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`**

```
// Fig. 6.9: create.h
// Definition of class CreateAndDestroy.
// Member functions defined in create.cpp.
#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {
public:
   CreateAndDestroy( int );  // constructor
   ~CreateAndDestroy();      // destructor
private:
   int data;
};

#endif
```

1. Create a header file

1.1 Include function prototypes for the destructor and constructor

```cpp
// Fig. 6.9: create.cpp
// Member function definitions for class CreateAndDestroy
#include <iostream>

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

#include "create.h"

CreateAndDestroy::CreateAndDestroy( int value )
{
   data = value;
   cout << "Object " << data << "   constructor";
}

CreateAndDestroy::~CreateAndDestroy()
   { cout << "Object " << data << "   destructor " << endl; }
```

Constructor and Destructor changed to print when they are called.

Outline

2. Load the header file

2.1 Modify the constructor and destructor

```
// 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;
}
```

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;
}
```

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!!!

```
// Fig. 6.11: time4.h
// Declaration of the Time class.
// Member functions defined in time4.cpp

// preprocessor directives that
// prevent multiple inclusions of header file
#ifndef TIME4_H
#define TIME4_H

class Time {
public:
   Time( int = 0, int = 0, int = 0 );
   void setTime( int, int, int );
   int getHour();
   int &badSetHour( int );  // DANGEROUS reference return
private:
   int hour;
   int minute;
   int second;
};

#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

Outline

1. Load header

1.1 Function definitions

```cpp
// Fig. 6.11: time4.cpp
// Member function definitions for Time class.
#include "time4.h"

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }

// Set the values of hour, minute, and second.
void Time::setTime( int h, int m, int s )
{
   hour   = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Get the hour value
int Time::getHour() { return hour; }

// POOR PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
   hour = ( hh >= 0 && hh < 24 ) ? hh : 0;

   return hour;  // DANGEROUS reference return
}
```

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

```
// 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

*********************************
```

1.2 Declare reference

2. Change data using a reference

3. Output results

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)

```
// Fig. 6.12: fig06_12.cpp
// Demonstrating that class objects can be assigned
// to each other using default memberwise copy
#include <iostream>

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

// Simple Date class
class Date {
public:
   Date( int = 1, int = 1, int = 1990 ); // default constructor
   void print();
private:
   int month;
   int day;
   int year;
};

// Simple Date constructor with no range checking
Date::Date( int m, int d, int y )
{
   month = m;
   day = d;
   year = y;
}

// Print the Date in the form mm-dd-yyyy
void Date::print()
   { cout << month << '-' << day << '-' << year; }
```

1. Define class

1.1 Define member functions

```

int main()
{
   Date date1( 7, 4, 1993 ), date2;  // d2 defaults to 1/1/90

   cout << "date1 = ";
   date1.print();
   cout << "\ndate2 = ";
   date2.print();

   date2 = date1;    // assignment by default memberwise copy
   cout << "\n\nAfter default memberwise copy, date2 = ";
   date2.print();
   cout << endl;

   return 0;
}
```

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

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

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

2. Create Date objects

2.1 Memberwise copy

3. Print values

Program Output

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