your class with a **proxy class** that knows only the public interface to your class enables the clients to use your class's services without giving the client access to your class's implementation details.

- Proxy class
  - Hide implementation details of another class
  - Knows only **public** interface of class being hidden
  - Enables clients to use class's services without giving access to class's implementation

- Forward class declaration
  - Used when class definition only uses pointer to another class
  - Prevents need for including header file
  - Declares class before referencing
  - Format: **class ClassToLoad;**

Implementation of a proxy class is demonstrated in Figs. 5-7.

74

```
// Fig. 7.20: implementation.h
// Header file for class Implementation

class Implementation {

public:

   // constructor
   Implementation( int v )
      : value( v )  // initialize value with v
   {
      // empty body

   } // end Implementation constructor

   // set value to v
   void setValue( int v )
   {
      value = v;  // should validate v

   } // end function setValue

```

public member function.

Outline

implementation.h
(1 of 2)

75

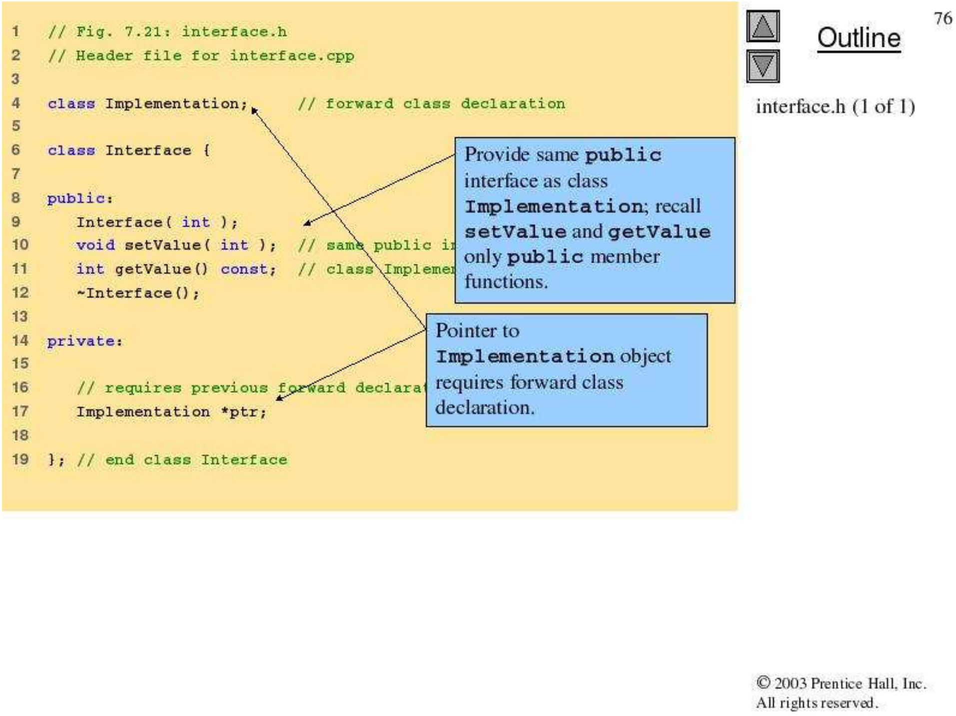
```
   // return value
   int getValue() const
   {
      return value;

   } // end function getValue

private:
   int value;

}; // end class Implementation
```

public member function.

Outline

implementation.h
(2 of 2)

Figure 5: **Implementation** class definition.



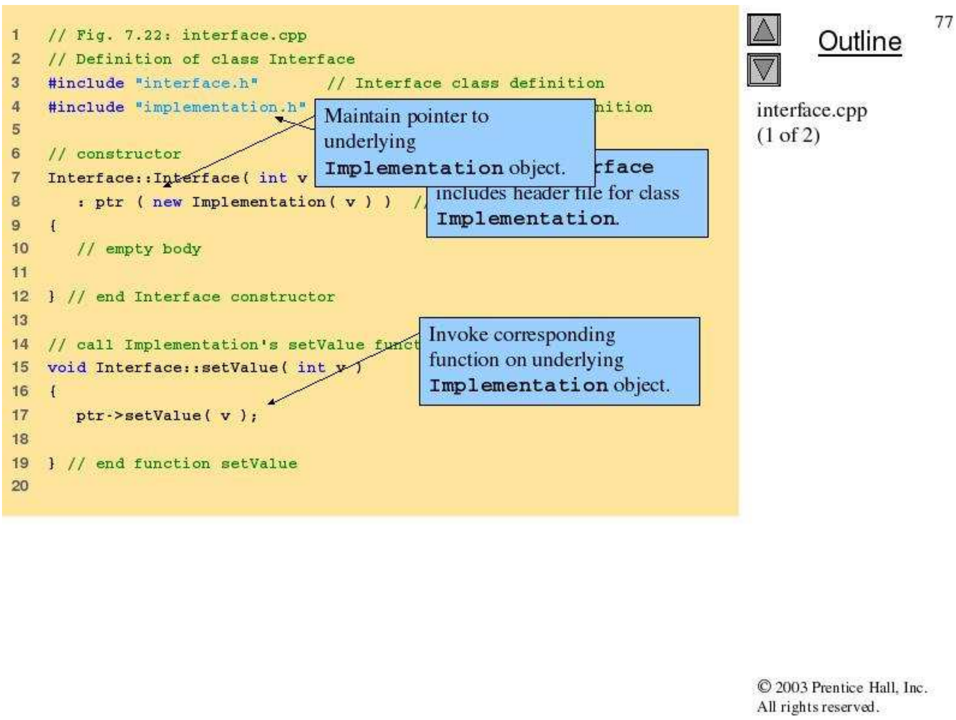


Figure 6: **Interface** class definition.

```
// call Implementation's getValue function
int Interface::getValue() const
{
   return ptr->getValue();

} // end function getValue

// destructor
Interface::~Interface()
{
   delete ptr;

} // end destructor ~Interface
```

Invoke corresponding function on underlying **`Implementation`** object.

Deallocate underlying **`Implementation`** object.

78

Outline

interface.cpp
(2 of 2)


```
// Fig. 7.23: fig07_23.cpp
// Hiding a class's private data with a proxy class.
#include <iostream>

using std::cout;
using std::endl;

#include "interface.h"  // Interface c

int main()
{
   Interface i( 5 );

   cout << "Interface contains: " << i.getValue()
        << " before setValue" << endl;

   i.setValue( 10 );

   cout << "Interface contains: " << i.getValue()
        << " after setValue" << endl;

   return 0;

```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```

Only include proxy class header file.

Create object of proxy class **`Interface`**; note no mention of **`Implementation`** class.

Invoke member functions via proxy class object.

79

Outline

fig07_23.cpp
(1 of 1)

fig07_23.cpp
output (1 of 1)

Figure 7: **Interface** class member-function definitions and Implementing a proxy class.