ADVANCED OOP

OVERVIEW

OVERVIEW

- **Brief history of object-oriented programming**
  - Simula-67 was first language to have classes and objects
  - Smalltalk was developed by Alan Kay at Xerox Parc in 1972
  - Smalltalk-80 was first released outside Xerox in 1980
  - Objective-C added Smalltalk features to C in 1984
  - Bjarne Stroustrup developed C++ at Bell Labs in 1985
  - Python was first released in 1991 by Guido van Rossum
  - Java designed by James Gosling at Sun in 1995
  - OOP concepts now in dozens of programming languages

OVERVIEW

- **Primary goals of object-oriented programming:**

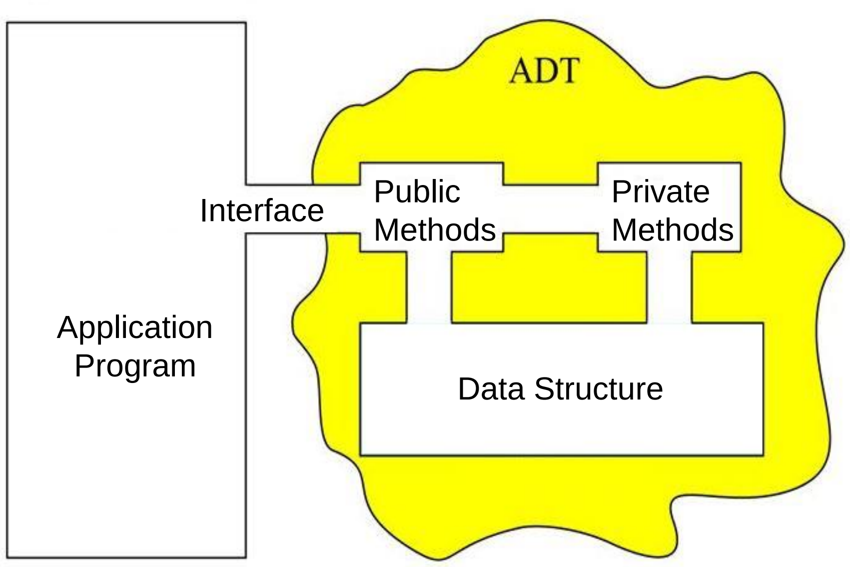
  1. Allow the user to define new data types
     - Specify the data fields in the data type
     - Specify the operations on this data

  2. Implement “information hiding” in the compiler
     - Provide an interface for manipulating data (public)
     - Prevent direct access to data (private)

  3. Simplify program syntax for users of classes
     - Reduced parameter passing, chaining of method calls
     - Operator overloading, templates / generics

OVERVIEW

- **Classes are used to implement abstract data types (ADTs)**
  - A data structure with an interface that provides operations on this data while hiding implementation details



OVERVIEW

- **ADTs often used for mathematical applications**
  - Choose data types specific to application
  - Choose operations specific to application
  - Examples: complex numbers, polynomials, matrices
- **ADTs also used to implement “classic” data structures**
  - Linked Lists
  - Stacks
  - Queues
  - Binary Trees
  - Hash Tables
  - Heaps

OVERVIEW

- **Every programming language tries to distinguish itself from other programming languages by changing the syntax of commands and/or adding unique features**
  - C++ added classes/objects to the language C but left the rest of the language largely unchanged
  - C++ introduced the idea of operator overloading that allows the programmer to redefine the meaning of traditional operators (+, -, *, /, etc.)
  - C++ also introduced the idea of templates where the user can specify the data type to be used in a function/class in the main program without having to recompile the code

ADVANCED OOP

REVIEW OF CLASSES

REVIEW OF CLASSES

- **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

REVIEW OF CLASSES

- **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

REVIEW OF CLASSES

- **To define a class (in class_name.h file)**
  - List the data fields inside the class
  - List the functions/methods that operate on this data
- **To implement a class (in class_name.cpp file)**
  - Implement constructor functions to initialize data fields
  - Implement methods to perform data operations
- **To use a class (after including class_name.h file)**
  - Declare objects of the class
  - Call methods on these objects

ADVANCED OOP

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 tilde 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

- **Programming convention**
  - C++ classes are defined in class_name.h file
  - Use #ifndef compiler flag so code is included only once

- **Examples in source code folder:**
  - student.h – stores student record information
  - book.h – stores information on books
  - video.h – stores information on video clips
  - linear.h – stores linear equations of form Ax + By + C = 0

ADVANCED OOP

IMPLEMENTING CLASSES

IMPLEMENTING CLASSES

- **Class 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

- **Start by creating "skeleton methods"**
  - Copy and paste the method headers from class definition
  - Remove the semicolon at the end of each method 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

- **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
  - Start with getters and setters and the print method
  - Add complex methods last after the others are working
  - This is a classic "incremental development" technique
  - We always have a compiling / running program !!!

IMPLEMENTING CLASSES

- **Programming convention**
  - C++ classes are implemented in class_name.cpp file
  - We must include class_name.h file

- **Examples in source code folder:**
  - student.h – stores student record information
  - book.h – stores information on books
  - video.h – stores information on video clips
  - linear.h – stores linear equations of form Ax + By + C = 0

ADVANCED OOP

USING CLASSES

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 (parameters, return type)**

- **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 been using several built-in classes for some time**

- **The string class**

```
#include <string>
string name = "John";
int len = name.length();
name.append("Smith");
name.insert(4, " ");
cout << name << endl;
```

USING CLASSES

- **We have been using several built-in classes for some time**

- **The ifstream class**

```
#include <fstream>
ifstream din;
din.open(“input.txt”);
if (din.fail()) return;
int number;
while (!din.eof())
{ din >> number; cout << number << “ ”; }
din.close();
```

USING CLASSES

- **Programming convention**
  - Use other classes in main.cpp file
  - We must include class_name.h file

- **Examples in source code folder:**
  - student.h – stores student record information
  - book.h – stores information on books
  - video.h – stores information on video clips
  - linear.h – stores linear equations of form Ax + By + C = 0

ADVANCED OOP

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**
  - Pass in one value parameter of class_type
  - Return a value of class_type after doing operation

ADVANCED OOP

COMPLEX NUMBERS

COMPLEX NUMBERS

- **In a traditional implementation of a complex numbers class, we can define mathematical operations as follows**

```
Complex Add(const Complex num) const;
Complex Subtract(const Complex num) const;
Complex Multiply(const Complex num) const;
Complex Divide(const Complex num) const;
```

- **We can use these methods to perform calculations**

```
Complex x(1,1), y(2,0), z(0,3);
Complex sum = x.Add(y);
Complex product = y.Multiply(z);
```

COMPLEX NUMBERS

- **To convert this to operator overloading, we replace the method names with “operator X” as shown below**

```
Complex operator + (const Complex num) const;
Complex operator - (const Complex num) const;
Complex operator * (const Complex num) const;
Complex operator / (const Complex num) const;
```

- **We can use these methods to perform calculations**

```
Complex x(1,1), y(2,0), z(0,3);
Complex sum = x + y;
Complex product = y * z;
```

COMPLEX NUMBERS

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

   Complex Add(const Complex num) const;
   Complex Subtract(const Complex num) const;
   Complex Multiply(const Complex num) const;
   Complex Divide(const Complex num) const;
   …
private:
   float Re;
   float Im;
};
```

Traditional methods for addition, subtraction, multiplication, division

COMPLEX NUMBERS

```
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 NUMBERS

```
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 NUMBERS

```
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 NUMBERS

```
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 NUMBERS

```
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 NUMBERS

```
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

COMPLEX NUMBERS

```
Complex Complex::operator / (const Complex num) const
{
   // Calculate magnitude of num
   float magnitude = num.Re * num.Re + num.Im * num.Im;
   if (magnitude <= 0.0)
      magnitude = 1.0;

   // Calculate result
   Complex res;
   res.Re = (Re * num.Re + Im * num.Im) / magnitude;
   res.Im = (Im * num.Re - Re * num.Im) / magnitude;
   return res;
}
```

The formula for complex division is more complex (see Wikipedia for details)

ADVANCED OOP

POLYNOMIALS

POLYNOMIALS

- **In a traditional implementation of a Polynomial class, we can define mathematical operations as follows**

  Polynomial Add(const Polynomial num) const;

  Polynomial Subtract(const Polynomial num) const;

  Polynomial Multiply(const Polynomial num) const;

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

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

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

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

POLYNOMIALS

- **To use this Polynomial class, we declare Polynomial objects and call these methods**

```
Polynomial a(4,3,2);        // a(x) = 4+3x+2x^2
Polynomial b(1,2);          // b(x) = 1 + 2x
Polynomial c(3,4,5);        // c(x) = 3+4x+5x^2
Polynomial product = a.Multiply(b);
Polynomial sum = b.Add(c);
Polynomial difference = b.Subtract(c.Add(a));
```

POLYNOMIALS

- **We can use operator overloading in the Polynomial methods by replacing the method names with operators**

  Polynomial operator + (const Polynomial num) const;

  Polynomial operator - (const Polynomial num) const;

  Polynomial operator * (const Polynomial num) const;

- **Now we can use these operators in our Polynomial calculations instead of using method names**

  Polynomial product = a * b;

  Polynomial sum = b + c;

  Polynomial difference = b - (c + a)

POLYNOMIALS

```
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

POLYNOMIALS

```
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

POLYNOMIALS

```
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

POLYNOMIALS

```
Polynomial::Polynomial(float p0, float p1, float p2, float p3)
{
   if (p3 != 0) degree = 3;
   else if (p2 != 0) degree = 2;        <-- Store the polynomial degree
   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;                        <-- Store the polynomial coefficients
   coeff[1] = p1;
   coeff[0] = p0;
}
```

POLYNOMIALS

```
Polynomial Polynomial::operator + (const Polynomial p)
{
  Polynomial res;
  if (degree >= p.degree)
    res.degree = degree;          <-- Calculate degree of output polynomial
  else
    res.degree = p.degree;

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

POLYNOMIALS

```
Polynomial Polynomial::operator - (const Polynomial p)
{
  Polynomial res;
  if (degree >= p.degree)
    res.degree = degree;          ← Calculate degree of output polynomial
  else
    res.degree = p.degree;

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

POLYNOMIALS

```
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

ADVANCED OOP

TEMPLATES

TEMPLATES

- **What is a template? (old definition)**
  - A shaped piece of metal, wood, card, plastic, or other material used as a pattern for processes such as painting, cutting out, shaping, or drilling.

TEMPLATES

- **What is a template? (newer definition)**
  - A preset format for a document or file, used so that the format does not have to be recreated each time it is used. Example: presentation templates, resume templates, etc.

TEMPLATES

- **What is a template? (C++ definition)**
  - Templates are the foundation of generic programming, which involves writing code in a way that is independent of any data type.
  - We can create functions or classes with a "placeholder" data type and later call this code with different data types in our program.
- **Examples:**
  - Create templated sorting function, and then call the function to sort arrays of integers, floats, or strings.
  - Create an image processing class, and use this class to store and process integer, float, or complex images.

TEMPLATES

- **Function templates:**

```
template <class myType>
myType Max(myType p1, myType p2)
{
   if (p1 > p2)
      return p1;
   else
      return p2;
}

 int max1 = Max<int>(42, 17);
 float max2 = Max<float>(4.2, 1.7);
```

We use myType as the placeholder for the data type

Calling two versions of the functions that expect different parameter types

TEMPLATES

- **Class templates:**

```
template <class myType>
class MyClass
{
Public:
   myType method1();
   void method2(myType param);

Private:
   myType variable;
};
```

We use myType as the placeholder for the data type used in class methods and private variables

TEMPLATES

- **Class templates:**

```
template <class myType>
myType MyClass::method1()
{
…
}

template <class myType>
void MyClass::method2(myType param)
{
…
}
```

We use myType placeholder when implementing class methods too

TEMPLATES

- **Class templates:**

```
MyClass <int> object1;
int result = object1.method1();
object1.method2(42);

MyClass <float> object2;
float answer = object2.method1()
object2.method2(3.14159);
```

We specify the desired data types when we create objects

TEMPLATES

- **Compiling class templates:**
  - We must specify the placeholder data type in the class definition in the class.h file and for each method implementation in the class.cpp file
  - In most compilers, template classes must be defined and implemented at the same time
  - This can be accomplished by #including the class.cpp file at the bottom of the class.h file
  - Compiling template classes within an IDE can be tricky and may require renaming the class.cpp file

ADVANCED OOP

NUMBERS CLASS

NUMBERS CLASS

- **We will illustrate class templates with the Numbers class**
  - The private variables for this class include a fixed size array with a placeholder data type
  - Public methods calculate the maximum, minimum, and median value in this array, and return a value of the placeholder data type
  - When using the Numbers class in a program, we can create Numbers objects that contain an array of integers, or an array of floats depending on the application needs

NUMBERS CLASS

```
template <class DataType>
class Numbers
{
 public:
   Numbers();
   Numbers(string filename);
   ~Numbers();

   DataType findMin();
   DataType findMax();
   DataType findMean();
```

We are using DataType as the placeholder for the data type

The return type for these methods is also the placeholder DataType

NUMBERS CLASS

```
…

 private:
   static const int SIZE = 100;
   DataType Data[SIZE];
   int Count;
};
```

The private array is also declared using the DataType placeholder

NUMBERS CLASS

```
template <class DataType>
Numbers<DataType>::Numbers()
{
   // Initialize variables
   Count = 0;
   for (int i = 0; i < SIZE; i++)
      Data[i] = 0;
}
```

We initialize the private Data array with zeros

NUMBERS CLASS

```
template <class DataType>
Numbers<DataType>::Numbers(string filename)
{
  // Open input file
  ifstream din;
  din.open(filename.c_str());
  if (din.fail())
    return;
```

NUMBERS CLASS

```
…
  // Read integers into Data array
  Count = 0;
  double num;
  din >> num;
  while (!din.eof() && Count < SIZE)
  {
    Data[Count++] = (DataType)num;
    din >> num;
  }
  din.close();
}
```

We cast the input value into the correct type for Data array

NUMBERS CLASS

```
template <class DataType>
DataType Numbers<DataType>::findMin()
{
   // Search array for min
   DataType min = Data[0];
   for (int index = 0; index < Count; index++)
      if (min > Data[index])
            min = Data[index];
   return min;
}
```

We search for minimum value in private Data array

NUMBERS CLASS

```
int main()
{
   cout << "processing integer.txt" << endl;
   Numbers <int> num("integer.txt");
   num.print();
   cout << "min = " << num.findMin() << endl;
   cout << "max = " << num.findMax() << endl;
   cout << "mean = " << num.findMean() << endl;
```

Constructor function reads input file into Data array

NUMBERS CLASS

```
…

  cout << "processing float.txt" << endl;
  Numbers <float> num2("float.txt");
  num2.print();
  cout << "min = " << num2.findMin() << endl;
  cout << "max = " << num2.findMax() << endl;
  cout << "mean = " << num2.findMean() << endl;
}
```

Constructor function reads input file into Data array

ADVANCED OOP

VECTOR CLASS

VECTOR CLASS

- **The vector class is a widely used example of C++ templates**
  - Internally the vector class stores data in a dynamic array
  - This array can grow or shrink as data is inserted or deleted
  - Because it uses a dynamic array, random access is very fast
  - Because it uses templates, vectors can store any data type

- **In this section we will**
  - Describe how dynamic arrays can be implemented
  - Create a Vector class with a dynamic array of integers
  - Convert this Vector class into a templated class

VECTOR CLASS

- **Creating dynamic arrays**
  - We use the new command to allocate memory
  - The delete command is used to free memory

- **Start with an array containing 10 integers**

```
// Allocate array
int * data = new int[10];

// Store data in array
for (int i=0; i<10; i++)
   data[i] = 42;
```

VECTOR CLASS

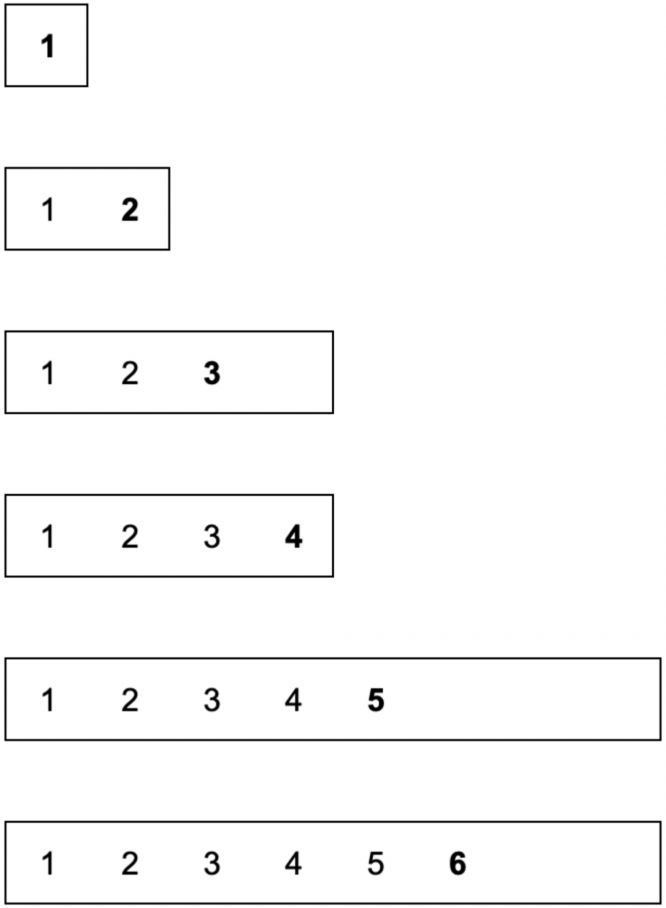
- **Extend array to contain 5 more integers**

```
// Allocate new array
int * copy = new int[15];

// Copy data into new array
for (int i=0; i<10; i++)
   copy[i] = data[i];
for (int i=10; i<15; i++)
   copy[i] = 101;

// Adjust array pointers
delete [] data
data = copy;
```

← Copying data into the new array can be very time consuming for large arrays

VECTOR CLASS

- **To minimize the number of new, copy, delete operations the C++ vector class always doubles the size of the array when additional space is needed**
  - Here we are appending values 1,2,3,4,5,6 to the vector
  - The array length goes from 1,2,4,8 as data is added
  - This guarantees that array is always at least half full



VECTOR CLASS

```
class Vector
{
 public:
   // Constructors
   Vector();
   Vector(const int size);
   Vector(const int size, const int & val);
   Vector(const Vector & copy);
   ~Vector();
```

Our Vector class has four constructors for initializing the dynamic array of integers

VECTOR CLASS

```
// Capacity methods
int size();
void resize(const int size);
void resize(const int size, const int & val);
int capacity();
bool empty();
void reserve(const int size);
…
```

We have several methods to adjust the size and capacity of the vector object

VECTOR CLASS

```
// Element access methods
int get(const int pos);
void set(const int pos, const int & val);
int front();
int back();
```

Methods that allow the user to access data in the vector (with limited error checking)

```
// Modifier methods
void push_back(const int & val);
void pop_back();
void insert(const int pos, const int & val);
void erase(const int pos);
void clear();
```

Methods that allow the user modify the contents of the vector (add/remove data)

VECTOR CLASS

```
 private:
   int Size;
   int Capacity;
   int * Data;
};
```

Private variables store the dynamic array of integers and the current size and capacity of the vector object

VECTOR CLASS

```
Vector::Vector(const int size, const int & val)
{
   if (size < 0)
      Size = 0;
   else
      Size = size;
   Capacity = Size;
   if (Capacity == 0)
      Data = NULL;
   else
      Data = new int[Capacity];
   for (int i=0; i<Capacity; i++)
      Data[i] = val;
}
```

Error checking on Size

Allocating space for array

Initializing data in array

VECTOR CLASS

```
void Vector::reserve(const int size)
{
   if (size > Capacity)
   {
      Capacity = size;
      int *data = new int[Capacity];
      for (int i=0; i<Size; i++)
         data[i] = Data[i];
      delete [] Data;
      Data = data;
   }
}
```

← Checking if new space is needed

← Allocating space for new array

← Copying data from old array to new array

VECTOR CLASS

```
void Vector::push_back(const int & val)
{
   // Allocate space
   if (Size == 0)
      reserve(1);
   else if (Size == Capacity)          <-- Allocating space for new array
      reserve(2*Capacity);

   // Save value
   Data[Size] = val;                   <-- Storing value at last location of array
   Size++;
}
```

VECTOR CLASS

```
void Vector::erase(const int pos)
{
   // Error checking
   if ((pos < 0) || (pos > Size))
      return;

   // Move data
   for (int i=pos; i<Size; i++)
      Data[i] = Data[i+1];
   Size--;
}
```

← Error checking pos is within range

← Moving data to the left one position

VECTOR CLASS

```
int main()
{
   Vector vect(4, 42);          42 42 42 42

   vect.push_back(31);          42 42 42 42 31  -   -   -

   vect.push_back(20);          42 42 42 42 31 20  -   -

   vect.insert(1, 17)           42 17 42 42 42 31 20  -

   vect.erase(3);               42 17 42 42 31 20  -   -
}
```

VECTOR CLASS

- **Converting vector.h into a templated class**
  - Add “template <class DataType>” before class definition
  - Use “DataType” instead of “int” for all parameters and return types for dynamic data values
  - #include “vector.cpp” at bottom of the vector.h file

- **Converting vector.h into a templated class**
  - Add “template <class DataType>” before class methods
  - Change “Vector::” into “Vector<DataType>::” in class methods
  - Use “DataType” instead of “int” for all parameters, variables and return types for dynamic data values

VECTOR CLASS

```
template <class DataType>
class Vector
{
 public:
   // Constructors
   Vector();
   Vector(const int size);
   Vector(const int size, const DataType & val);
   Vector(const Vector & copy);
   ~Vector();
```

Our Vector class has four constructors for initializing the dynamic array of any type

VECTOR CLASS

```
  // Capacity methods
  int size();
  void resize(const int size);
  void resize(const int size, const DataType & val);
  int capacity();
  bool empty();
  void reserve(const int size);
…
```

Change int to DataType when referring to vector data values

Do not change types for size or position variables or parameters

VECTOR CLASS

```
// Element access methods
 DataType get(const int pos);
void set(const int pos, const DataType & val);
 DataType front();
 DataType back();

// Modifier methods
void push_back(const DataType & val);
void pop_back();
void insert(const int pos, const DataType & val);
void erase(const int pos);
void clear();
```

Replace int with DataType in parameters and return types of methods that refer to data in the vector

VECTOR CLASS

```
 private:
   int Size;
   int Capacity;
    DataType * Data;
};
```

← We make the vector generic (able to store any data type) by using DataType here

VECTOR CLASS

```
template <class DataType>
Vector<DataType>::Vector(const Vector & copy)
{
   Size = copy.Size;
   Capacity = copy.Capacity;
   if (Capacity == 0)
      Data = NULL;
   else
      Data = new DataType[Capacity];
   for (int i=0; i<Size; i++)
      Data[i] = copy.Data[i];
}
```

Changing method header

Changing array data type

VECTOR CLASS

```
template <class DataType>
void Vector<DataType>::reserve(const int size)
{
   if (size > Capacity)
   {
      Capacity = size;
      DataType *data = new DataType[Capacity];
      for (int i=0; i<Size; i++)
         data[i] = Data[i];
      delete [] Data;
      Data = data;
   }
}
```

Changing method header

Changing array data type

VECTOR CLASS

```
template <class DataType>
DataType Vector<DataType>::get(const int pos)
{
   if ((pos >= 0) && (pos < Size))
      return Data[pos];
   else
      return -1;
}
```

We added get/set methods to our Vector class instead of operator overloaded [ ] access

VECTOR CLASS

```
template <class DataType>
void Vector<DataType>::set(const int pos, const DataType & val)
{
   if ((pos >= 0) && (pos < Size))
      return;
   else
      Data[pos] = val;
}
```

We added get/set methods to our Vector class instead of operator overloaded [ ] access

VECTOR CLASS

int main()	
{	
Vector<int> vect(4, 42);	42 42 42 42
vect.push_back(31);	42 42 42 42 31 - - -
vect.push_back(20);	42 42 42 42 31 20 - -
vect.insert(1, 17)	42 17 42 42 42 31 20 -
vect.erase(3);	42 17 42 42 31 20 - -
}	

VECTOR CLASS

```
int main()
{
   Vector<string> vect(2, “hi”);

   vect.push_back(“mom”);

   vect.push_back(“dad”);

   vect.insert(3, “and”)

   vect.erase(1);
}
```

hi hi

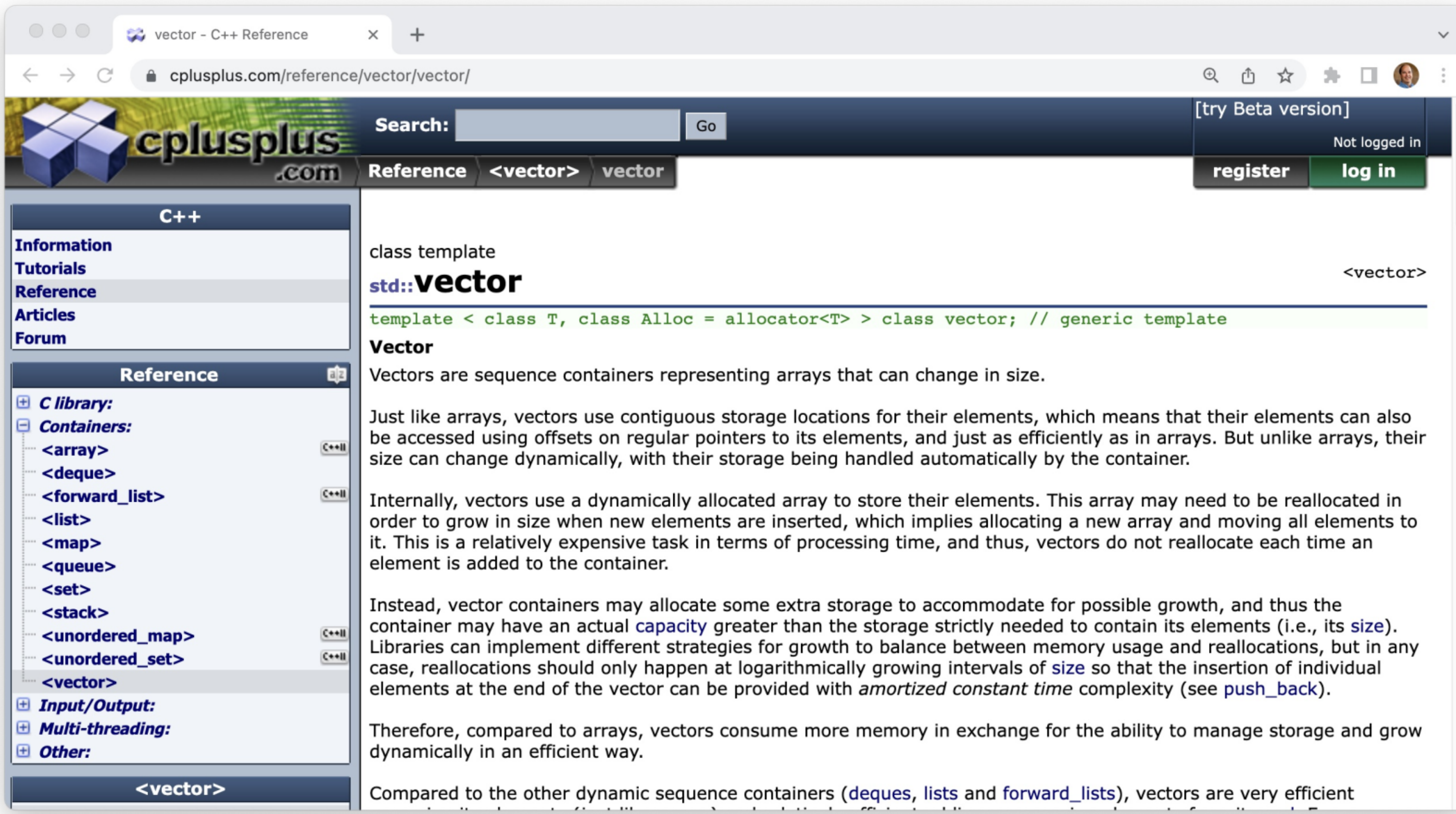
hi hi mom -

hi hi mom dad

hi hi mom and dad - - -

hi mom and dad - - - -

VECTOR CLASS



ADVANCED OOP

SUMMARY

SUMMARY

- **Classes are used to implement abstract data types (ADTs)**
  - A data structure with an interface that provides operations on this data while hiding implementation details
- **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++
- **Templates are the foundation of generic programming writing code in a way that is independent of any data type.**
  - Functions or classes use “placeholder” data type and later call this code with different data types in our program.