The this Pointer

- By default, the C++ compiler provides each member function of a class with an **implicit parameter** that points to the object through which the member function was called. This impliocit parameter is called **"this"**.

- **Example:**

```
class Example
{
   int x;
 public:
   Example(int a)        //constructor
   {
      x = a;       //Note the implicit use of a pointer to the object
   }
   void setValue(int);
   int getValue();
   void printAddressAndValue();
};
```

- **Member Functions:**

```
void Example::setValue(int x)
{
  this->x = x;    //Note the explicit use of the this pointer
}

int Example::getValue()
{
  return x;       //Note the implicit use of a pointer to the object
                  // calling getValue().
}

void Example::printAddressAndValue()
{
  cout << "The object at address " << this << " has "
     << "value " << this->x << endl;
```

● Main() Program Illustrating the this Pointer:

```
// This program illustrates the this pointer.
#include <iostream>
#include "ThisExample.h"
using namespace std;

int main()
{
  Example ob1(10), ob2(20);

  // Print the addresses of the two objects.
  cout << "In main():" << endl;
  cout << "The object at address " << &ob1 << " has "
     << "value " << ob1.getValue() << endl;
  cout << "The object at address " << &ob2 << " has "
     << "value " << ob2.getValue() << endl;
  cout << endl;


  // Print the addresses and values from within
  // the member function.
  cout << "Calling printAddressAndValue():" << endl;
  ob1.printAddressAndValue();
  ob2.printAddressAndValue();

  return 0;
}
```

Constant Member Functions

- A constant member function is one that does not modify the object through which it was called.
- When placed right after the parameter list in the definition of a member function, the const keyword servers as an indication to the compiler that the member function should not be allowed to modify its object.
- **Example:**

```
class ConstExample
{
    int x;
  public:
    ConstExample(int a)        //constructor
    {
        x = a;
    }
    void setValue(int);
    int getValue() const;
};


int ConstExample::getValue() const
{
  return x;       //Note: getValue() will not be able to modify x
}
```

Static Members

- **Static Member Variables:**
  - a member variable that is shared by all objects of the class.
  - it must be declared inside the class with the keyword **static.**
  - it must be defined outside of the class definition.
  - it can be accessed and modified by all objects of the class.
  - modifications by one object of the class are visible by all objects of the class.

- **Example - Class Specification with Static Member Variables:**

```
#ifndef BUDGET_H
#define BUDGET_H

class Budget
{
  private:
    static double corpBudget;             //static member variable
    double divBudget;
  public:
    Budget()
    {
      divBudget = 0;
    }
    void addBudget(double b)
    {
      divBudget += b;
     corpBudget += divBudget;             //modifies the static variable
    }
    double getDivBudget()
    {
      return divBudget;
    }
    double getCorpBudget()
    {
      return corpBudget;
    }
};
#endif
```

```
// This program demonstrates a static class member variable.
#include <iostream>
#include <iomanip>
#include "budget.h"          // For Budget class declaration
using namespace std;

// Definition of the static member of the Budget class.
double Budget::corpBudget = 0;

int main()
{
   const int N_DIVISIONS = 4;

   // Create instances of the Budget class.
   Budget divisions[N_DIVISIONS];

   // Get the budget request for each division.
   for (int count = 0; count < N_DIVISIONS; count++)
   {
      double bud;

      cout << "Enter the budget request for division ";
      cout << (count + 1) << ": ";
      cin >> bud;
      divisions[count].addBudget(bud);  //modifies the static variable
   }

   // Display the budget request for each division.
   cout << setprecision(2);
   cout << showpoint << fixed;
   cout << "\nHere are the division budget requests:\n";
   for (int count = 0; count < N_DIVISIONS; count++)
   {
      cout << "Division " << (count + 1) << "\t$ ";
      cout << divisions[count].getDivBudget() << endl;
   }

   // Display the total budget request.
   cout << "Total Budget Requests:\t$ ";
   cout << divisions[0].getCorpBudget() << endl;

   return 0;
}
```

- **Static Member Functions:**
  - typically used to access static member variables.
  - can be called before any objects of the class are created.
  - declared by prefixing its declaration with the keyword **static**
  - can be called independently of class objects by using the class name (see example).

- **Example - Class Specification with Static Member Functions:**

```
#ifndef BUDGET_H
#define BUDGET_H

class Budget
{
private:
   static double corpBudget;          //static member variable
   double divBudget;
public:
   Budget()
   {
      divBudget = 0;
   }
   void addBudget(double b)
   {
      divBudget += b;
      corpBudget += divBudget;        //uses the static variable
   }
   double getDivBudget()
   {
      return divBudget;
   }
   static double getCorpBudget()      //static member function
   {
      return corpBudget;
   }
   static void mainOffice(double);    //static member function
};
#endif
```

● **Static Member Function Implementation:**

```
#include "budget2.h"

// Definition of the static member of Budget class.
double Budget::corpBudget = 0;

//*********************************************
// Definition of static member function mainOffice
// This function adds the main office's budget request to
// the corpBudget variable.
//*********************************************
void Budget::mainOffice(double budReq)
{
   corpBudget += budReq;    //uses the static variable
}
```

```
// This program demonstrates a static class member function.
#include <iostream>
#include <iomanip>
#include "budget2.h"       // For Budget class declaration
using namespace std;

int main()
{
   const int N_DIVISIONS = 4;

   // Get the budget requests for each division.
   cout << "Enter the main office's budget request: ";
   double amount;
   cin >> amount;

   // Call the static member function of the Budget class.
   Budget::mainOffice(amount);      //Note: no instance yet created

   // Create instances of the Budget class.
   Budget divisions[N_DIVISIONS];
   for (int count = 0; count < N_DIVISIONS; count++)
   {
      double bud;

      cout << "Enter the budget request for division ";
      cout << (count + 1) << ": ";
      cin >> bud;
      divisions[count].addBudget(bud);
   }
   // Display the budget for each division.
   cout << setprecision(2);
   cout<< showpoint << fixed;
   cout << "\nHere are the division budget requests:\n";
   for (int count = 0; count < N_DIVISIONS; count++)
   {
      cout << "\tDivision " << (count + 1) << "\t$ ";
      cout << divisions[count].getDivBudget() << endl;
   }
   // Print total budget requests.
   cout << "Total Requests (including main office): $ ";
   cout << Budget::getCorpBudget() << endl;    //Note call
   return 0;
}
```

Friends of Classes

- A **friend function** is a function that is not a member of a class, but **has access to the private members of the class**.
- A friend function can be a stand-alone function or a member function of another class.
- A friend function is declared a friend with the **friend** keyword in the function prototype.
- An entire class can be declared a friend of a class. In this case all member functions of the friend class have unrestricted access to the class.

● **Example - Class Specification - Including Friend Function:**

```
#ifndef BUDGET3_H
#define BUDGET3_H
#include "auxil.h"  // For Aux class declaration

// Budget class declaration.
class Budget
{
  private:
    static double corpBudget;
    double divBudget;
  public:
    Budget() { divBudget = 0; }
    void addBudget(double b)
       { divBudget += b; corpBudget += divBudget; }
    double getDivBudget() { return divBudget; }
    static double getCorpBudget() { return corpBudget; }
    static void mainOffice(double);
    friend void Aux::addBudget(double); //friend function
                                        // prototype
};
#endif
```

● **Class Implementation File:**

```
#include "budget3.h"

// Definition of static member.
double Budget::corpBudget = 0;

//*****************************************
// Definition of static member function mainOffice
// This function adds the main office's budget request to
// the corpBudget variable.
//*****************************************
void Budget::mainOffice(double budReq)
{
   corpBudget += budReq;
}
```

● **Aux Class Specification:**

```
#ifndef AUXIL_H
#define AUXIL_H

// Aux class declaration.
class Aux
{
  private:
    double auxBudget;
  public:
    Aux()
    {
       auxBudget = 0;
    }

    void addBudget(double);      //this is the friend function

    double getDivBudget()
    {
       return auxBudget;
    }
};
#endif
```

● **Aux Implementation File (Contains the Friend Function):**

```
#include "auxil.h"
#include "budget3.h"

//*********************************************
// Definition of member function addBudget
// This function is declared a friend by the Budget class
// It adds the value of argument b to the static corpBudget
// member variable of the Budget class.
//*********************************************
void Aux::addBudget(double b)
{
   auxBudget += b;
   Budget::corpBudget += auxBudget;
}
```

```
// This program demonstrates a static class member variable.
#include <iostream>
#include <iomanip>
#include "budget3.h"
using namespace std;

int main()
{
  const int N_DIVISIONS = 4;

   // Get the budget requests for the divisions and offices.
   cout << "Enter the main office's budget request: ";
   double amount;
   cin >> amount;
   Budget::mainOffice(amount);

   // Create the division and auxilliary offices.
   Budget divisions[N_DIVISIONS];
   Aux auxOffices[N_DIVISIONS];

   for (int count = 0; count < N_DIVISIONS; count++)
   {
      double bud;

      cout << "Enter the budget request for Division ";
      cout << (count + 1) << ": ";
      cin >> bud;
      divisions[count].addBudget(bud);
      cout << "Enter the budget request for Division ";
      cout << (count + 1) << "'s\nauxiliary office: ";
      cin >> bud;
      auxOffices[count].addBudget(bud);   //uses friend function
   }
   // Print the budgets.
   cout << setprecision(2);
   cout << showpoint << fixed;
   cout << "Here are the division budget requests:\n";
   for (int count = 0; count < N_DIVISIONS; count++)
   {
     cout << "\tDivision " << (count + 1) << "\t\t\t$ ";
     cout << setw(7);
     cout << divisions[count].getDivBudget() << endl;
     cout << "\tAuxiliary Office of Division " << (count+1);
     cout << "\t$  ";
     cout << auxOffices[count].getDivBudget() << endl;
   }

   // Print total requests.
   cout << "\tTotal Requests (including main office): $ ";
   cout << Budget::getCorpBudget() << endl;
   return 0;
}
```

Memberwise Assignment

- The assignment operator, = , can be used to assign one object to another, or to initialize one object with another object's data.

- By default, each member of one object is copied to its counterpart in the other object.

- Example:

```
DemoClass object1, object2;

...

object2 = object1;                  //assignment

DemoClass object3 = object1;     //declare and initialize
```

Copy Constructors

- A **copy constructor** is a special constructor that is called whenever a new object is created and initialized with the data of another object of the same class.

- If the programmer does not specify a copy constructor for the class, the compiler calls a **default copy constructor** which simply copies all the data of the existing object to the new object using memberwise assignment.

```
// This program demonstrates the operation of the
// default copy constructor.
#include <iostream>
using namespace std;

class Address
{
  private:
    string street;
  public:
    Address() { street = ""; }
    Address(string st) { setStreet(st); }
    void setStreet(string st) { street = st; }
    string getStreet() { return street; }
};

int main()
{
    // Mary and Joan live at same address.
    Address mary("123 Main St");
    Address joan = mary;        //invokes the default copy constructor
    cout << "Mary lives at " << mary.getStreet() << endl;
    cout << "Joan lives at " << joan.getStreet() << endl;

    // Now Joan moves out.
    joan.setStreet("1600 Pennsylvania Ave");
    cout << "Now Mary lives at " << mary.getStreet() << endl;
    cout << "Now Joan lives at " << joan.getStreet() << endl;

    return 0;
}
```

- **Default copy constructors can cause difficulties when objects contain pointers to dynamic memory.**

- Default copy constructors can cause "sharing of storage"; that is, pointers of multiple objects pointing to the same storage.

- Consequently, modification of memory by one object affects other objects sharing that memory.

- **Destructors of one object can delete memory still in use by another object.**

- **Example: Class Specification with Pointer to Dynamic Storage**

```
#include <iostream>
using namespace std;

class NumberArray
{
  private:
    double *aPtr;      //pointer to dynamic storage
    int arraySize;
  public:
    NumberArray(int size, double value);
    // The destructor is commented out to avoid problems with
    // the default copy constructor.
    /********
    ~NumberArray()
    {
       if (arraySize > 0)
       delete [ ] aPtr;
    }
    *********/
    void print();
    void setValue(double value);
};
```

● **Class Implementation File:**

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

//*******************************************
//Constructor allocates an array of the given size and sets all its
// entries to the given value.
//*******************************************
NumberArray::NumberArray(int size, double value)
{
   arraySize = size;
   aPtr = new double[arraySize];     //uses dynamic allocation
   setValue(value);
}

//*******************************************
//Sets all the entries of the array to the same value.
//*******************************************
void NumberArray::setValue(double value)
{
   for(int index = 0; index < arraySize; index++)
      aPtr[index] = value;
}

//*************************************
//Prints all the entries of the array.
//*************************************
void NumberArray::print()
{
   for(int index = 0; index < arraySize; index++)
      cout << aPtr[index] << "  ";
}
```

```
// This program demonstrates the deficiencies of
// the default copy constructor.
#include <iostream>
#include <iomanip>
#include "NumberArray.h"
using namespace std;

int main()
{
   // Create an object.
   NumberArray first(3, 10.5);

   // Make a copy of the object.
   NumberArray second = first;        //invokes copy constructor

   // Display the values of the two objects.
   cout << setprecision(2) << fixed << showpoint;
   cout << "Value stored in first object is ";
   first.print();
   cout << endl << "Value stored in second object is ";
   second.print();
   cout << endl << "Only the value in second object "
       << "will be changed." << endl;

   // Now change the  value stored in the second object.
   second.setValue(20.5);

   // Display the values stored in the two objects.
   cout << "Value stored in first object is ";
   first.print();
   cout << endl << "Value stored in second object is ";
   second.print();

   return 0;
}
```

- Changing the data in one object changes the data in the other object because the default copy constructor copies the value of the pointer in the first object to the pointer in the second object. Hence, **both pointers point to the same object!**

Programmer-Defined Copy Constructors

- A programmer defined copy constructor must have **a sinlge parameter that is a reference to the same class.**
- The copy constructor uses the data in the object passed to initialize the object being created.
- The copy constructor can **allocate separate memory to hold the new objects dynamic data.** The new object's pointer will point to this memory.
- The old object's data will be copied - not its pointer.
- It is a good idea to make a copy constructor's parameter constant by specifying the **const** keyword:
- **Example - Class with Programmer-Defined Copy Constructor**:

```
#include <iostream>
using namespace std;

class NumberArray
{
  private:
    double *aPtr;
    int arraySize;
  public:
    NumberArray(const NumberArray &);  //programmer-defined
                                       //copy constructor
    NumberArray(int size, double value);  //constructor

    ~NumberArray()                     //destructor
    {
       if (arraySize > 0)
       delete [] aPtr;
    }

    void print();
    void setValue(double value);
};
```

● **Class Implementation File:**

```
#include <iostream>
#include "NumberArray2.h"
using namespace std;

//*****************************************
//Copy constructor allocates a new array and copies into it the
//entries of the array in the other object.
//*****************************************
NumberArray::NumberArray(const NumberArray &obj)
{
   arraySize = obj.arraySize;
   aPtr = new double[arraySize];
   for(int index = 0; index < arraySize; index++)
      aPtr[index] = obj.aPtr[index];
}

//*********************************************
//Constructor allocates an array of the given size and sets all its
// entries to the  given value.
//*********************************************
NumberArray::NumberArray(int size, double value)
{
   arraySize = size;
   aPtr = new double[arraySize];
   setValue(value);
}
//************************************************
//Sets all the entries of the array to the same value.
//************************************************
void NumberArray::setValue(double value)
{
   for(int index = 0; index < arraySize; index++)
      aPtr[index] = value;
}
//**************************************
//Prints all the entries of the array.
//**************************************
void NumberArray::print()
{
   for(int index = 0; index < arraySize; index++)
      cout << aPtr[index] << "  ";
}
```

```
// This program demonstrates the use of copy constructors.
#include <iostream>
#include <iomanip>
#include "NumberArray2.h"

using namespace std;

int main()
{
   NumberArray first(3, 10.5);

   //Make second a copy of first object.
   NumberArray second = first;      //invokes copy constructor

   // Display the values of the two objects.
   cout << setprecision(2) << fixed << showpoint;
   cout << "Value stored in first object is ";
   first.print();
   cout << "\nValue stored in second object is ";
   second.print();

   cout <<  "\nOnly the value in second object will "
      <<  "be changed.\n";

   //Now change value stored in second object.
   second.setValue(20.5);

   // Display the values stored in the two objects.
   cout << "Value stored in first object is ";
   first.print();
   cout << endl << "Value stored in second object is ";
   second.print();

   return 0;
}
```

Operator Overloading

- C++ allows you to redefine how standard operators (e.g., =, + , ...) work when used with class objects.
- The name of the function for the overloaded operator is **operator followed by the operator symbol**; e.g.,

```
operator=()      //for overloading the = operator
operaor+()       //for overloading the + operator
```

- Operators can be overloaded as either:
  - instance member functions
  - friend functions
- The overloaded operator must have the same number of parameters as the standard version.
- A binary operator overloaded as an instance member function needs only one parameter - the right operand.

```
class OpClass {
    int x;
  public:
    OpClass operator+(OpClass right);
};
```

- The left operand of the overloaded binary operator is the calling object and is accessed through the **this** pointer.

```
OpClass OpClass::operator+(OpClass right)
{
    OpClass sum;
    sum.x = this->x + right.x;
    return sum;
}
```

- Sample Usage:

```
OpClass a,b,s;
s = a.operator+(b)       //invoke as member function
s = a + b;               //more conventional use
```

Overloading the Assignment = Operator

- Overloading the assignment operator solves problems with object assignment when the object contains a pointer to dynamic memory.

- The assignment operator is most naturally overloaded as an instance of a member function.

- It needs to return the value of the assigned object to allow for cascaded assignments; such as, a = b = c;

- In general, the assignment operator should be overloaded whenever a non-default copy constructor is used.

- In particular, classes allocating dynamic memory to a pointer member in any constructor should define both a copy constructor and an overloaded assignment operator.

- **Example - Class with Overloaded Assignment Operator**:

```
#include <iostream>
using namespace std;

class NumberArray
{
  private:
    double *aPtr;
    int arraySize;
  public:
    // Overloaded operator function.
    void operator=(const NumberArray &right);

    // Constructors and other member functions.
    NumberArray(const NumberArray &);       //copy constructor
    NumberArray(int size, double value);    //constructor
    ~NumberArray()                          //destructor
    {
       if (arraySize > 0) delete [ ] aPtr;
    }
    void print();
    void setValue(double value);
};
```

● **Class Implementation File**:

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

//*******************************************
//The overloaded operator function for assignment.
//*******************************************
void NumberArray::operator=(const NumberArray &right)
{
   if (arraySize > 0) delete [] aPtr;
   arraySize = right.arraySize;
   aPtr = new double[arraySize];
   for (int index = 0; index < arraySize; index++)
      aPtr[index] = right.aPtr[index];
}

//*******************************************
//Copy constructor.
//*******************************************
NumberArray::NumberArray(const NumberArray &obj)
{
   arraySize = obj.arraySize;
   aPtr = new double[arraySize];
   for(int index = 0; index < arraySize; index++)
      aPtr[index] = obj.aPtr[index];
}

//*******************************************
//Constructor.
//*******************************************
NumberArray::NumberArray(int size1, double value)
{
   arraySize = size1;
   aPtr = new double[arraySize];
   setValue(value);
}
```

```
//*******************************************/
/Sets the value stored in all entries of the array.
//******************************************v
oid NumberArray::setValue(double value)
{
   for(int index = 0; index < arraySize; index++)
      aPtr[index] = value;
}

//*************************************
//Print out all entries in the array.   *
//*************************************
void NumberArray::print()
{
   for(int index = 0; index < arraySize; index++)
      cout << aPtr[index] << "  ";
}
```

● **Sample Program:**

```
// This program demonstrates overloading of
// the assignment operator.
#include <iostream>
#include <iomanip>
#include "overload.h"
using namespace std;

int main()
{
   NumberArray first(3, 10.5);
   NumberArray second(5, 20.5);

   // Display the values of the two objects.
   cout << setprecision(2) << fixed << showpoint;
   cout << "First object's data is ";
   first.print();
   cout << endl << "Second object's data is ";
   second.print();

   // Call the overloaded operator.
   cout << "\nNow we will assign the second object "
         << "to the first." << endl;
   first = second;          //invokes overloaded = operator

   // Display the new values of the two objects.
   cout << "First object's data is ";
   first.print();
   cout << endl << "The second object's data is ";
   second.print();

   return 0;
}
```

General Issues with Operator Overloading

- If desired, you can change the entire meaning of an operator:

```
class Weird {
    int value;
  public:
    Weird(int v)
    {
        value = v;
    }
    void operator=(const Weird &right)
    {
        cout << right.value << endl;
    }
};
```

- Now, consider the following code:

```
Weird a(5), b(5);     //uses the constructor
a = b;                //uses overloaded assignment operator
```

  Although the statement a = b looks like an assignment statement, it actually causes the contents of b's "value" to be displayed on the screen.

- When overloading an operator, you can not change the number of operands.

- Most operators can be overloaded.

- You can not overload the following operators:

```
?:    .    .*    ::    sizeof
```

Overloading Math and Relational Operators

- Many classes would benefit from overloaded math operators.

- Overloading the prefix ++operator:
  - Since unary operators only affect the object making the operator function call, there is no need for a parameter (i.e., the member function will have a void parameter).

- Overloading the postfix ++operator:
  - The member function will have a **dummy parameter**. When the C++ compiler sees the dummy parameter in an operator function, it knows that the function is designed to be used in postfix mode.
  - The member function must use a temporary local object which is initialized to the value of the object making the function call.

- Overloading Relational Operators:
  - Relational operators can be overloaded so that objects of classes can be compared using regular relational expressions.
  - Overloaded relational operators should return a **bool** value.

- **Example: Class Specification File**

```
#ifndef FEETINCHES_H
#define FEETINCHES_H

// A class to hold distances or measurements expressed
// in feet and inches.
class FeetInches {
  private:
    int feet;
    int inches;
    void simplify();
  public:
    FeetInches(int f = 0, int i = 0)
    {
        feet = f;
        inches = i;
        simplify();
    }
    void setData(int f, int i)
    {
        feet = f;
        inches = i;
        simplify();
    }
    int getFeet()
    {
        return feet;
    }
    int getInches()
    {
        return inches;
    }

    // overloaded arithmetic and boolean operators.
    FeetInches operator + (const FeetInches &) const;
    FeetInches operator - (const FeetInches &) const;
    FeetInches operator++();
    FeetInches operator++(int);
    bool operator>(const FeetInches &) const;
    bool operator<(const FeetInches &) const;
    bool operator==(const FeetInches &) const;
};
#endif
```

● **Example: Class Implementation File**

```
#include "feetinch4.h"

//*******************************************
// Definition of member function simplify. This function checks for
// values in the inches member greater than 12  or less than 0.
// If such a value is found, the numbers in feet and inches are
// adjusted to conform to a standard feet and inches expression.
// Thus, 3 feet 14 inches would be adjusted to 4 feet 2 inches and
// 5 feet -2 inches would be adjusted to 4 feet 10 inches.
//*******************************************void
FeetInches::simplify()
{
   inches = 12*feet + inches;
   feet = inches / 12;
   inches = inches % 12;
}

//*******************************************
// Overloaded binary + operator.
//*******************************************
FeetInches FeetInches::operator+(const FeetInches &right) const
{
   FeetInches temp;

   temp.inches = inches + right.inches;
   temp.feet = feet + right.feet;
   temp.simplify();
   return temp;
}

//*******************************************
// Overloaded binary - operator.
//*******************************************
FeetInches FeetInches::operator-(const FeetInches &right) const
{
   FeetInches temp;

   temp.inches = inches - right.inches;
   temp.feet = feet - right.feet;
   temp.simplify();
   return temp;
}
```

```
//*******************************************
// Overloaded prefix ++ operator. Causes the
// inched member to be incremented. Returns the
// incremented  object.
//*******************************************
FeetInches FeetInches::operator++()
{
     ++inches;
    simplify();
    return *this;
}

//*******************************************
// Overloaded postfix ++ operator. Causes the
// inches member to be incremented. Returns the
// value of the object before the increment.
//*******************************************
FeetInches FeetInches::operator++(int)   //int is a dummy parameter
{
    FeetInches temp(feet, inches);

    inches++;
    simplify();
    return temp;
}
```

```
//*******************************************************
// Overloaded > operator. Returns true if the
// current object is set to a value greater than
// that of right.
//*******************************************************
bool FeetInches::operator>(const FeetInches &right) const
{
   if (feet > right.feet)
      return true;
   if (feet == right.feet && inches > right.inches)
      return true;
   return false;
}

//*******************************************************
// Overloaded < operator. Returns true if the
// current object is set to a value less than
// that of right.
//*******************************************************
bool FeetInches::operator<(const FeetInches &right) const
{
  return right > *this;
}

//*******************************************************
// Overloaded == operator. Returns true if the
// current object is set to a value equal to that
// of right.
//*******************************************************
bool FeetInches::operator==(const FeetInches &right) const
{
   if (feet == right.feet && inches == right.inches)
      return true;
   else
      return false;
}
```

● **Sample Program:**

```
// This program demonstrates the FeetInches class's
// overloaded math and relational operators.
#include <iostream>
#include "feetinch4.h"
using namespace std;

int main()
{
   FeetInches first, second, third;
   int f, i;

   //Demonstrating overload + and - operators
   cout << "Demonstrating overlaoded + and - operators.\n";
   cout << "Enter a distance in feet and inches: ";
   cin  >> f >> i;
   first.setData(f, i);
   cout << "Enter another distance in feet and inches: ";
   cin  >> f >> i;
   second.setData(f, i);
   third = first + second;
   cout << "first + second = ";
   cout << third.getFeet() << " feet, ";
   cout << third.getInches() << " inches.\n";
   third = first - second;
   cout << "first - second = ";
   cout << third.getFeet() << " feet, ";
   cout << third.getInches() << " inches.\n";
   cout << endl;

   //Demonstrating overload prefix ++ operator
   cout << "Demonstrating overlaoded prefix ++ operator.\n";
   for (int count = 0; count < 12; count++)
   {
      first = ++second;
      cout << "first: " << first.getFeet() << " feet, ";
      cout << first.getInches() << " inches. ";
      cout << "second: " << second.getFeet() << " feet, ";
      cout << second.getInches() << " inches.\n";
   }
```

```
    //Demonstrating overload postfix ++ operator
    cout << "\nDemonstrating overloaded postfix ++ operator.\n";
    for (int count = 0; count < 12; count++)
    {
        first = second++;
        cout << "first: " << first.getFeet() << " feet, ";
        cout << first.getInches() << " inches. ";
        cout << "second: " << second.getFeet() << " feet, ";
        cout << second.getInches() << " inches.\n";
    }
    cout << endl;

    //Demonstrating overload relational operators
    cout << "Demonstrating overlaoded relational operators.\n";
    cout << "Enter a distance in feet and inches: ";
    cin >> f >> i;
    first.setData(f, i);
    cout << "Enter another distance in feet and inches: ";
    cin >> f >> i;
    second.setData(f, i);
    if (first == second)
        cout << "First is equal to second.\n";
    if (first > second)
        cout << "First is greater than second.\n";
    if (first < second)
        cout << "First is less than second.\n";

    return 0;
}
```

Overloading The < < and > > Operators

- **cout** and **cin** are respective members of the **ostream** and **istream** classes defined in the C + + runtime library. (The cout and cin objects are instances of ostream and istream.

- The overloaded < < (> >) function, will require two input parameters - **a reference to the actual ostream (istream) object on the left side of the < < (> >) operator and a reference to the object on the right side of the < < (> >) operator**.

- The overloaded < < (> >) function, returns a reference to the actual ostream (istream) object so as to allow for a “chained” operation; such as,

```
cout << object1 << " " << object2 << endl;

cin >> object1 >> object2 >> object3;
```

- **Example: Class Specification File**

```
#ifndef FEETINCHES_H
#define FEETINCHES_H

#include <iostream>
using namespace std;

class FeetInches
{
 private:
   int feet;
   int inches;
   void simplify();
 public:
   FeetInches(int f = 0, int i = 0)
      { feet = f; inches = i; simplify(); }
   void setData(int f, int i)
      { feet = f; inches = i; simplify(); }
   int getFeet()
      { return feet; }
   int getInches()
      { return inches; }

   // Overloaded arithmetic, boolean, and io operators.
   FeetInches operator + (const FeetInches &) const;
   FeetInches operator - (const FeetInches &) const;
   FeetInches operator++();
   FeetInches operator++(int);
   bool operator>(const FeetInches &) const;
   bool operator<(const FeetInches &) const;
   bool operator==(const FeetInches &) const;
   friend ostream &operator<<(ostream &, FeetInches &);
   friend istream &operator>>(istream &, FeetInches &);
};
#endif
```

- Note: The **friend** designation gives the operator functions direct access to the class's private members.

● **Example: Class Implementation File for << and >>**

```
//*********************************************
// Overloaded << operator. Gives cout the ability to
// directly display FeetInches objects.
//*********************************************o
stream &operator<<(ostream &strm, FeetInches &obj)
{
   strm << obj.feet << " feet, " << obj.inches << " inches";
   return strm;
}

//*********************************************
// Overloaded >> operator. Gives cin the ability to
// store user input directly into FeetInches objects.
//*********************************************
istream &operator>>(istream &strm, FeetInches &obj)
{
   strm >> obj.feet >> obj.inches;
   return strm;
}
```

● **Sample Program:**

```
// This program demonstrates the << and >> operators,
// overloaded to work with the FeetInches class.
#include <iostream>
#include "feetinch5.h"
using namespace std;

int main()
{
   FeetInches first, second;

   cout << "Enter a distance in feet and inches:\n";
   cin >> first;
   cout << "Enter another distance in feet and inches:\n";
   cin >> second;
   cout << "The values you entered are:\n";
   cout << first << " and " << second << endl;
   return 0;
}
```

Overloading The [] Operator

- Overloading the [] operator gives you the ability to create classes with array-like behavior. (For example, the string class overloads the [] operator so that you can access individual characters stored in string class objects.)

- Can be used to provide bounds checking.

- The overloaded operator [] function can only have **a single input parameter.**

- The overloaded operator [] function **must return a reference to an object**, not an object itself.

- **Example: Class Specification File**

```
#ifndef INTARRAY_H
#define INTARRAY_H
#include <iostream>
using namespace std;

class IntArray
{
  private:
    int *aptr;
    int arraySize;
    void subError();             // Handles subscripts out of range
  public:
    IntArray(int);                 // Constructor
    IntArray(const IntArray &);  // Copy constructor
    ~IntArray();                   // Destructor
    int size()
    {
       return arraySize;
    }
    int &operator[](int);       // Overloaded [] operator
};
#endif
```

● **Example: Class Implementation File**

```
#include "intarray.h"

//*******************************************************
// Constructor for IntArray class. Sets the size of
// the array and allocates memory for it.
//*******************************************************
IntArray::IntArray(int s)
{
   arraySize = s;
   aptr = new int [s];
   for (int count = 0; count < arraySize; count++)
      *(aptr + count) = 0;
}

//*******************************************************
// Copy constructor for IntArray class.
//*******************************************************
IntArray::IntArray(const IntArray &obj)
{
   arraySize = obj.arraySize;
   aptr = new int [arraySize];
   for(int count = 0; count < arraySize; count++)
      *(aptr + count) = *(obj.aptr + count);
}

//*******************************************************
// Destructor for IntArray class.
//*******************************************************
IntArray::~IntArray()
{
   if (arraySize > 0)
      delete [] aptr;
}
```

```
//*******************************************
// subError function. Displays an error message and
// exits the program when a subscript is out of range.
//*******************************************
void IntArray::subError()
{
   cout << "ERROR: Subscript out of range.\n";
   exit(0);
}

//*******************************************
// Overloaded [] operator. The argument is a subscript
// This function returns a reference to the element
// in the array indexed by the subscript.
//*******************************************
int &IntArray::operator[](int sub)
{
   if (sub < 0 || sub >= arraySize)
      subError();
   return aptr[sub];
}
```

● **Sample Program:**

```
// This program demonstrates a class that behaves like an array.
#include <iostream>
#include "intarray.h"
using namespace std;

int main()
{
   IntArray table(10);

   // Store values in the array.
   for (int x = 0; x < table.size(); x++)
      table[x] = (x * 2);          //table[x] is a reference to an int

   // Display the values in the array.
   for (int x = 0; x < table.size(); x++)
      cout << table[x] << " ";
   cout << endl;

   // Use the built-in + operator on array elements.
   for (int x = 0; x < table.size(); x++)
      table[x] = table[x] + 5;

   // Display the values in the array.
   for (int x = 0; x < table.size(); x++)
      cout << table[x] << " ";
   cout << endl;

   // Use the built-in ++ operator on array elements.
   for (int x = 0; x < table.size(); x++)
      table[x]++;

   // Display the values in the array.
   for (int x = 0; x < table.size(); x++)
      cout << table[x] << " ";

   cout << endl;
   return 0;
}
```

Type Conversion Operators

- **Conversion Operators** are member functions that tell the compiler how to convert an object of the class type to a value of another type.

- The conversion information provided by a conversion operator is automatically used by the compiler in assignments, initializations, and parameter passing.

- A conversion operator must be a member function of the class you are converting from.

- The **name of the operator is the name of the type** you are converting to.

- A conversion operator does not specify a return type.

- Example - Convert from a class IntClass object to an integer:

```
class IntClass
{
    int value;
  public:
    IntClass(int a = 0)          //constructor
    {
        value = a;
    }
    operator int()               //conversion operator
    {
        return value;
    }
};
```

- Sample Usage - Automatic Conversion During Assignment:

```
IntClass obj(15);
int i;

i = obj;               // automatic conversion
cout << i;             // prints 15
```

Convert Constructors

- **Convert Constructors** provide a method to convert a value of a given data type to an object of the class.
- A convert constructor takes a single parameter of a type other than its class type.
- Example - Convert from an integer to a class IntClass object:

```
class IntClass {
    int value;
  public:
    IntClass(int intValue)               //convert constructor
    {
        value = intValue;
    }
    int getValue()
    {
        return value;
    }
};
```

- Note: The **string** class has a convert constructor that converts from C-strings:

```
class string {
  public:
    string(char *);                      //convert constructor
    ...
};
```

- Convert constructors are automatically invoked by the compiler to create an object from the value passed as a parameter.

```
    string s("hello!");                  //convert C-string
```

- C++ allows convert constructors to be invoked with assignment-like notation:

```
    string s = "hello!";                 //convert C-string
```

- Convert constructors allow functions that take the class type as a parameter to take parameters of other types.

Object Composition (The Has-A Relation)

- Object composition occurs when a class contains an object of another class (i.e., the **‘has-a**’ relation).
- Uses the same notation as for nested structures.
- See sample program.

Inheritance (The Is-A Relation)

- Inheritance is a way of creating a new class by starting with an existing class and adding new members.
- The new class can replace or extend the functionality of the existing class.
- Inheritance models the **'is-a'** relationship between classes
- The **existing class** is called the **base class** (or **parent class**, or **superclass**).
- The **new class** is called the **derived class** (or **child class**, or **subclass**).
- To define a class by inheritance, we need to specify the base class plus the additional members that the derived class adds to the base class.
- The derived class inherits all the characteristics of the base class (e.g., the member variables and member functions of the base class. **Note: private members of the base class are part of the derived class but can not be accessed directly by the derived class**).
- The derived class definition will contain the base class access specification which affects how members of the base class will be accessed by the member functions of the derived class, and by code outside the two classes.

● **Example - Class Specification File with Inheritance:**

```
#include <string>
using namespace std;

enum Discipline {
  ARCHEOLOGY, BIOLOGY, COMPUTER_SCIENCE
};

enum Classification {
  FRESHMAN, SOPHOMORE, JUNIOR, SENIOR
};

//Base Class:
class Person
{
  private:
    string name;
  public:
    Person()
    {
       setName("");
    }
    Person(string pName)
    {
       setName(pName);
    }
    void setName(string pName)
    {
       name = pName;
    }
    string getName()
    {
       return name;
    }
};
```

```
//Derived Class
class Student : public Person        //Note the single :
{
  private:
    Discipline major;
    Person *advisor;
  public:
    void setMajor(Discipline d)
    {
        major = d;
    }
    Discipline getMajor()
    {
        return major;
    }
    void setAdvisor(Person *p)
    {
        advisor = p;
    }
    Person *getAdvisor()
    {
        return advisor;
    }
};

//Derived Class
class Faculty : public Person
{
  private:
    Discipline department;
  public:
    void setDepartment(Discipline d)
    {
        department = d;
    }
    Discipline getDepartment( )
    {
        return department;
    }
};
```

● **Sample Program:**

```
// This program demonstrates the creation and use
// of objects of derived classes.
#include <iostream>
#include "inheritance.h"

using namespace std;

// These arrays of string are used to print the
// enumerated types.
const string dName[] = {
   "Archeology", "Biology", "Computer Science"
};

const string cName[] = {
   "Freshman", "Sophomore", "Junior", "Senior"
};

int main()
{
   // Create a  Faculty object.
   Faculty prof;

   // Use a Person member function.
   prof.setName("Indiana Jones");

   // Use a Faculty member function.
   prof.setDepartment(ARCHEOLOGY);

   cout << "Professor " << prof.getName()
      << " teaches in the " << "Department of ";

   // Get department as an enumerated type.
   Discipline dept = prof.getDepartment();

   // Print out the department in string form.
   cout << dName[dept] << endl;

   return 0;
}
```

Protected Members and Class Access

- **Protected Members** of a base class are like private members except that they can be accessed by derived classes.

- **Base Class Access Specification** determines how private, protected, and public base class members are accessed when they are inherited by the derived class (Note: Member Access Specification determines general (global) access to member defined in the class.):

- C++ supports three **inheritance modes**, also called **base class access modes**:
  - public inheritance
  class Child : public Parent { };
  - private base class members are inaccessible to the derived class
  - protected base class members are protected members of the derived class
  - public base class members are public members of the derived class

  - protected inheritance
  class Child : protected Parent{ };
  - private base class members are inaccessible to the derived class
  - protected base class members are protected members of the derived class
  - public base class members are protected members of the derived class

  - private inheritance
  class Child : private Parent{ };
  - private base class members are inaccessible to the derived class
  - protected base class members are private members of the derived class
  - public base class members are private members of the derived class

Constructors, Destructors, and Inheritance

- By inheriting every member of the base class, a derived class object contains a base class object.

- The derived class constructor can specify which base class constructor should be used to initialize the base class object.

- When an object of a derived class is created, **the base class's constructor is called first**, followed by the derived class's constructor.

- When an object of a derived class is destroyed, **the derived class's destructor is called first**, followed by the destructor of the base class.

● **Demo Program:**

```
// This program demonstrates the order in which base and
// derived class constructors and destructors are called.
// For the sake of simplicity, all the class declarations
// are in this file.
#include <iostream>
using namespace std;

// Base class.
class BaseDemo
{
  public:
    BaseDemo()  // Constructor
        { cout << "This is the BaseDemo constructor.\n"; }
    ~BaseDemo() // Destructor
        { cout << "This is the BaseDemo destructor.\n"; }
};

// Derived class.
class DeriDemo : public BaseDemo
{
  public:
    DeriDemo()  //Constructor
        { cout << "This is the DeriDemo constructor.\n"; }
    ~DeriDemo()  // Destructor
        { cout << "This is the DeriDemo destructor.\n"; }
};

int main()
{
    cout << "We will now create a DeriDemo object." << endl;
    DeriDemo object;
    cout << endl;
    cout << "The program is now going to end." << endl;

    return 0;
}
```

Passing Arguments to Base Class Constructors

- Allows for selection between multiple base class constructors.
- Can specify arguments to the base constructor on the derived constructor heading.
- Can also be done with inline constructors.
- Must be done if base class has no default constructor.
- It is important to remember that the arguments to the base class constructor **must be specified in the definition (implementation) of the derived class**, rather than in its declaration.

Overriding Base Class Functions

- A derived class can **override** a member function of its base class by defining a derived class member function with the same name and parameter list.

- Typically used to replace a function in the base class with different actions in the derived class.

- **Not the same as overloading.** With overloading, the parameter lists must be different

- When a function is overridden, all objects of the derived class use the overriding function.

- If it is necessary to also access the overridden version of the function, it can be done using the scope resolution operator with the name of the base class and the name of the function:
  ```
  Student::getName();
  ```