Preview

- Constant Object and Constant Functions
- Composition: Object as Member of Classes
- Friend functions
- The Use of Friend Functions
- Friend Classes

Constant Objects and Member Functions

- Some object need to be modifiable and some do not.
- A programmer can define a constant object with keyword const.
- Once we define a constant object, it cannot be modifiable.

```
const Time noon( 12, 0, 0 );   // constant object
```


Constant Objects and Member Functions

- We can also define a constant member function with const keyword.
- A constant function is a "read-only" function that does not modify the object for which it is called.
- To declare a constant member function, place the **const** keyword after the closing parenthesis of the argument list.

Constant Objects and Member Functions

- C++ compiler does not allow any member function call for constant objects unless the member functions themselves are also declared constant
- A constant function can be called for non-constant object.
- Non-constant function cannot be called for a constant object


```
// Declaration of the class Time.
#ifndef TIME_H
#define TIME_H

class Time {
public:
   Time( int = 0, int = 0, int = 0 );  // default constructor

   // set functions
   void setTime( int, int, int );  // set time
   void setHour( int );     // set hour
   void setMinute( int );   // set minute
   void setSecond( int );   // set second

   // get functions (normally declared const)
   int getHour() const;     // a constant function return hour
   int getMinute() const;   // a constant function return minute
   int getSecond() const;   // a constant function rreturn second

   // print functions (normally declared const)
   void printMilitary() const; // print military time
   void printStandard();  // print standard time
private:
   int hour;              // 0 - 23
   int minute;            // 0 - 59
   int second;            // 0 - 59
};

#endif
```



```
// Member function definitions for Time class.
#include <iostream>

using namespace std;

#include "time.h"

// Constructor function to initialize private
data.
// 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 )
{
   setHour( h );
   setMinute( m );
   setSecond( s );
}

// Set the hour value
void Time::setHour( int h )
   { hour = ( h >= 0 && h < 24 ) ? h : 0; }

// Set the minute value
void Time::setMinute( int m )
   { minute = ( m >= 0 && m < 60 ) ? m : 0; }

// Set the second value
void Time::setSecond( int s )
   { second = ( s >= 0 && s < 60 ) ? s : 0; }
```

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

// Get the minute value
int Time::getMinute() const { return minute; }

// Get the second value
int Time::getSecond() const { return second; }

// Display military format time: HH:MM
void Time::printMilitary() const
{
   cout << ( hour < 10 ? "0" : "" ) << hour <<
":"
        << ( minute < 10 ? "0" : "" ) <<
minute;
}

// Display standard format time: HH:MM:SS AM
(or PM)
void Time::printStandard()  // should be const
{
   cout << ( ( hour == 12 ) ? 12 : hour % 12 )
<< ":"
        << ( minute < 10 ? "0" : "" ) << minute
<< ":"
        << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Constant Objects and Member Functions

```
// main function with time.h and time.cpp.
#include "time.h"

int main()
{
   Time wakeUp( 6, 45, 0 );       // non-constant object
   const Time noon( 12, 0, 0 );   // constant object

                          // MEMBER FUNCTION    OBJECT
   wakeUp.setHour( 18 );  // non-const          non-const

   noon.setHour( 12 );    // non-const          const -> it is not ok

   wakeUp.getHour();      // const              non-const->  it is ok

   noon.getMinute();      // const              const
   noon.printMilitary();  // const              const
   noon.printStandard();  // non-const          const -> it is not ok
   return 0;
}
```


Composition: Objects as Members of Classes

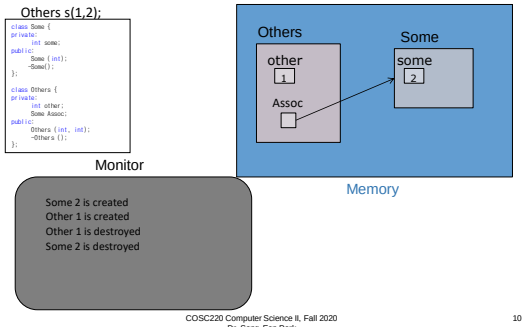
- An object can be a member of other class.
- Composition is simply defining a class as a member of another class.
- Since member objects are constructed in the other where they are declared, the object member constructor must be called first for the memory allocation for an object.

Composition: Objects as Members of Classes

```
// SomeOther.h header file
// Composition: Objects as Members of Classes

#ifndef SOMEOTHER_H
#define SOMEOTHER_H

class Some {
private:
        int some;
public:
        Some (int = 0);
        ~Some();
};

class Others {
private:
        int other;
        Some Assoc; // an object is a member of a class
public:
        Others (int, int);
        ~Others ();
};
#endif
```

```
#include <iostream>
#include "SomeOther.h"
using namespace std;

Some::Some(int x) // constructor for class Some
{
        some = x;
        cout << "Some "<< some <<" is created"<<endl;
}

Some::~Some() // Destructor for a class 'Some'
{
        cout << "Some "<<some <<" is destroyed"<<endl;
}

// Constructor for a class 'Others'
Others::Others(int x, int y):Assoc(y)
{
        other = x;
        cout <<"Other " <<other<< " is created"<<endl;
}

Others::~Others() // Destructor for a class 'Others
{
        cout <<"Other " <<other<< " is destroyed"<<endl;
}

int main()
{
        Others s(1,2);
        return 0;
}
```


Composition: Objects as Members of Classes



Composition: Objects as Members of Classes

```
// date1.h
// Declaration of the Date class.
// Member functions defined in date1.cpp
#ifndef DATE1_H
#define DATE1_H

class Date {
public:
   Date( int = 1, int = 1, int = 1900 ); // default arguments
   void print() const;  // print date in month/day/year format
   ~Date();  // provided to confirm destruction order
private:
   int month;  // 1-12
   int day;    // 1-31 based on month
   int year;   // any year
   // utility function to test proper day for month and year
   int checkDay( int );
};

#endif
```


Composition: Objects as Members of Classes

```
// emply1.h
// Declaration of the Employee class.
// Member functions defined in emply1.cpp
#ifndef EMPLY1_H
#define EMPLY1_H

#include "date1.h"

class Employee {
public:
   Employee( char *, char *, int, int, int, int, int, int );
   void print() const;
   ~Employee();  // provided to confirm destruction order
private:
   char firstName[ 25 ];
   char lastName[ 25 ];
   const Date birthDate;
   const Date hireDate;
};

#endif
```

Composition: Objects as Members of Classes

```cpp
// date1.cpp
// Member function definitions for Date class.
#include <iostream>
#include "date1.h"
using namespace std;

// Constructor: Confirm proper value for month and day
Date::Date( int mn, int dy, int yr )
{
   if ( mn > 0 && mn <= 12 )       // validate the month
      month = mn;
   else {
      month = 1;
      cout << "Month " << mn << " invalid. Set to month 1.\n";
   }

   year = yr;                      // should validate yr
   day = checkDay( dy );           // validate the day

   cout << "Date object constructor for date ";
   print();      // interesting: a print with no arguments
   cout << endl;
}

// Print Date object in form  month/day/year
void Date::print() const
   { cout << month << '/' << day << '/' << year; }
```

```cpp
// Destructor: provided to confirm destruction order
Date::~Date()
{
   cout << "Date object destructor for date ";
   print();
   cout << endl;
}

// Utility function to confirm proper day value
// based on month and year.
// Is the year 2000 a leap year?
int Date::checkDay( int testDay )
{
   static const int daysPerMonth[ 13 ] =
      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

   if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
      return testDay;

   if ( month == 2 &&      // February: Check for leap year
        testDay == 29 &&
        ( year % 400 == 0 ||
        ( year % 4 == 0 && year % 100 != 0 ) ) )
      return testDay;

   cout << "Day " << testDay << " invalid. Set to day 1.\n";

   return 1;  // leave object in consistent state if bad value
}
```


Composition: Objects as Members of Classes

```cpp
// emply1.cpp
// Member function definitions for Employee class.
#include <iostream>
#include <cstring>
#include "emply1.h"
#include "date1.h"
using namespace std;
Employee::Employee( char *fname, char *lname,
                    int bmonth, int bday, int byear,
                    int hmonth, int hday, int hyear )
   : birthDate( bmonth, bday, byear ),
     hireDate( hmonth, hday, hyear )
{
   // copy fname into firstName and be sure that it fits
   int length = strlen( fname );
   length = ( length < 25 ? length : 24 );
   strncpy( firstName, fname, length );
   firstName[ length ] = '\0';

   // copy lname into lastName and be sure that it fits
   length = strlen( lname );
   length = ( length < 25 ? length : 24 );
   strncpy( lastName, lname, length );
   lastName[ length ] = '\0';

   cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;
}
```

```cpp
// emply1.cpp
// Member function definitions for Employee class.
#include <iostream>
#include <cstring>
#include "emply1.h"
#include "date1.h"
using namespace std;
Employee::Employee( char *fname, char *lname,
                    int bmonth, int bday, int byear,
                    int hmonth, int hday, int hyear )
   : birthDate( bmonth, bday, byear ),
     hireDate( hmonth, hday, hyear )
{
   // copy fname into firstName and be sure that it fits
   int length = strlen( fname );
   length = ( length < 25 ? length : 24 );
   strncpy( firstName, fname, length );
   firstName[ length ] = '\0';

   // copy lname into lastName and be sure that it fits
   length = strlen( lname );
   length = ( length < 25 ? length : 24 );
   strncpy( lastName, lname, length );
   lastName[ length ] = '\0';

   cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;
}
```


Composition: Objects as Members of Classes

```cpp
// Demonstrating composition: an object with member objects.
#include <iostream>

using namespace std;

#include "emply1.h"

int main()
{
   Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );

   cout <<endl;
   e.print();
   cout <<endl;
   cout << "Test Date constructor with invalid values: "<<endl;
   Date d( 14, 35, 1994 );  // invalid Date values
   cout << endl;
   return 0;
}
```


Friend Functions

- The private member data of a class can be accessed only by the class' member functions. But, there is one exception.
- A friend function will be friendly with a class even though it is not a member of that class.
- By the term friendly we mean that the friend function can access the private members of the class.

Friend Functions

- **Remember:** Friend functions are not members of any class but they can access private data of the class to which they are a friend.
- **Beware:** Since they are not members of any class, you should not call them using the dot operator.


```cpp
#include <iostream>

using namespace std;
class car
{
private:
    int speed;
    char color[20];
public:
        car();
friend void display(car); //Friend of the class 'car'
};

car::car( )
{
        cout<<"Enter the color : ";
        cin>>color;
        cout<<"Enter the speed : ";
        cin>>speed;
}

void display(car x)
{
        cout<<"The color of the car is : "<<x.color <<endl;
        cout<<"The speed of the car is : "<<x.speed<<endl;
}
int main( )
{
        car mine;
        display(mine); //passing the object 'mine'to the friend function
        return 0;
}
```

The Use of Friend Function

- A friend function can be friendly to 2 or more classes.
- The friend function does not belong to any class, so it can be used to access private data of two or more classes.
- No other function can do that!
- you can write individual member functions in each class but the use of a friend function makes it simpler and easier to understand the logic.


```
// The usage of friend function
#include <iostream>
using namespace std;
class virus; // forward declaration of 'virus' class
class bacteria{
private:
      int life;
public:
            bacteria( ) {life =1;}
friend void check(bacteria, virus);
};

class virus{
private:
      int life;

public:
            virus() {life = 0;}
friend void check(bacteria, virus);
};
void check (bacteria b,virus v)
{
            if (b.life==1 || v.life==1)
                        cout<<"Something is alive"<<endl;
            if (b.life == 1)
                        cout<<"A bacteria is alive."<<endl;
            if (v.life == 1)
                        cout<<"A virus is alive."<<endl;
}
int main( )
{
            bacteria fever;
            virus cholera;
            check(fever, cholera);
return 0;
}
```


Friend Classes

- It is often useful for one class to see the private variables of another class, even though these variables should probably not be made part of the public interface that the class supports.
- Just like functions are made friends of classes, we can also make one class to be a friend of another class.

Friend Classes

- For instance, if you were writing a Linked List class, you might want to use a Node class that has private data, but it would still be convenient for the functions that actually combine nodes together to be able to access the data directly without having to work through the Node interface.

Friend Classes

```
// A header file node.h for definition of the class "Node"
#ifndef NODE_H
#define NODE_H

class Node {
private:
        char LastName[20]; // student last name
        char FirstName[20]; // student first name
        int IDNumber; // student ID number
        Node *Next; // point to next node in the linked list
public:
        Node(); // constructor
        ~Node(){;};// destructor
        void PrintNode();
        // LinkedList class is a friend of Node class
        friend class LinkedList;
};

#endif
```


Friend Classes

```
// A header file List.h for LinkedList class definition
#include "node.h"

#ifndef LIST_H
#define LIST_H

class LinkedList{
private:
        Node *List;
        Node *SearchLocation (int);
        Node *CreateNode ();
public:
        LinkedList(); //constructor
        void PrintList ();
        void InsertNode ();
        void DeleteNode ();
        void SearchNode();
};
#endif
```

```
//*************************************************
// Insert a new node into the sorted linked list
// Parameter: None
// Return Type: None
//*************************************************
void LinkedList::InsertNode ()
{
        Node *NewNode = CreateNode(); // create a new node
        Node *Tmp = List; // temperay pointer to traverse linked list
        // Case1: list is empty
        if (List ==NULL)
                List= NewNode;
        // Case 2: a new node as a first node
        else if (List->IDNumber > NewNode->IDNumber)
        {
                NewNode->Next = List;
                List = NewNode;
        }
        else
        {
                Tmp = SearchLocation(NewNode->IDNumber);
                //Case 4:Insert as a last element
                if (Tmp->Next == NULL)
                        Tmp->Next = NewNode;
                //Case 3: Insert between two nodes
                else if (Tmp->Next->IDNumber > NewNode->IDNumber)
                {
                        NewNode->Next= Tmp->Next;
                        Tmp->Next =NewNode;
                }
        }
}
```