W6.1

- Destructors
- Data Members and Member Functions
- Returning a Reference to a Private Data Member
- Default Memberwise Copy
- Software Reusability

6.12 Using Destructors

- Destructors
  - Are member function of class
  - Perform termination housekeeping before the system reclaims the object's memory
  - Complement of the constructor
  - Name is tilde (**~**) followed by the class name (i.e., **`~Time`**)
    - Recall that the constructor's name is the class name
  - Receives no parameters, returns no value
  - One destructor per class
    - No overloading allowed

6.13 When Constructors and Destructors Are Called

- Constructors and destructors called automatically
  - Order depends on scope of objects
- Global scope objects
  - Constructors called before any other function (including **`main`**)
  - Destructors called when **`main`** terminates (or **`exit`** function called)
  - Destructors not called if program terminates with **`abort`**
- Automatic local objects
  - Constructors called when objects are defined
  - Destructors called when objects leave scope
    - i.e., when the block in which they are defined is exited
  - Destructors not called if the program ends with **`exit`** or **`abort`**

6.13 When Constructors and Destructors Are Called

- Static local objects
  - Constructors called when execution reaches the point where the objects are defined
  - Destructors called when **`main`** terminates or the **`exit`** function is called
  - Destructors not called if the program ends with **`abort`**

```
1  // Fig. 6.9: create.h
2  // Definition of class CreateAndDestroy.
3  // Member functions defined in create.cpp.
4  #ifndef CREATE_H
5  #define CREATE_H
6
7  class CreateAndDestroy {
8  public:
9     CreateAndDestroy( int );  // constructor
10    ~CreateAndDestroy();      // destructor
11 private:
12    int data;
13 };
14
15 #endif
```

Outline

5

1. Create a header file

1.1 Include function prototypes for the destructor and constructor


```
16 // Fig. 6.9: create.cpp
17 // Member function definitions for class CreateAndDestroy
18 #include <iostream>
19
20 using std::cout;
21 using std::endl;
22
23 #include "create.h"
24
25 CreateAndDestroy::CreateAndDestroy( int value )
26 {
27    data = value;
28    cout << "Object " << data << "   constructor";
29 }
30
31 CreateAndDestroy::~CreateAndDestroy()
32    { cout << "Object " << data << "   destructor " << endl; }
```

Outline

6

2. Load the header file

2.1 Modify the constructor and destructor

Constructor and Destructor changed to print when they are called.

```
// Fig. 6.9: fig06_09.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>

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

#include "create.h"

void create( void );   // prototype

CreateAndDestroy first( 1 );  // global object

int main()
{
   cout << "   (global created before main)" << endl;

   CreateAndDestroy second( 2 );        // local object
   cout << "   (local automatic in main)" << endl;

   static CreateAndDestroy third( 3 );  // local object
   cout << "   (local static in main)" << endl;

   create();  // call function to create objects

   CreateAndDestroy fourth( 4 );        // local object
   cout << "   (local automatic in main)" << endl;
   return 0;
}
```


Outline

3. Create multiple objects of varying types

```

// Function to create objects
void create( void )
{
   CreateAndDestroy fifth( 5 );
   cout << "   (local automatic in create)" << endl;

   static CreateAndDestroy sixth( 6 );
   cout << "   (local static in create)" << endl;

   CreateAndDestroy seventh( 7 );
   cout << "   (local automatic in create)" << endl;
}
```


Outline

Program Output

```
OUTPUT
Object 1   constructor   (global created before main)
Object 2   constructor   (local automatic in main)
Object 3   constructor   (local static in main)
Object 5   constructor   (local automatic in create)
Object 6   constructor   (local static in create)
Object 7   constructor   (local automatic in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local automatic in main)
Object 4   destructor
Object 2   destructor
Object 6   destructor
Object 3   destructor
Object 1   destructor
```

Notice how the order of the constructor and destructor call depends on the types of variables (automatic, global and **`static`**) they are associated with.

6.14 Using Data Members and Member Functions

- Member functions
  - Allow clients of the class to *set* (i.e., write) or *get* (i.e., read) the values of private data members
  - Example:

    Adjusting a customer's bank balance
    - **`private`** data member **`balance`** of a class **`BankAccount`** could be modified through the use of member function **`computeInterest`**
    - A member function that sets data member **`interestRate`** could be called **`setInterestRate`**, and a member function that returns the **`interestRate`** could be called **`getInterestRate`**
  - Providing *set* and *get* functions does not make **`private`** variables **`public`**
  - A set function should ensure that the new value is valid

6.15 A Subtle Trap: Returning a Reference to a Private Data Member

- Reference to an object
  - Alias for the name of the object,
  - May be used on the left side of an assignment statement, makes perfectly acceptable *lvalue.*
  - Reference can receive a value, which changes the original object as well
- Returning references
  - **`public`** member functions can return non-**`const`** references to **`private`** data members
    - Should be avoided, breaks encapsulation
- Please avoid using references in this way, very, very bad!!!

```
1  // Fig. 6.11: time4.h
2  // Declaration of the Time class.
3  // Member functions defined in time4.cpp
4
5  // preprocessor directives that
6  // prevent multiple inclusions of header file
7  #ifndef TIME4_H
8  #define TIME4_H
9
10 class Time {
11 public:
12    Time( int = 0, int = 0, int = 0 );
13    void setTime( int, int, int );
14    int getHour();
15    int &badSetHour( int );  // DANGEROUS reference return
16 private:
17    int hour;
18    int minute;
19    int second;
20 };
21
22 #endif
```

Notice how member function **badSetHour** returns a reference (**int &** is the return type).

Outline

1. Define class

1.1 Function prototypes

1.2 Member variables


```
23 // Fig. 6.11: time4.cpp
24 // Member function definitions for Time class.
25 #include "time4.h"
26
27 // Constructor function to initialize private data.
28 // Calls member function setTime to set variables.
29 // Default values are 0 (see class definition).
30 Time::Time( int hr, int min, int sec )
31    { setTime( hr, min, sec ); }
32
33 // Set the values of hour, minute, and second.
34 void Time::setTime( int h, int m, int s )
35 {
36    hour   = ( h >= 0 && h < 24 ) ? h : 0;
37    minute = ( m >= 0 && m < 60 ) ? m : 0;
38    second = ( s >= 0 && s < 60 ) ? s : 0;
39 }
40
41 // Get the hour value
42 int Time::getHour() { return hour; }
43
44 // POOR PROGRAMMING PRACTICE:
45 // Returning a reference to a private data member.
46 int &Time::badSetHour( int hh )
47 {
48    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
49
50    return hour;  // DANGEROUS reference return
51 }
```

badSetHour returns a reference to the **private** member variable **hour.** Changing this reference will alter **hour** as well.

Outline

1. Load header

1.1 Function definitions

Outline

1.2 Declare reference

2. Change data using a reference

3. Output results

```
// Fig. 6.11: fig06_11.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
// Time class has been trimmed for this example.
#include <iostream>

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

#include "time4.h"

int main()
{
   Time t;
   int &hourRef = t.badSetHour( 20 );

   cout << "Hour before modification: " << hourRef;
   hourRef = 30;  // modification with invalid value
   cout << "\nHour after modification: " << t.getHour();

   // Dangerous: Function call that returns
   // a reference can be used as an lvalue!
   t.badSetHour(12) = 74;
   cout << "\n\n*********************************\n"
        << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
        << "badSetHour as an lvalue, Hour: "
        << t.getHour()
        << "\n*********************************" << endl;

   return 0;
}
```

Declare **Time** object **t** and reference **hourRef** that is assigned the reference returned by the call **t.badSetHour(20)**.

```
Hour before modification: 20
```

Alias used to set the value of **hour** to 30 (an invalid value).

```
Hour after modification: 30
```

Function call used as an *lvalue* and assigned the value **74** (another invalid value).

```
*********************************
POOR PROGRAMMING PRACTICE!!!!!!!!
badSetHour as an lvalue, Hour: 74
*********************************
```


Outline

Program Output

```
Hour before modification: 20
Hour after modification: 30

*********************************
POOR PROGRAMMING PRACTICE!!!!!!!!
badSetHour as an lvalue, Hour: 74
*********************************
```

HourRef used to change **hour** to an invalid value. Normally, the function **setbadSetHour** would not have allowed this. However, because it returned a reference, **hour** was changed directly.

6.16 Assignment by Default Memberwise Copy

- Assigning objects
  - An object can be assigned to another object of the same type using the assignment operator (=)
  - Member by member copy
- Objects may be
  - Passed as function arguments
  - Returned from functions (call-by-value default)

Outline

1. Define class

1.1 Define member functions

```
1  // Fig. 6.12: fig06_12.cpp
2  // Demonstrating that class objects can be assigned
3  // to each other using default memberwise copy
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  // Simple Date class
10 class Date {
11 public:
12    Date( int = 1, int = 1, int = 1990 ); // default constructor
13    void print();
14 private:
15    int month;
16    int day;
17    int year;
18 };
19
20 // Simple Date constructor with no range checking
21 Date::Date( int m, int d, int y )
22 {
23    month = m;
24    day = d;
25    year = y;
26 }
27
28 // Print the Date in the form mm-dd-yyyy
29 void Date::print()
30    { cout << month << '-' << day << '-' << year; }
```

Outline

2. Create `Date` objects

2.1 Memberwise copy

3. Print values

```
31
32 int main()
33 {
34    Date date1( 7, 4, 1993 ), date2;  // d2 defaults to 1/1/90
35
36    cout << "date1 = ";
37    date1.print();
38    cout << "\ndate2 = ";
39    date2.print();
40
41    date2 = date1;    // assignment by default memberwise copy
42    cout << "\n\nAfter default memberwise copy, date2 = ";
43    date2.print();
44    cout << endl;
45
46    return 0;
47 }
```

`date2` set equal to **`date1`**, and all member variables are copied.

Program Output

```
date1 = 7-4-1993
date2 = 1-1-1990

After default memberwise copy, date2 = 7-4-1993
```


6.17 Software Reusability

- Software resusability
  - Implementation of useful classes
  - Class libraries exist to promote reusability
    - Allows for construction of programs from existing, well-defined, carefully tested, well-documented, portable, widely available components
  - Speeds development of powerful, high-quality software