CLASSES

OVERVIEW

OVERVIEW

- **In this section, we will see how to define, implement and use classes in object oriented programs**

- **What is a class?**
  - A class is a user defined data type that contain variables (called attributes) and a collection of operations on these variables (called methods)
  - The primary advantage of classes is that they give us a natural way to create robust and reliable code that can be reused in a wide range of applications

OVERVIEW

- **A class is normally created by one programmer and used by many other programmers**
  - Only the creator needs to know implementation details
  - Users can ignore details and build code on top of the class
  - This allows teams of programmers to work on separate classes to build very large and complex applications

- **Class libraries**
  - The standard C++ class library contains dozens of general purpose classes that can be used in any program
  - We have already been using the string, cin, cout, ifstream, and ofstream classes in our programs

OVERVIEW

- **To define a class,**
  - List the data fields inside the class
  - List the functions/methods that operate on this data
- **To implement a class**
  - Implement constructor functions to initialize data fields
  - Implement methods to perform data operations
- **To use a class**
  - Declare objects of the class
  - Call methods on these objects

OVERVIEW

- **Lesson objectives:**
  - Learn how to create and use simple classes
  - Learn how to create and use composite classes
  - Study example programs with classes
  - Complete online labs on classes
  - Complete programming project using classes

CLASSES

PART 1

DEFINING CLASSES

DEFINING CLASSES

- **The main purpose of a class is to bundle together the data and operations that make up an abstract data type**
  - We must give variable declarations for all of the data fields that make up the abstract data type
  - We must give function prototypes for all of the methods that operate on these data fields

- **We must also specify how the class can be used**
  - We must specify which of the variables and functions are public and can be accessed directly by users of this class
  - We must also specify which of the variables and functions are private and hidden from users of this class

DEFINING CLASSES

- **Overview of the C++ "class" definition syntax**

```
class class_name
{
private:
   data_type variable_name;
   data_type variable_name;
   …
```

We give the name of the class here

DEFINING CLASSES

- **Overview of the C++ "class" syntax**

```
class class_name
{
private:
   data_type variable_name;
   data_type variable_name;
   …
```

Everything after the word “private” is hidden from users of the class

DEFINING CLASSES

- **Overview of the C++ "class" syntax**

```
class class_name
{
private:
   data_type variable_name;
   data_type variable_name;
   …
```

These variable declarations define the data fields inside the class that make up the abstract data type

DEFINING CLASSES

```
…
public:
   class_name();
   ~class_name();

   return_type method_name( parameter_list );
   return_type method_name( parameter_list );
   return_type method_name( parameter_list );
};
```

Everything after the word “public” is visible to users of the class

DEFINING CLASSES

```
…
public:
   class_name();
   ~class_name();

   return_type method_name( parameter_list );
   return_type method_name( parameter_list );
   return_type method_name( parameter_list );
};
```

The constructor function has the same name as the class, it is used to initialize data fields

DEFINING CLASSES

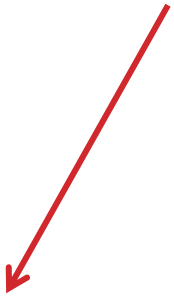
```
…
public:
   class_name();
   ~class_name();

   return_type method_name( parameter_list );
   return_type method_name( parameter_list );
   return_type method_name( parameter_list );
};
```

The destructor function has the same name as the class with a tilda character in front, it is used to finalize data fields

DEFINING CLASSES

```
…
public:
   class_name();
   ~class_name();

   return_type method_name( parameter_list );
   return_type method_name( parameter_list );
   return_type method_name( parameter_list );
};
```

These function prototypes specify the methods that implement operations on the data fields



DEFINING CLASSES

```
…
public:
   class_name();
   ~class_name();

   return_type method_name( parameter_list );
   return_type method_name( parameter_list );
   return_type method_name( parameter_list );
};
```

We need to put a semicolon here after the curly bracket

DEFINING CLASSES

- **Consider the problem of keeping track of the time of day in a program**
  - We need an integer hour value [0..23]
  - We need an integer minute value [0..59]
  - We need an integer second value [0..59]

- **We need to operations that safely manipulate the hour, minute, second values**
  - Provide functions to access/modify time values
  - Provide functions to input/output time values
  - Make sure the user can not create invalid times

DEFINING CLASSES

- **Let us consider the problem of creating a Time class**

- **We declare variables for all of the data fields in the "private" section of the Time class**
  - Use integers for hour, minute, second values

- **We give function headers for the constructor/destructor and all methods in the "public" section of the Time class**
  - Use “Get” method to access data fields
  - Use “Set” method to modify data fields
  - Use “Read” method to input time data fields
  - Use “Print” method to output time data fields

DEFINING CLASSES

```
class Time
{
private:
    int hour;
    int minute;
    int second;
…
```

These variable declarations define the data fields inside the Time class

DEFINING CLASSES

```
  …
public:
     Time();
    ~Time();

     void Set(const int Hr, const int Min, const int Sec);
     void Get(int &Hr, int &Min, int &Sec) const;
     void Read();
     void Print() const;
};
```

The constructor / destructor functions for the Time class

DEFINING CLASSES

```
  …
public:
     Time();
    ~Time();

     void Set(const int Hr, const int Min, const int Sec);
     void Get(int &Hr, int &Min, int &Sec) const;
     void Read();
     void Print() const;
};
```

These class methods define operations on Time data fields

DEFINING CLASSES

- **The "Set" method has three value parameters that allow the user to specify the three Time data fields**

```
void Set(const int Hr, const int Min, const int Sec);
```

- **The "Get" method has three reference parameters that allow the user to see the values of the Time data fields**

```
void Get(int &Hr, int &Min, int &Sec) const;
```

DEFINING CLASSES

- **The “Read” method has no parameters because the function will prompt the user to enter Time data**

```
void Read();
```

- **The “Print” method has no parameters because the function simply prints the Time data**

```
void Print() const;
```

DEFINING CLASSES

- **We have yet another use of "const" in the method headers**

  void Get(int &Hr, int &Min, int &Sec) const;

  void Print() const;

- **The "const" tells the compiler that these methods do NOT change any of the variables in the object**
  - These are called accessor methods

- **Methods without the "const" ARE allowed to change any of the variables in the object**
  - These are called mutator methods

DEFINING CLASSES

- **It is possible to extend the Time class in many ways…**
  - Add more data fields (eg. days, microseconds)
  - Add more methods to manipulate Time values
  - A function to print time in military time
  - A function to compare two time values
  - A function to add H hours, M minutes, S seconds
  - A function to subtract H hours, M minutes, S seconds

DEFINING CLASSES

- **Where do we put the Time class definition?**
  - By convention, the definition of a class is stored in a "header file" with the same name as the class
  - For example, the Time class would be stored in "time.h"

- **How can we use the Time class in our program?**
  - We must add #include "time.h" at the top of our main program to define the Time class in our program
  - By convention, the implementation of the Time class is stored in "time.cpp"
  - We compile the Time class and our main program using: g++ -Wall time.cpp main.cpp

DEFINING CLASSES

- **To include a user defined class we use quotes " "**

  #include "time.h"

The compiler will look in your current directory for this file

- **To include predefined C++ class we use < >**

  #include <cmath>

  #include <cstdlib>

  #include <iostream>

  #include <fstream>

The compiler will look in the C++ class library folder for these files

SUMMARY

- **A class is used to bundle together the data and operations that make up an abstract data type**
  - Data fields are stored in class variables (attributes)
  - Operations on this data are defined by methods

- **The class definition also tells us how to use a class**
  - The "public section" lists variables and methods that can be accessed by users of the class
  - The "private section" lists variables and methods that are hidden from users of the class

CLASSES

PART 2

IMPLEMENTING AND USING CLASSES

IMPLEMENTING CLASSES

- **Lets see how classes are implemented**
  - Methods are implemented just like regular functions
  - We must add “class name::” before the method name
  - This tells the C++ compiler that this method has access to the private variables of the class

```
return_type class_name::method_name( parameter_list )
{
 // Code for method goes here
}
```

IMPLEMENTING CLASSES

- **Easy way to start is to create "skeleton methods"**
  - Copy and paste the method headers from class definition
  - Remove the semicolon at the end of each header
  - Add “class_name::” before the method_name
  - Add a debugging statement to print the method name

```
return_type class_name::method_name( parameter_list )
{
 cout << "method_name\n";
}
```

IMPLEMENTING CLASSES

- **Now lets implement the Time class**
  - First, we create a file called “time.cpp” to contain the Time class implementation
  - At the top of time.cpp we must add #include “time.h” to define the Time class
  - Next, we implement of all Time methods in “time.cpp” in the same order as they appear in “time.h”
  - It is always a good idea to start with skeleton methods to debug the parameter passing and returns types

IMPLEMENTING CLASSES

```
#include “time.h”

Time::Time()
{
      cout << "Constructor\n";
}

Time::~Time()
{
      cout << "Destructor\n";
}
```

The constructor and destructor methods are unique and do NOT have any return types (not even void)

IMPLEMENTING CLASSES

```
void Time::Set(const int Hr, const int Min, const int Sec)
{
	cout << "Set\n";
}

void Time::Get(int &Hr, int &Min, int &Sec) const
{
	cout << "Get\n";
}
```

IMPLEMENTING CLASSES

```
void Time::Read()
{
      cout << ”Read\n";
}

void Time::Print() const
{
      cout << "Print\n";
}
```

IMPLEMENTING CLASSES

- **How do we compile the Time class?**
  - We use “g++ -Wall -c time.cpp” to check the syntax of time.cpp and create an intermediate output file “time.o”

- **After we have the "skeleton methods" compiling**
  - Add the desired code for each method one at a time
  - Compile and debug each method one at a time
  - This is a classic "incremental development" technique
  - We always have a compiling / running program !!!

IMPLEMENTING CLASSES

```
// Time constructor method
Time::Time()
{
   cout << "Constructor\n";
   hour = 0;
   minute = 0;
   second = 0;
}
```

Give all of the private variables some initial values

IMPLEMENTING CLASSES

```
// Time destructor method
Time::~Time()
{
   cout << "Destructor\n";
   hour = 0;
   minute = 0;
   second = 0;
}
```

We don’t really have to do this but it doesn’t hurt to clear the memory of old data

IMPLEMENTING CLASSES

```
// Time Set method
void Time::Set(const int Hr, const int Min, const int Sec)
{
   cout << "Set\n";
   hour = Hr;
   minute = Min;
   second = Sec;
}
```

Later, we should add some error checking to make sure the input Time is valid

IMPLEMENTING CLASSES

```
// Time Get method
void Time::Get(int &Hr, int &Min, int &Sec) const
{
   cout << "Get\n";
   Hr = hour;
   Min = minute;
   Sec = second;
}
```

We fill in multiple reference parameters to give data back to the main program

IMPLEMENTING CLASSES

```
// Time Read method
void Time::Read()
{
   cout << "Enter hour: ";
   cin >> hour;
   cout << "Enter minute: ";
   cin >> minute;
   cout << "Enter second: ”;
   cin >> second;
}
```

Later, we should add some error checking to make sure the input Time is valid

IMPLEMENTING CLASSES

```
// Time Print method
void Time::Print() const
{
   if (hour < 10) cout << "0";
   cout << hour << ":";
   if (minute < 10) cout << "0";
   cout << minute << ":";
   if (second < 10) cout << "0";
   cout << second << endl;
}
```

This will print Time values in the 00:00:00 format

IMPLEMENTING CLASSES

- **How should we test that a Time value is valid?**
  - We need to check that the hour, minute, second values are within their expected 0..23, 0..59, 0..59 ranges

- **How should we correct invalid Time values?**
  - Simple solution uses modulo arithmetic to “throw away” any overflow or underflow that occurs
    (10:66:90 becomes 10:06:30)
  - Fancy solution “wraps around” the hour, minute, second values if they overflow or underflow
    (10:66:90 becomes 11:07:30)

IMPLEMENTING CLASSES

```
// Simple time validation
hour = hour % 24;
minute = minute % 60;
second = second % 60;
```

This method will “throw away” any value overflow
It will not change valid hour, minute, second values

IMPLEMENTING CLASSES

```
// Fancy time validation
minute = minute + second / 60;
second = second % 60;
hour = hour + minute / 60;
minute = minute % 60;
hour = hour % 24;
```

This method will “wrap around” any value overflow
It will not change valid hour, minute, second values

USING CLASSES

- **Using classes is a three step process**

  1) Include the class definition at top of program
     #include <class_name> for built in classes
     #include “class_name.h” for user defined classes

  2) Declare objects of the class like we declare variables
     class_name object_name;

  3) Use object by calling methods using the “dot notation”
     object_name.method_name();
     object_name.method_name(param1, param2);

USING CLASSES

- **The compiler will look at the class definition to check that we are using a class properly**

- **The compiler WILL allow us to call public class methods in the class using the dot notation**

```
object_name.method_name( param1, param2 );
```

- **The compiler will NOT allow us to access the private data fields in the class using the dot notation**

```
object_name.variable_name = 42;
```

This will cause a compiler error

USING CLASSES

- **We have actually been using classes for some time because string is a built in class on most systems**

```
#include <string>
```
← This lets us use strings

```
string name;
```
← This creates a string object

```
name = “george”;
name[0] = ‘G’;
int len = name.length();
```
← We use the string object by calling methods in string class using the “dot notation”

USING CLASSES

- **We have also been using classes for file input/output because ifstream and ofstream are also built in classes**

```
#include <fstream>
```
← This lets us use file streams

```
ifstream din;
```
← This creates an ifstream object

```
din.open("junk.txt");
din >> value;
din.close();
```
We use the ifstream object by calling methods in ifstream class using the “dot notation”

USING CLASSES

- **Now lets see how we can use the Time class using the three step process described above**

1) Include the Time class

   #include “time.h”

2) Declare Time objects

   Time Now;

3) Use Time methods on objects

   Now.Print();

USING CLASSES

```
#include “time.h”
```

This lets us use the Time class in this program

```
// Simple program using the Time class
int main()
{
   int Hr, Min, Sec;
   Time Now, Then;
```

Declaring two Time objects to store time information

```
   Now.Print();
   Then.Print();
```

This prints out the default values of the Time objects

USING CLASSES

```
Now.Set( 10, 30, 0 );
Now.Print();
```

This will set the fields in the Now object and then print 10:30:00

```
Then = Now;
Then.Print();
```

This will copy all time data from Now object into Then object and print 10:30:00

USING CLASSES

```
   Now.Get( Hr, Min, Sec );
   cout << Hr << " "
        << Min << " "
        << Sec << endl;

   Now.minute = 42;
   cout << Then.hour;
}
```

This will extract info from the Now object and print 10 30 0

This code will NOT compile because these data fields are private variables of the Time class

USING CLASSES

- **How should we compile and run this program?**
  - Assume time.h contains the class definition
  - Assume time.cpp contains the class implementation
  - Assume main.cpp contains the program
  - Use “g++ -o main.exe time.cpp main.cpp” to compile both of the C++ files and create main.exe

SUMMARY

- **In this section, we saw how to implement and use classes in sample programs**
  - Define class in “class.h”
  - Implement class methods in “class.cpp”
  - Implement main program in “main.cpp”

- **This separation means that users of a class only need to look at the class definition, and not the implementation**
  - For example, you have been using cin/cout/strings without looking at their implementations
  - This is one of the main advantages of data abstraction and object oriented programming

CLASSES

PART 3

SIMPLE CLASS EXAMPLES

SIMPLE CLASS EXAMPLES

- **The goal of object oriented programming is to create applications that build upon a collection of a classes**

- **There are three steps to this process:**
  - Decide what information is needed to describe object
    - What private variables to declare
  - Decide what operations on the object are necessary
    - What public methods to create
  - Build one or more applications using class
    - How to create and use objects in a program

SIMPLE CLASS EXAMPLES

- **In this section, we will illustrate object oriented programming by creating two simple classes:**
  - Student class
    - Stores basic information about a student
    - Very basic operations to access information
    - Could be used as part of large university database
  - Linear class
    - Store information about linear equations
    - Classic mathematical operations for linear equations
    - Could be used in an engineering application

STUDENT CLASS

- **What student information might be of interest?**
  - Student ID number (int)
  - First name, middle name, last name (string)
  - Home address, campus address (string)
  - ACT, SAT test scores (int)
  - Undergraduate major (string)
  - Current GPA (float)

- **We store information in private variables in the class**

STUDENT CLASS

- **What operations could we perform on a student?**
  - Change address
  - Update test scores
  - Change major
  - Update GPA
  - Print all information

- **We use get and set methods to implement operations**

STUDENT CLASS

```
class Student
{
 private:
   int ID;
   string Name;
   string Address;
   float GPA;

public:
   Student();
 ~Student();
```

We are using four private variables here but more data fields could be added

Standard constructor and destructor methods

STUDENT CLASS

```
…
   int getID() const;
   string getName() const;
   string getAddress() const;
   float getGPA() const;

   void setID(const int id);
   void setName(const string name);
   void setAddress(const string address);
   void setGPA(const float gpa);

   void print() const;
};
```

One get method for each private variable

One set method for each private variable

STUDENT CLASS

```
Student::Student()
{
   // Initialize private variables
   ID = 0;
   Name = "none";
   Address = "none";
   GPA = 0.0;
}

Student::~Student()
{
   // Empty
}
```

Constructor implementation

Destructor implementation

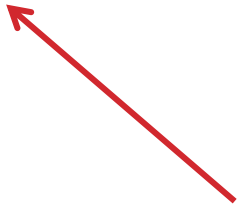
STUDENT CLASS

```
int Student::getID() const { return ID; }
string Student::getName() const { return Name; }
string Student::getAddress() const { return Address; }
float Student::getGPA() const { return GPA; }

void Student::setID(const int id) { ID = id; }
void Student::setName(const string name) { Name = name; }
void Student::setAddress(const string address) { Address = address; }
void Student::setGPA(const float gpa) { GPA = gpa; }
```

One line get set methods can be used to save space in the program

STUDENT CLASS

```
void Student::print()
{
   cout << "ID: " << ID << endl
        << "Name: " << Name << endl
        << "Address: " << Address << endl
        << "GPA: " << GPA<< endl;
}
```



The format of the output may depend on the needs of the application

STUDENT CLASS

```
int main()
{
   cout << "Testing the Student class\n";
   Student student;
   student.setID(123456);
   student.setName("John Gauch");
   student.setAddress("518 JB Hunt");
   student.setGPA(3.14);
   student.print();
}
```

We can test the Student class by calling each of the methods

LINEAR CLASS

- **How can we represent a linear equation?**
  - Slope intercept formula: $y = mx + b$
    - Store m, b values
  - Geometric formula: $(n_x,n_y) \cdot (x,y) = d$
    - Store normal (n_x,n_y) and distance from origin d
  - Parametric formula: $(x_1,y_1) + t\,(x_2-x_1,y_2-y_1)$
    - Store points on line (x_1,y_1) and (x_2,y_2)
  - Classic formula: $ax + by + c = 0$
    - Store a, b, c values

- **We only have to store the linear equation in one way, and we can convert to any of the other representations**

LINEAR CLASS

- **What operations could we perform on a linear equation?**
  - Get and set the line equation coefficients
  - Print the line in y=mx+b or ax+by+c=0 format
  - Check if line is vertical or horizontal
  - Check if two lines are parallel or perpendicular
  - Solve for x when given y
  - Solve for y when given x
  - Calculate the intersection point of two lines
  - Calculate distance from a point to the line

- **Users of this class do not need to know how these operations are implemented – just how to call them**

LINEAR CLASS

```
class Linear
{
 private:
   float A, B, C;

 public:
   Linear();
 ~Linear();
   void SetCoefficients(const float a, const float b, const float c);
   void GetCoefficients(float &a, float &b, float &c) const;
   void PrintABC() const;
   void PrintMB() const;
```

We are using the
Ax + By + C = 0
line representation

We are using methods to get/set all three line coefficients at one time

LINEAR CLASS

```
   …
   bool Vertical() const;
   bool Horizontal() const;
   bool Parallel(Linear eq) const;
   bool Perpendicular(Linear eq) const;


   bool SolveX(float &x, const float y) const;
   bool SolveY(const float x, float &y) const;
   bool Intersect(Linear eq, float &x, float &y) const;
   bool Distance(float x, float y, float &dist) const;
};
```

Return true/false answer to line property question

Return true/false error checking status flag

Calculation results are stored in reference parameters

LINEAR CLASS

```
void Linear::SetCoefficients(const float a, const float b, const float c)
{
   A = a;
   B = b;
   C = c;
}

void Linear::GetCoefficients(float &a, float &b, float &c) const
{
   a = A;
   b = B;
   c = C;
}
```

LINEAR CLASS

```
void Linear::PrintABC() const
{
   cout << A << "x + " << B << "y + " << C << " = 0\n";
}

void Linear::PrintMB() const
{
   if (B != 0)
      cout << "y = " << -A/B << "x + " << -C/B << endl;
   else if (A != 0)
      cout << "x = " << -C/A << endl;
   else
      cout << C << " = 0\n";
}
```

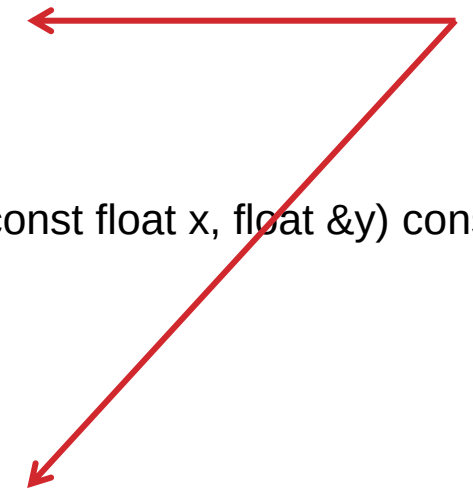
LINEAR CLASS

```
bool Linear::Horizontal() const { return (A == 0.0); }
bool Linear::Vertical() const { return (B == 0.0); }

bool Linear::Parallel(Linear eq) const
{  // check that slope of one line (-A/B) equals slope of the other line (-eq.A/eq.B)
   // to avoid potential divide by zero we multiply equation by both denominators
   return (A * eq.B == B * eq.A);
}

bool Linear::Perpendicular(Linear eq) const
{  // check that slope of one line (-A/B) is equal to the
   // negative inverse slope of the other line (eq.B/eq.A)
   // to avoid potential divide by zero we multiply equation by both denominators
   return (- A * eq.A == B * eq.B);
}
```

LINEAR CLASS

```
bool Linear::SolveX(float &x, const float y) const
{
   if (Horizontal())
      return false;
   x = -(B * y + C) / A;
   return true;
}
bool Linear::SolveY(const float x, float &y) const
{
   if (Vertical())
      return false;
   y = -(A * x + C) / B;
   return true;
}
```

We perform error checking before solving for x or y, and return a true/false error status



LINEAR CLASS

```
bool Linear::Intersect(Linear eq, float &x, float &y) const
{
   // Error checking
   if (Parallel(eq))
      return false;


   // Solve for x and y
   x = (B * eq.C - C * eq.B) / (A * eq.B - B * eq.A);
   y = (A * eq.C - C * eq.A) / (B * eq.A - A * eq.B);
   return true;
}
```

Our error checking will avoid potential divide-by-zero errors in this calculation

LINEAR CLASS

```
bool Linear::Distance(float x, float y, float &dist) const
{
    // Error checking
   if ((A == 0) && (B == 0))
      return false;


   // Calculate distance from line
   dist = fabs(A * x + B * y + C) / sqrt(A * A + B * B);
   return true;
}
```

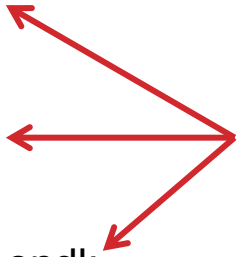
Our error checking will avoid potential divide-by-zero errors in this calculation

LINEAR CLASS

```
int main()
{
   cout << "Testing the Linear class\n";
   float a,b,c, x, y, dist;
   cout << "Enter line coefficients: ";
   cin >> a >> b >> c;

   // Create a linear equation
   Linear eq1;
   eq1.SetCoefficients(a, b, c);
   eq1.PrintABC();
   eq1.PrintMB();
```

LINEAR CLASS

```
// Test some basic line properties
cout << "Vertical test: " << eq1.Vertical() << endl;
cout << "Horizontal test: " << eq1.Horizontal() << endl;

// Test some basic line calculations
if (eq1.SolveX(x, 1))
   cout << "SolveX(1): " << x << endl;
if (eq1.SolveY(1, y))
   cout << "SolveY(1): " << y << endl;
if (eq1.Distance(0, 0, dist))
   cout << "Distance(0,0): " << dist << endl;
```



We only print results if the calculation was successful

LINEAR CLASS

```
  // Create a second linear equation
  cout << "Enter line coefficients: ";
  cin >> a >> b >> c;
  Linear eq2;
  eq2.SetCoefficients(a, b, c);

  // Test advanced line methods
  cout << "Parallel test: " << eq1.Parallel(eq2) << endl;
  cout << "Perpendicular test: " << eq1.Perpendicular(eq2) << endl;
  if (eq1.Intersect(eq2, x, y))
    cout << "Intersection point: " << x << " " << y << endl;
}
```

When we print a boolean
0 will be printed if false
1 will be printed if true

SUMMARY

- **In this section, we showed how two simple classes could be defined, implemented, and used in a program**
  - The Student class illustrated how separate get/set methods could be used for each private variable
  - The Student class methods do not have any error checking, but this could be added (eg. GPA < 4.0)
  - The Linear class uses get/set methods with multiple parameters to access/store private variables
  - The Linear class illustrated how true/false status flags can be used to return error checking results

CLASSES

PART 4

ADVANCED CLASSES

ADVANCED CLASSES

- **Assignment of objects**
  - Assume Now and Then are objects of the same class
  - We CAN use "Now = Then;" to assign an object
  - The program will make a field-by-field copy of object

- **Comparison of two objects**
  - We can NOT use "if (Now == Then)" to compare objects
  - Instead we must compare two objects on a field-by-field basis inside a comparison method to see if they are equal
  - Eg: "if (Now.IsEqual(Then))"

ADVANCED CLASSES

- **Printing an object**
  - We can NOT just print using "cout << Now;"
  - Instead we call a method in the class
  - Eg: Now.Print();

- **Initialization of an object**
  - We can give initial values when declaring an object
  - We must create a constructor method with parameters
  - Eg: Time Later(12, 34, 56);
  - Same as Time Later; Later.set(12,34,56);

ADVANCED CLASSES

- **Objects as parameters**
  - Objects can used as value parameters or reference parameters in functions
  - Using reference parameters is slightly faster because the data does not need to be copied

- **Objects as return values**
  - An object can also be used as a return type for a function
  - This gives us a way to return more than one value at one time from the function

ADVANCED CLASSES

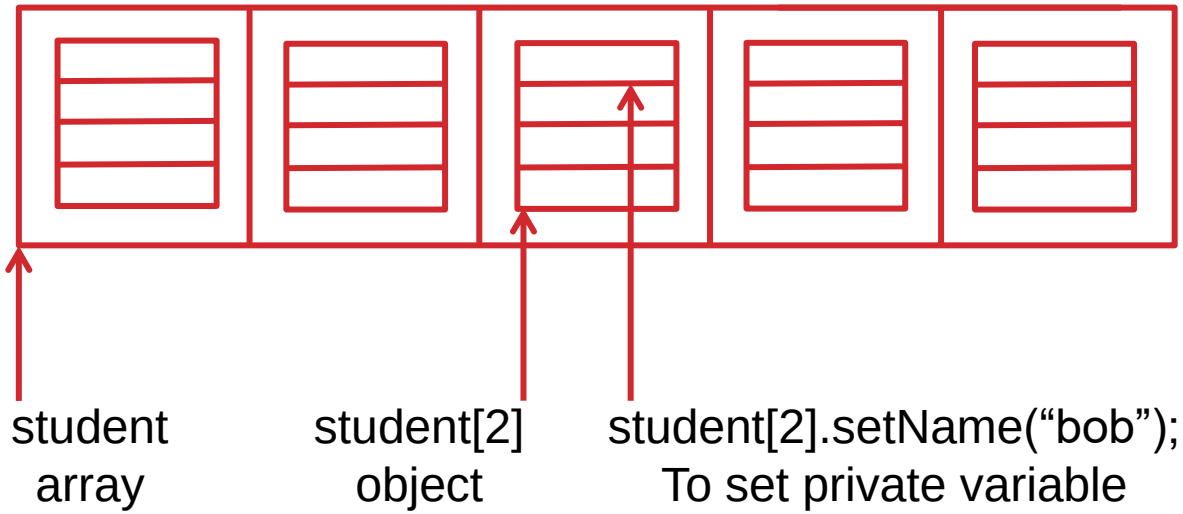
- **We are allowed to make methods in a class private**
  - Put the method prototype under "private"
  - Private methods can be called by other methods in the class, but not by users of this class in the main program
  - This is useful for basic operations we need to implement the class, but the user should never need to use
  - We could have a "CheckValid" method in the Time class to make sure the hour, minute, second represent a valid time

ADVANCED CLASSES

- **We are also allowed to make variables in a class public**
  - Put the variable declaration under "public"
  - Public variables can be read and modified by users of the class in the main program
  - Doing this will "break" the data hiding principal in object oriented programming
  - Some programmers will do this on purpose to avoid the computational overhead of get/set methods

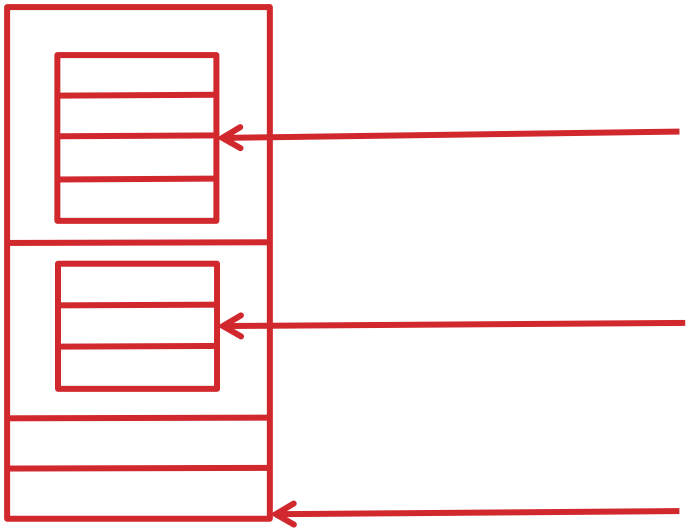
ADVANCED CLASSES

- **Arrays of objects**
  - An array of objects can be used to store data
  - "student_class student[5]" creates an array of objects to store all student information (name, address, major, gpa)



ADVANCED CLASSES

- **Nested classes**
  - A class can contain other classes as variables
  - By nesting classes we can build more complex ADTs
  - For example, we can store track and field race results using a class that contains two other classes



Person class keeps track of (name, address, age, gender)

Time class keeps track of (hour, minute seconds)

Result class keeps track of all track and field race results

DEFAULT CONSTRUCTOR

- **The default constructor is the method that is automatically called when you define objects using a class**

- **The default constructor normally has no parameters and sets all data fields in the class to some initial value**

  class_name();

- **We create object of this class using the following:**

  class_name object;

DEFAULT CONSTRUCTOR

- **The default constructor can also have one or more parameters with pre-defined initial values**

```
class_name(float param1=0, int param2=0);
```

- **You can create objects using any of the following:**

```
class_name object1;          // param1=0, param2=0
class_name object2(42);      // param1=42, param2=0
class_name object3(42, 17);  // param1=42, param2=17
```

  - Notice that the values provided above fill in parameters from left to right in the parameter list

DEFAULT CONSTRUCTOR

- **The implementation of a default constructor uses the input parameters to initialize the private fields**

```
class_name::class_name(float param1, int param2)
{
   field1 = param1;
   field2 = param2;
   …
}
```

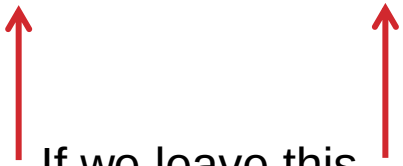
You do not need to repeat the default parameter values here

We initialize the data fields using parameter values provided by the user or the default values

COPY CONSTRUCTOR

- **The copy constructor is a special method that creates a new object by making a copy of another object**
  - This method is automatically called by the program when we pass an object by value into a function
  - The copy constructor must have one object passed in as a const reference parameter

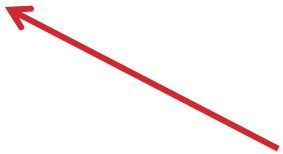
```
class_name(const class_name & object);
```



COPY CONSTRUCTOR

- **The implementation of a copy constructor simply copies the fields from the input object into the private fields**

```
class_name::class_name(const class_name & object)
{
   field1 = object.field1;
   field2 = object.field2;
   field3 = object.field3;
   …
}
```



We are allowed to access the data fields of object because we are inside one of the class methods

COPY CONSTRUCTOR

- **The copy constructor can also be used to create objects**

```
class_name object1;              // creates object1
object1.set(72.5, 49);           // stores 72.5 and 49 in object1
class_name object2(object1);  // object2 is a copy of object1
```

- **This can also be done using object assignment**

```
class_name object1;              // creates object1
object1.set(72.5, 49);           // stores 72.5 and 49 in object1
class_name object2;              // creates object2
object2 = object1;               // copies object1 into object2
```

COPY CONSTRUCTOR

- **Copy constructor is automatically called when passing an object by value into a function**
- **Add example here**

STATIC CONSTANTS

- **The keyword "static" can also be used in a class definition to create "static constants"**
  - Only one memory location is allocated for this static constant which is shared by all of the objects in the class
  - Most static constants are private but some are public

- **Static constants can be used to:**
  - Specify the size of a private array
  - Specify the min/max values on private variables
  - Specify boolean flags for debugging or printing
  - Specify mathematical constants

STATIC CONSTANTS

```
// Store information about a popular TV family
class family
{
public:
   …
private:
   static const int NUM_CHILDREN = 19;
   string mother, father;
   string children[NUM_CHILDREN];
   …
};
```

This constant is used inside the class to control loops reading or writing the children names

SUMMARY

- **In this section, we discussed the following:**
  - Assignment and comparison of objects
  - Using objects as parameters and return values
  - Private methods and public variables
  - Composite classes (arrays of objects, nested objects)
  - Default constructors
  - Copy constructors
  - Static constants

CLASSES

PART 5

ADVANCED CLASS EXAMPLES

ADVANCED CLASS EXAMPLES

- **Consider the problem of creating a 2D platform video game like Donkey Kong or Super Mario Bros**
  - We need to know the location of players on the screen
  - We need geometric models for platforms and objects
  - We need some way to store Points, Lines and Polygons

- **We can use a collection of classes to store this geometric information and implement operations on this data**
  - These classes will demonstrate many of the advanced features discussed in the previous section

POINT CLASS

- **What data do we need to store?**
  - For a 2D point we need the (x,y) coordinates

- **What operations do we need to implement?**
  - Basic get and set methods
  - Some way to print or display points
  - Distance between two points
  - Geometric transformations (translate, rotate, scale)

POINT CLASS

```
class Point
{
public:
   Point();
   Point(const Point &p);
 ~Point();

   void Set(const float X, const float Y);
   void Get(float &X, float &Y) const;
   void Print() const;
   …
```

Standard constructor, copy constructor and destructor methods

Get and set methods let user access (x, y) coordinates of Point

POINT CLASS

```
…
   float Distance(Point &p) const;
   void Translate(const float deltaX, const float deltaY);
   void Rotate(const float angle);
   void Scale(const float scale);

private:
   float x, y;
};
```

Geometric operations for Point objects

Coordinates of Point

POINT CLASS

```
Point::Point()
{
   x = 0;
   y = 0;
}
```

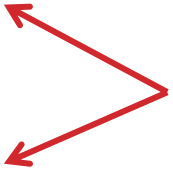
Constructor method

```
Point::Point(const Point &p)
{
   x = p.x;
   y = p.y;
}
```

Copy constructor method

POINT CLASS

```
void Point::Set(const float X, const float Y)
{
   x = X;
   y = Y;
}
void Point::Get(float &X, float &Y) const
{
   X = x;
   Y = y;
}
void Point::Print() const
{
   cout << "(" << x << "," << y << ")";
}
```



Set and Get methods

Print method

POINT CLASS

```
float Point::Distance(Point &p) const
{
   float dx = x - p.x;
   float dy = y - p.y;
   return sqrt(dx*dx + dy*dy);
}
```

Calculate distance between two Points

```
void Point::Translate(const float deltaX, const float deltaY)
{
   x += deltaX;
   y += deltaY;
}
```

Translate the (x,y) coordinates of Point

POINT CLASS

```
void Point::Rotate(const float angle)
{
   float newX = x * cos(angle) - y * sin(angle);
   float newY = x * sin(angle) + y * cos(angle);
   x = newX;
   y = newY;
}
```

Rotate the (x,y) coordinates of Point

```
void Point::Scale(const float scale)
{
   x *= scale;
   y *= scale;
}
```

Scale the (x,y) coordinates of Point

LINE CLASS

- **What data do we need to store?**
  - Lines can be defined in terms of two Points
  - From this, we can derive y=mx+b or Ax+By+C=0

- **What operations do we need to implement?**
  - Basic get and set methods
  - Some way to print or display lines
  - Distance between points and a line
  - Geometric transformations (translate, rotate, scale)

LINE CLASS

```
class Line
{
public:
   Line();
   Line(const Line &line);
 ~Line();

   void Set(const Point point1, const Point point2);
   void Get(Point &point1, Point &point2) const;
   void Print() const;
…
```

Standard constructor, copy constructor and destructor methods

Get and set methods let user access two Points that define Line

LINE CLASS

…

```
   float Distance(Point &point) const;
   void Translate(const float deltaX, const float deltaY);
   void Rotate(const float angle);
   void Scale(const float scale);

private:
   Point p1, p2;
};
```

Geometric operations for Line objects

The two Points that define the Line equation store (x1,y1) (x2,y2)

LINE CLASS

```
Line::Line()
{
   // Point constructors are called automatically
}

Line::Line(const Line &line)
{
   // Copy two Points that define Line
   p1 = line.p1;
   p2 = line.p2;
}
```

LINE CLASS

```
void Line::Set(const Point point1, const Point point2)
{
   // Copy parameters into private variables
   p1 = point1;
   p2 = point2;
}

void Line::Get(Point &point1, Point &point2) const
{
   // Copy private variables into parameters
   point1 = p1;
   point2 = p2;
}
```

LINE CLASS

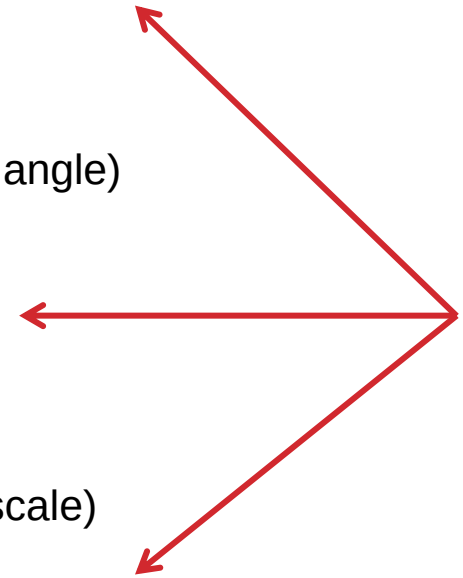
```
void Line::Print() const
{
   float x1, y1, x2, y2;
   p1.Get(x1,y1);
   p2.Get(x2,y2);

   float A = (y2-y1);
   float B = (x1-x2);
   float C = - x2*A - y2*B;
   cout << A << "x + " << B << "y + " << C << " = 0\n";
}
```

Get (x,y) coordinates
for both Points on line

Print line using
Ax+By+C=0 format

LINE CLASS

```
void Line::Translate(const float deltaX, const float deltaY)
{  p1.Translate(deltaX, deltaY);
   p2.Translate(deltaX, deltaY);
}

void Line::Rotate(const float angle)
{  p1.Rotate(angle);
   p2.Rotate(angle);
}

void Line::Scale(const float scale)
{  p1.Scale(scale);
   p2.Scale(scale);
}
```



Call methods in the Point class to perform geometric operations on Line objects

POLYGON CLASS

- **What data do we need to store?**
  - A polygon object is a closed sequence of line segments
  - We can define a polygon using an array of Points

- **What operations do we need to implement?**
  - Basic get and set methods
  - Some way to print or display points
  - Geometric transformations (translate, rotate, scale)

POLYGON CLASS

```
class Polygon
{
public:
   Polygon();
   Polygon(const Polygon &poly);
 ~Polygon();

   void Set(int index, const Point point);
   void Get(int index, Point &point) const;
   void Print() const;
…
```

Standard constructor, copy constructor and destructor methods

Get and set methods let user access each Point on polygon

POLYGON CLASS

```
…

   void Translate(const float deltaX, const float deltaY);
   void Rotate(const float angle);
   void Scale(const float scale);

private:
   static const int MAX_POINT_COUNT = 10;
   Point point_array[MAX_POINT_COUNT];
   int point_count;
};
```

Geometric operations for Polygon objects

An array of Points is used to define Polygon object

Number of Points that make up this Polygon

POLYGON CLASS

```
Polygon::Polygon()
{
   // Set Point count to zero
   point_count = 0;
}

Polygon::Polygon(const Polygon &poly)
{
   // Copy Point information
   point_count = poly.point_count;
   for (int index = 0; index <= point_count; index++)
      point_array[index] = poly.point_array[index];
}
```

POLYGON CLASS

```
void Polygon::Set(int index, const Point point)
{
   // Copy parameters into private variables
   if ((index >= 0) && (index < MAX_POINT_COUNT))
   {
      point_array[index] = point;
      if (point_count < index)
         point_count = index;
   }
}
```

Updates point_count as Points are added to the private point_array

POLYGON CLASS

```
void Polygon::Get(int index, Point &point) const
{
   // Copy private variables into parameters
   if ((index >= 0) && (index < point_count))
      point = point_array[index];
}
```

Checks that index is within valid range before returning Point

POLYGON CLASS

```
void Polygon::Print() const
{
   for (int index = 0; index <= point_count; index++)
   {
      cout << "Point[" << index << "]=";
      point_array[index].Print();
      cout << endl;
   }
   cout << endl;
}
```

Loop over all Points

Call the Point print method to display the (x,y) coordinates of point_array[index]

POLYGON CLASS

```
void Polygon::Translate(const float deltaX, const float deltaY)
{
   for (int index = 0; index <= point_count; index++)
      point_array[index].Translate(deltaX, deltaY);
}

void Polygon::Rotate(const float angle)
{
   for (int index = 0; index <= point_count; index++)
      point_array[index].Rotate(angle);
}
```

Loop over all Points

Call Point methods to translate or rotate the (x,y) coordinates of point_array[index]

COMPILING WITH MAKEFILES

- **It is common for applications with several classes to break the implementation into different files**
  - Easier for a group of programmers to work together
  - Put all of the class headers into "class_name.h" files
  - Put all of the method implementations in "class_name.cpp"
  - Put the main program that call the classes in "main.cpp"
  - #include "class_name.h" at top of files that use the class

- **When we do this, we can compile multiple files at one time**
  - g++ -Wall -o main.exe main.cpp class_name.cpp

COMPILING WITH MAKEFILES

- **We can also recompile each source file separately and then combine their object files**
  - g++ -Wall -c class_name.cpp (produces class_name.o)
  - g++ -Wall -c main.cpp (produces main.o)
  - g++ -o main.exe main.o class_name.o
- **If we only recompile source files that have changed since the last compile we can greatly reduce compile time**
  - The program "make" reads commands from a "makefile" to do this clever form of compilation

COMPILING WITH MAKEFILES

- **Makefiles have "rule blocks" with following syntax:**
  - Comments starting with a # character

Example:

```
# compile class_name.cpp
```

COMPILING WITH MAKEFILES

- **Makefiles have "rule blocks" with following syntax:**
  - Comments starting with a # character
  - Output file name followed by colon character

Example:

compile class_name.cpp

class_name.o :

COMPILING WITH MAKEFILES

- **Makefiles have "rule blocks" with following syntax:**
  - Comments starting with a # character
  - Output file name followed by colon character
  - List of input files needed to create the output file

Example:

```
# compile class_name.cpp
class_name.o : class_name.cpp class_name.h
```

COMPILING WITH MAKEFILES

- **Makefiles have "rule blocks" with following syntax:**
  - Comments starting with a # character
  - Output file name followed by colon character
  - List of input files needed to create the output file
  - Command to compile inputs and create output
    - This line MUST be indented with a TAB character

Example:

```
# compile class_name.cpp
class_name.o : class_name.cpp class_name.h
        g++ -c class_name.cpp
```

COMPILING WITH MAKEFILES

- **The "rule blocks" are executed based on:**
  - Rule position
    - The first rule in the makefile is executed first
    - Other rules are ONLY executed if they are needed by the first rule to create missing intermediate files
  - Time stamps
    - Look at the time stamp on input files
    - Look at the time stamp on output file
    - Execute command if input files are NEWER than output file

MAKEFILE EXAMPLE

```
# simple way to create main.exe
main.exe: class_name.cpp main.cpp
        g++ -Wall -o main.exe class_name.cpp main.cpp
```

- Make will compare the time stamps of main.exe to both source files
- If either source file is newer than main.exe the Make will execute compile rule
- The g++ command will compile both source files to create main.exe

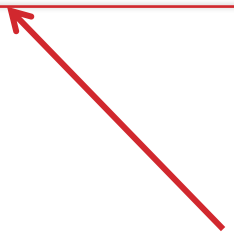
MAKEFILE EXAMPLE

```
# fancy way to create main.exe
main.exe: class_name.o main.o
        g++ -Wall -o main.exe class_name.o main.o

# compile class_name.cpp
class_name.o: class_name.cpp class_name.h
        g++ -Wall -c class_name.cpp

# compile main.cpp
main.o: main.cpp class_name.h
        g++ -Wall -c main.cpp
```

Start first rule: compare the time stamps on input and output files

MAKEFILE EXAMPLE

```
# fancy way to create main.exe
main.exe: class_name.o main.o
        g++ -Wall -o main.exe class_name.o main.o

# compile class_name.cpp
class_name.o: class_name.cpp class_name.h
        g++ -Wall -c class_name.cpp

# compile main.cpp
main.o: main.cpp class_name.h
        g++ -Wall -c main.cpp
```



If class_name.o does not exist (or is out of date) we execute this rule

MAKEFILE EXAMPLE

```
# fancy way to create main.exe
main.exe: class_name.o main.o
        g++ -Wall -o main.exe class_name.o main.o

# compile class_name.cpp
class_name.o: class_name.cpp class_name.h
        g++ -Wall -c class_name.cpp

# compile main.cpp
main.o: main.cpp class_name.h
        g++ -Wall -c main.cpp
```

If main.o does not exist (or is out of date) we execute this rule

MAKEFILE EXAMPLE

```
# fancy way to create main.exe
main.exe: class_name.o main.o
        g++ -Wall -o time.exe class_name.o main.o

# compile class_name.cpp
class_name.o: class_name.cpp class_name.h
        g++ -Wall -c class_name.cpp

# compile main.cpp
main.o: main.cpp class_name.h
        g++ -Wall -c main.cpp
```

Finish first rule: compile input files to create output file

SUMMARY

- **In this section, we described three advanced classes**
  - The Point class stores (x,y) coordinates
  - The Line class is defined using two Point objects
  - The Polygon class is defined using an array of Points
  - The geometric operations in the Line class and the Polygon class call methods in the Point class

- **We also described how makefiles can be used**
  - Break class definition and implementation into multiple files
  - Write makefile rules to compile each class separately
  - Write makefile rule to combine to create output program

CLASSES

PART 6

OPERATOR OVERLOADING

OPERATOR OVERLOADING

- **Operator overloading in C++ allows the programmer to give new meanings to predefined C++ operators**
  - This is done by creating class methods whose "name" is given by one of the predefined operators in C++
  - For example, the C++ string class has defined "+" to perform string concatenation instead of addition

- **Programmers are allowed to "overload" the meaning of almost all C++ operators**
  - arithmetic operations (+, -, *, /, %)
  - comparison operations (<, <=, >, >=, ==, !=)
  - input / output operations (>>, <<)

OPERATOR OVERLOADING

- **The syntax for operator overloading is a little tricky**
  - When we are defining a method we use the keyword "operator" followed by the operator we wish to use
  - For example, we can replace the "add" method with "operator +" and replace "subtract" with "operator -"

- **In order to build classic looking arithmetic expressions, we need to use the following parameter passing rules**
  - Only pass in ONE value parameter of class_type
  - Return a value of class_type after doing operation

COMPLEX CLASS

- **In the "Complex" class we can define (+, -, *, /) methods to add, subtract, multiply and divide complex numbers**

```
Complex x(1,0), y(1,2), z(2,-1);   // y = 1+2i
Complex sum = x + y;               // sum = x.add(y)
Complex product = y * z;           // prod = y.mult(z)
```

- **The implementation of each of these operations must follow the traditional rules for complex arithmetic**

$$(a + bi) + (c + di) = (a+c) + (b+d)i$$
$$(a + bi) - (c + di) = (a-c) + (b-d)i$$
$$(a + bi) * (c + di) = (ac-bd) + (ac+bd)i$$

COMPLEX CLASS

```
class Complex
{
 public:
   Complex(float re = 0.0, float im = 0.0);
   Complex(const Complex & num);
   ~Complex();

   Complex operator +(const Complex num) const;
   Complex operator -(const Complex num) const;
   Complex operator *(const Complex num) const;
   Complex operator /(const Complex num) const;
   …
private:
   float Re;
   float Im;
};
```

Standard constructor, copy constructor and destructor methods

COMPLEX CLASS

```
class Complex
{
 public:
   Complex(float re = 0.0, float im = 0.0);
   Complex(const Complex & num);
   ~Complex();

   Complex operator +(const Complex num) const;
   Complex operator -(const Complex num) const;
   Complex operator *(const Complex num) const;
   Complex operator /(const Complex num) const;
   …
private:
   float Re;
   float Im;
};
```

Operator overloaded addition,subtraction, multiplication, division

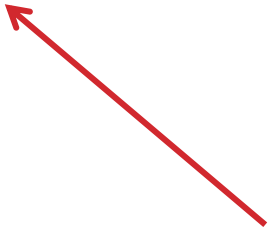
COMPLEX CLASS

```
class Complex
{
 public:
   Complex(float re = 0.0, float im = 0.0);
   Complex(const Complex & num);
   ~Complex();

   Complex operator +(const Complex num) const;
   Complex operator -(const Complex num) const;
   Complex operator *(const Complex num) const;
   Complex operator /(const Complex num) const;
   …
private:
   float Re;
   float Im;
};
```

Private variables for the real and imaginary parts of complex number

COMPLEX CLASS

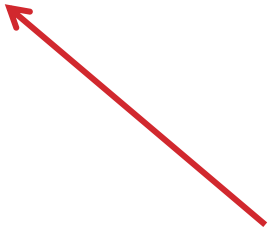
```
Complex Complex::operator + (const Complex num) const
{
   Complex res;
   res.Re = Re + num.Re;
   res.Im = Im + num.Im;
   return res;
}
```



We perform addition
using local variable "res"
and then return this value

COMPLEX CLASS

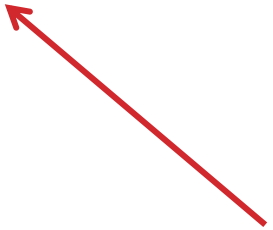
```
Complex Complex::operator - (const Complex num) const
{
   Complex res;
   res.Re = Re - num.Re;
   res.Im = Im - num.Im;
   return res;
}
```



We perform subtraction using local variable "res" and then return this value

COMPLEX CLASS

```
Complex Complex::operator * (const Complex num) const
{
   Complex res;
   res.Re = Re * num.Re - Im * num.Im;
   res.Im = Re * num.Im + Im * num.Re;
   return res;
}
```



We perform multiplication using local variable "res" and then return this value

POLYNOMIAL CLASS

- **In the "Polynomial" class we can define methods to add, subtract, multiply and divide polynomial equations**

  Polynomial a(4,3,2), b(1,2), c(3,4,5); // a = $4+3x+2x^2$

  Polynomial product = a * b;

  Polynomial sum = b + c;

- **The implementation of each of these operations must follow the traditional rules for polynomial arithmetic**

  $(ax^2 + bx + c) + (dx^2 + ex + f) = (a+d)x^2 + (b+e)x + (c+f)$

  $(ax^2 + bx + c) - (dx^2 + ex + f) = (a-d)x^2 + (b-e)x + (c-f)$

  $(bx + c) * (ex + f) = (be)x^2 + (bf+ce)x + cf$

POLYNOMIAL CLASS

```
class Polynomial
{
 public:
   Polynomial (float p0 = 0.0, float p1 = 0.0, float p2 = 0.0, float p3 = 0.0);
   Polynomial (const Polynomial & p);
   ~Polynomial ();

   Polynomial operator +(const Polynomial p) const;
   Polynomial operator -(const Polynomial p) const;
   Polynomial operator *(const Polynomial p) const;
   Polynomial operator /(const Polynomial p) const;
   …
private:
   float coeff[max_degree];
   int degree;
};
```

Standard constructor, copy constructor and destructor methods

POLYNOMIAL CLASS

```
class Polynomial
{
 public:
   Polynomial (float p0 = 0.0, float p1 = 0.0, float p2 = 0.0, float p3 = 0.0);
   Polynomial (const Polynomial & p);
   ~Polynomial ();

   Polynomial operator +(const Polynomial p) const;
   Polynomial operator -(const Polynomial p) const;
   Polynomial operator *(const Polynomial p) const;
   Polynomial operator /(const Polynomial p) const;
   …
private:
   float coeff[max_degree];
   int degree;
};
```

Operator overloaded addition,subtraction, multiplication, division

POLYNOMIAL CLASS

```
class Polynomial
{
 public:
   Polynomial (float p0 = 0.0, float p1 = 0.0, float p2 = 0.0, float p3 = 0.0);
   Polynomial (const Polynomial & p);
   ~Polynomial ();

   Polynomial operator +(const Polynomial p) const;
   Polynomial operator -(const Polynomial p) const;
   Polynomial operator *(const Polynomial p) const;
   Polynomial operator /(const Polynomial p) const;
   …
private:
   float coeff[max_degree];
   int degree;
};
```

Private variables store an array of polynomial coefficients and the degree

POLYNOMIAL CLASS

```
Polynomial::Polynomial(float p0, float p1, float p2, float p3)
{
   if (p3 != 0) degree = 3;
   else if (p2 != 0) degree = 2;
   else if (p1 != 0) degree = 1;
   else degree = 0;

   for (int d = 0; d < max_degree; d++)
       coeff[d] = 0;
   coeff[3] = p3;
   coeff[2] = p2;
   coeff[1] = p1;
   coeff[0] = p0;
}
```

Store the polynomial degree

Store the polynomial coefficients

POLYNOMIAL CLASS

```
Polynomial Polynomial::operator + (const Polynomial p)
{
   Polynomial res;
   if (degree >= p.degree)
      res.degree = degree;
   else
      res.degree = p.degree;

   for (int d = 0; d <= res.degree; d++)
      res.coeff[d] = coeff[d] + p.coeff[d];
   return res;
}
```

Calculate degree of output polynomial

Add the polynomial coefficients

POLYNOMIAL CLASS

```
Polynomial Polynomial::operator - (const Polynomial p)
{
   Polynomial res;
   if (degree >= p.degree)
      res.degree = degree;
   else
      res.degree = p.degree;

   for (int d = 0; d <= res.degree; d++)
      res.coeff[d] = coeff[d] - p.coeff[d];
   return res;
}
```

Calculate degree of output polynomial

Subtract the polynomial coefficients

POLYNOMIAL CLASS

```
Polynomial Polynomial::operator * (const Polynomial p)
{
   Polynomial res;
   res.degree = degree + p.degree;

   for (int d = 0; d <= res.degree; d++)
      res.coeff[d] = 0;

   for (int da = 0; da <= degree; da++)
      for (int db = 0; db <= p.degree; db++)
            res.coeff[da + db] += coeff[da] * p.coeff[db];
   return res;
}
```

Calculate degree of output polynomial

Initialize output polynomial coefficients

Multiply the polynomial coefficients

SUMMARY

- **In this section, we described the syntax for implementing operator overloading in C++**
  - Methods are named "operator +" instead of "add"

- **We illustrated operator overloading with two examples:**
  - The Complex class stores the (real, imaginary) parts of a complex number in private variables
  - The Polynomial class stores the coefficients and degree of a polynomial equation in private variables
  - Similar classes can be used to implement other mathematical objects (Rationals, Matrices, etc.)