CENG 213
Data Structures

Assoc. Prof. Yusuf Sahillioğlu

Department of Computer Engineering
Middle East Technical University
CENG 213

Instructor: Yusuf Sahillioğlu
Office: B107
Email: ys@ceng.metu.edu.tr
Lecture Hours:
• Tue 11:40 - 13:30 (BMB1), Thu 14:40 - 15:30 (BMB3)

Web Page: http://www.ceng.metu.edu.tr/~ys/ceng213-ds
Official: http://saksagan.ceng.metu.edu.tr/courses/ceng213

Thanks to Prof. Nihan Kesim Çičekli for helping with the course lecture notes.
Course Description

Course Objectives: To introduce abstract concepts for data organization and manipulation, to show how these concepts are useful (e.g., fast) in problem solving.

Text Book and References
5. The C++ Tutorial: http://www.learnCPP.com/
Grading //tentative

- Midterm: 30%
- Final Exam: 35%
- 3 Programming Assignments: 15%
- 3 Quizes: 15%
- 6 Lab Sessions: 5%
- All assignments are to be your own work. No group projects or assignments are allowed.
- Exams are closed-book.
Course Outline

- Overview of object-oriented programming with C++ [chapter 1]
- Algorithm analysis [chapter 2]
- Sorting [chapter 7]
- Lists, stacks, queues [chapter 3]
- Trees [chapter 4]
- Priority queues [chapter 6]
- Hashing [chapter 5]
- Graphs [chapter 9]
Motivational Example

• All these structures are there to efficiently store and process data

• Even a simple data structure may be very useful if used in the right context

• Store nxn entries of an N-matrix in an array A
  – Huge profit considering, e.g., n=1 million
  – Column 1, column n, diagonal (remaining)
  – Size of the compact array A?
Motivational Example

• All these structures are there to efficiently store and process data
• Even a simple data structure may be very useful if used in the right context
• Store nxn entries of an N-matrix in an array A
  – Huge profit considering, e.g., n=1 million
  – Column 1, column n, diagonal (remaining)
  – n + n + n-2 = 3n-2
  – Implement set(i,j,v) to update A[] accordingly
Motivational Example

- All these structures are there to efficiently store and process data
- Even a simple data structure may be very useful if used in the right context
- Store nxn entries of an N-matrix in an array A
  - Huge profit considering, e.g., n=1 million
  - Column 1, column n, diagonal (remaining)
  - 3n-2
  - if (j==1) A[i] = v;
  - else if (j==n) A[n+i] = v;
  - else if (i==j) A[2n+i-1] = v;
  - else invalid i,j provided to set();
Programming in C++

- C++
  - Improves on many of C's features
  - Has object-oriented capabilities
    - Increases software quality and reusability
  - Developed by Bjarne Stroustrup at Bell Labs
    - Called "C with classes"
    - C++ (increment operator) - enhanced version of C
  - Superset of C
    - Can use a C++ compiler to compile C programs
    - Gradually evolve the C programs to C++
Object Oriented Programming

➢ The emphasis is on creating a set of tools which can be used cleanly, with a minimum knowledge about implementation in the user’s driver files. The following concepts are relevant to accomplishing clean interface:

1. Data Abstraction
   – Providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details
     • TV: you can turn on and off, change the channel, adjust the volume, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.
     • sort: you can sort an array with this C++ call, BUT you do not know the algorithm.
   – In C++, we use classes to define our own abstract data types (ADT).
Object Oriented Programming

➢ The emphasis is on creating a set of tools which can be used cleanly, with a minimum knowledge about implementation in the user’s driver files. The following concepts are relevant to accomplishing clean interface:

1. Data Abstraction
   – A good way to achieve this is to separate implementation files (.cpp or .hpp) from header files (.h) in C++.

The main benefit of having a separate interface or header file is that it reduces the cognitive load on the reader. If you are trying to understand a large system, you can tackle one implementation file at a time, and you need to read only the interfaces of the other implementations/classes/modules it depends on. This is a major benefit, and languages that do not require separate interface files (such as Java) or cannot even express interfaces in separate files (such as Haskell) often provide tools such as Doxygen or Haddock so that a separate interface, for people to read, is generated from the implementation.
Object Oriented Programming

➢ The emphasis is on creating a set of tools which can be used cleanly, with a minimum knowledge about implementation in the user’s driver files. The following concepts are relevant to accomplishing clean interface:

1. Data Abstraction

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

int main()
{
    int radius = getPosInt("Enter a positive integer for "
                  "the radius of a circle/sphere: ");
    double aCircle = areaOfCircle(radius);
```
Object Oriented Programming

➢ The emphasis is on creating a set of tools which can be used cleanly, with a minimum knowledge about implementation in the user’s driver files. The following concepts are relevant to accomplishing clean interface:

1. Data Abstraction
Object Oriented Programming

➢ The emphasis is on creating a set of tools which can be used cleanly, with a minimum knowledge about implementation in the user’s driver files. The following concepts are relevant to accomplishing clean interface:

1. Data Abstraction

```cpp
#include <iostream>
#include <string>
#include <cmath>
#include "MyMathFunctions.h"
using namespace std;

int getInt(string msg)
{
    int num = 0;
    do
    {
        cout << msg << endl;
        cin >> num;
    }while(num <= 0);
```
1. Data Abstraction

– Example:

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

class Adder{
public:
    // constructor
    Adder(int i = 0) {
        total = i;
    }
    // interface to outside world
    void addNum(int number) {
        total += number;
    }
    // interface to outside world
    int getTotal() {
        return total;
    }
private:
    // hidden data from outside world
    int total;
};

int main() {
    Adder a;
    a.addNum(10);
    a.addNum(20);
    a.addNum(30);
    cout << "Total " << a.getTotal() << endl;
    return 0;
}
```

The public members `addNum` and `getTotal` are the interfaces to the outside world and a user needs to know them to use the class. The private member `total` is something that the user doesn’t need to know about, but is needed for the class to operate properly.
2. Information hiding

- Example:

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

class Adder{
  public:
    // constructor
    Adder(int i = 0) {
      total = i;
    }
    // interface to outside world
    void addNum(int number) {
      total += number;
    }
    // interface to outside world
    int getTotal() {
      return total;
    }
  private:
    // hidden data from outside world
    int total;
};

int main() {
  Adder a;
  a.addNum(10);
  a.addNum(20);
  a.addNum(30);
  cout << "Total " << a.getTotal() << endl;
  return 0;
}
```

Restrict access to data so that it can be manipulated only in authorized ways. Separate class declarations from implementation (e.g. public, private in C++).
Object Oriented Programming

3. **Encapsulation**
   - bundling of data with the methods (or other functions) operating on that data (implementation of the methods).
C++ Techniques

➢ Relevant techniques include:

1. C++ classes, with *private* and *public* members
2. Function and operator name overloading to give "natural" function calls
3. Templates to allow the same code to be used on a variety of different data types
4. A clean built-in I/O interface, which itself involves overloading the input and output operators

➢ Learning these techniques is much of what C++ is all about.
A Basic C++ Program

#include <iostream> // input/output
#include <math.h> // usual C lib header file for math

using namespace std;

int main()
{
    float x;

    cout << "Enter a real number: " << endl;
    cin >> x; // scanf("%f", &x); in C

    cout << "The square root of " << x << " is: "
         << sqrt(x) << endl; // see comments part
}
A Basic C++ Program

```cpp
    cout << "Enter a real number: " << endl;
```

Here, you don't need to understand how `cout` displays the text on the user's screen. You need to only know the public interface and the underlying implementation of `cout` is free to change. //Data Abstraction

In C++, all I/O is done by `classes`. A class is set up to handle input and output `streams`. Output to the screen is handled by the stream with standard name `cout`. This is a variable of class `ostream`. Similarly for `cin`. 
A Basic C++ Program

// second C++ program

#include <iostream>

using namespace std;

int main()
{
    int a=23;
    int b=34;

    cout << "Enter two integers:" << endl;
    cin >> a >> b;
    cout << endl;

    cout << "a + b =" << a+b << endl;
    return 0;
}
A Basic C++ Program

// third C++ program

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

int main()
{
    double a=15.2;
    double b=34.3434343;

    cout << fixed << showpoint;
    cout << setprecision(2); //2 digits after the dot
    cout << setw(6) << a << endl;
    cout << setw(7) << b << endl;

    return 0;
}
// third C++ program

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

int main()
{
    double a=15.2;
    double b=34.343434343;

    cout << fixed << showpoint;
    cout << setprecision(2); //2 digits after the dot
    cout << setw(6) << a << endl; //15.20
    cout << setw(7) << b << endl; //34.34

    return 0;
}
Classes and Objects

• **Class:** a type definition that includes both
  – data properties, and
  – operations permitted on that data

• **Object:** a variable that
  – is declared to be of some Class
  – therefore includes both data and operations for that data

• **Appropriate usage:**
  “A variable is an instance of a type.”
  “An object is an instance of a class.”
Classes and Objects

- **Class**: a type definition that includes both
  - data properties, and
  - operations permitted on that data

- **Object**: a variable that
  - is declared to be of some Class
  - therefore includes both data and operations for that data

- **Appropriate usage**:  
  “A variable is an instance of a type.”  
  “An object is an instance of a class.”
Classes and Objects

• **Class:** a type definition that includes both
  – data properties, and
  – operations permitted on that data

• **Object:** a variable that
  – is declared to be of some Class
  – therefore includes both data and operations for that data

• **Appropriate usage:**
  “A variable is an instance of a type.”
  “An object is an instance of a class.”
Basic Class Syntax

• A class in C++ consists of its **members**.  
  – A member can be either **data** or **functions**.

• The functions are called **member functions** (or **methods**)

• Each instance of a class is an **object**.  
  – Each object contains the data components specified in class.  
  – Methods/functions are used to act on an object.
Class syntax - Example

// A class for simulating an integer memory cell

class IntCell
{
    public:
        IntCell()
        { storedValue = 0; }

        IntCell(int initialValue)
        { storedValue = initialValue; }

    int read()
    { return storedValue; }

    void write(int x)
    { storedValue = x; }

    private:
        int storedValue;
};
Class Members

• **Public** member is visible to all routines and may be accessed by any method in any class.
• **Private** member is not visible to non-class routines and may be accessed only by methods in its class.
• Typically,
  – Data members are declared private
  – Methods are made public
• Restricting access is known as *information hiding.*
Class Members

```cpp
#include <iostream>

using namespace std;

class Line
{
    public:
        double length;
        void setLength( double len );
        double getLength( void );
};

// Member functions definitions
double Line::getLength(void)
{
    return length;
}

void Line::setLength( double len )
{
    length = len;
}

// Main function for the program
int main( )
{
    Line line;

    // set line length
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;

    // set line length without member function
    line.length = 10.0; // OK: because length is public
    cout << "Length of line : " << line.length << endl;
    return 0;
}
```
Constructors

• A constructor is a method that executes when an object of a class is declared and sets the initial state of the new object.

• A constructor
  – has the same name with the class,
  – no return type
  – has zero or more parameters (the constructor without an argument is the default constructor)

• There may be more than one constructor defined for a class.

• If no constructor is explicitly defined, one that initializes the data members using language defaults is automatically generated.
// A class for simulating an integer memory cell

class IntCell
{
    public:
        IntCell( int initialValue = 0 ) :
            storedValue( initialValue ) {}

    int read() const
    {
        return storedValue;
    }

    void write( int x )
    {
        storedValue = x;
    }

    private:
        int storedValue;
};
Object Declaration

• In C++, an object is declared just like a primitive type.

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

int main()
{
    // correct declarations
    IntCell m1;
    IntCell m2 (12);
    IntCell *m3;

    // incorrect declaration
    IntCell m4(); // this is a function declaration, not an object
```
// program continues

m1.write(44);
m2.write(m2.read() + 1);
cout << m1.read() << "   " << m2.read() << endl;
m3 = new IntCell;
cout << "m3 = " << m3->read() << endl;
return 0;
Example: Class Time

class Time {
public:
    Time( int = 0, int = 0, int = 0 ); //default
    //constructor
    void setTime( int, int, int ); //set hr, min, sec
    void printMilitary(); // print am/pm format
    void printStandard(); // print standard format

private:
    int hour;
    int minute;
    int second;
};
Declaring Time Objects

// Note that implementation of class Time not given here.

int main(){
    Time t1, // all arguments defaulted
    t2(2), // min. and sec. defaulted
    t3(21, 34), // second defaulted
    t4(12, 25, 42); // all values specified
    ...
}
Class Interface and Implementation

• In C++, separating the class interface from its implementation is common.
  – The interface remains the same for a long time.
  – The implementations can be modified independently.
  – The writers of other classes and modules have to know the interfaces of classes only.

• The **interface** lists the class and its members (data and function prototypes) and describes what can be done to an object.

• The **implementation** is the C++ code for the member functions.
Separation of Interface and Implementation

• It is a good programming practice for large-scale projects to put the interface and implementation of classes in different files.
  • For small amount of coding it may not matter.

• **Header File**: contains the interface of a class. Usually ends with `.h` (an include file)

• **Source-code file**: contains the implementation of a class. Usually ends with `.cpp` (`.cc `
  or  `.C`)
  • `.cpp` file includes the `.h` file with the **preprocessor** command `#include`.
    » Example: `#include "myclass.h"`
Separation of Interface and Implementation

• A big complicated project will have files that contain other files.
  – There is a danger that an include file (.h file) might be read more than once during the compilation process.
    • It should be read only once to let the compiler learn the definition of the classes.

• To prevent a .h file to be read multiple times, we use preprocessor commands #ifndef and #define in the following way.
Class Interface

```cpp
#ifndef _IntCell_H_
#define _IntCell_H_

class IntCell {
    public:
        IntCell( int initialValue = 0 );
        int read( ) const;
        void write( int x );
    private:
        int storedValue;
};
#endif
```

IntCell class Interface in the file IntCell.h
#include <iostream>
#include "IntCell.h"
using std::cout;

//Construct the IntCell with initialValue
IntCell::IntCell( int initialValue)
    : storedValue( initialValue) {} 

//Return the stored value.
int IntCell::read( ) const
{
    return storedValue;
}

//Store x.
void IntCell::write( int x )
{
    storedValue = x;
}

IntCell class implementation in file IntCell.cpp
A driver program

```cpp
#include <iostream>
#include "IntCell.h"
using std::cout;
using std::endl;

int main()
{
    IntCell m;  // or IntCell m(0);

    m.write (5);
    cout << "Cell content: " << m.read() << endl;

    return 0;
}
```

A program that uses IntCell in file TestIntCell.cpp
Destructors

- Member function of class
- Performs termination housekeeping before the system reclaims the object’s memory
- Complement of the constructor
- Name is tilde (~) followed by the class name
- E.g. \(~\text{IntCell}()\);
- ~ \text{Time}();
- Receives no parameters, returns no value
- One destructor per class
Destructors

- A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
class IntCell{
    public:
        IntCell(int initialValue=0)
        { storedValue = new int (initialValue); }

    ~IntCell()
    { delete storedValue; }

    int read() const
    { return *storedValue; }

    void write( int x ) { *storedValue = x; }

    private:
        int *storedValue;
}
Destructor Example

```cpp
class Line
{
    public:
        void setLength( double len );
        double getLength( void );
        Line();  // This is the constructor declaration
        ~Line();  // This is the destructor declaration
    
    private:
        double length;
};

// Member functions definitions including constructor
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
Line::~Line(void)
{
    cout << "Object is being deleted" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}

// Main function for the program
int main( )
{
    Line line;

    // set line length
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;
    return 0;
}
```

Object is being created
Length of line : 6
Object is being deleted
When are Constructors and Destructors Called

- **Global scope objects**
  - Constructors called before any other function (including main)
  - Destructors called when main terminates (or exit function called)

- **Automatic local objects**
  - Constructors called when objects defined
  - Destructors called when objects leave scope (when the block in which they are defined is exited)

- **static local objects**
  - Constructors called when execution reaches the point where the objects are defined
  - Destructors called when main terminates or the exit function is called
Accessor and Modifier Functions

• A method that examines but does not change the state of its object is an accessor.
  – Accessor function headings end with the word *const*

• A member function that changes the state of an object is a *mutator.*
Another Example: Complex Class

```c++
#ifndef _Complex_H
#define _Complex_H

using namespace std;

class Complex
{
    float re, im;  // by default private
public:
    Complex(float x = 0, float y = 0)
    : re(x), im(y) { }

    Complex operator*(Complex rhs) const;
    float modulus() const;
    void print() const;
};

#endif
```

Complex class Interface in the file Complex.h
Implementation of Complex Class

```cpp
#include <iostream>
#include <cmath>
#include "Complex.h"

Complex Complex::operator*(Complex rhs) const
{
    Complex prod;
    prod.re = (re*rhs.re - im*rhs.im);
    prod.im = (re*rhs.im + im*rhs.re);
    return prod;
}

float Complex::modulus() const
{
    return sqrt(re*re + im*im);
}

void Complex::print() const
{
    std::cout << "(" << re << "\," << im << "\)");
}
```

Complex class implementation in file Complex.cpp
Using the class in a Driver File

```cpp
#include <iostream>
#include "Complex.h"
int main()
{
    Complex c1, c2(1), c3(1,2);
    float x;
    // overloaded * operator!!
    c1 = c2 * c3 * c2;

    // mistake! The compiler will stop here, since the
    // Re and Imag parts are private.
    x = sqrt(c1.re*c1.re + c1.im*c1.im);

    // OK. Now we use an authorized public function
    x = c1.modulus();

    c1.print();
    return 0;
}
```

A program that uses Complex in file TestComplex.cpp
Function Overloading

• Function overloading:
  – Functions with same name and different parameters
  – Overloaded functions performs similar tasks
    • Function to square ints and function to square floats
      
      ```
      int square(int x) { return x * x; }
      float square(float x) { return x * x; }
      ```

  – Program chooses function by signature
    • Signature determined by function name and parameter types
    • Type safe linkage - ensures proper overloaded function called
Function Overloading

- Function overloading:
  - Functions with same name and different parameters
  - Overloaded functions perform similar tasks
    ```
    inline int randoi(int a, int b)
    {
        // generate a random int between a (inclusive) and b > a (exclusive); that is [a,b)
        return (rand() % (b-a)) + a;
    }
    
    inline int randoi(int b)
    {
        // generate a random int between 0 (inclusive) and b > 0 (exclusive); that is [0,b)
        return (rand() % b);
    }
    
    inline float randof(float fMin, float fMax)
    {
        // generate a random real number between fMin and fMax, inclusive [fMin,fMax]
        float f = (float) rand() / RAND_MAX;
        return fMin + f * (fMax - fMin);
    }
    ```
  
- Program chooses function by signature
  - Signature determined by function name and parameter types
// Using overloaded functions
#include <iostream>
using std::cout;
using std::endl;
int square( int x ) { return x * x; }
double square( double y ) { return y * y; }

int main()
{
    cout << "The square of integer 7 is " << square( 7 )
        << "\nThe square of double 7.5 is " << square( 7.5 )
        << endl;

    return 0;
}
Overloaded Operators

• An operator with more than one meaning is said to be *overloaded*.  
  \[ 2 + 3 \quad 3.1 + 3.2 \quad \rightarrow \quad + \]  
is an overloaded operator

• To enable a particular operator to operate correctly on instances of a class, we may define a new meaning for the operator.  
  \[ \rightarrow \text{we may overload it} \]

Ex: `operator*` from Complex and Vector classes
Operator Overloading

• Format
  - Write function definition as normal
  - Function name is keyword `operator` followed by the symbol for the operator being overloaded.
  - `operator+` would be used to overload the addition operator (+)

• No new operators can be created
  - Use only existing operators

• Built-in types
  - You cannot change how two integers are added
Overloaded Operators -- Example

What if we want to multiply a complex number with a scalar? Define another function with the same name but different parameters.

class Complex
{
    ...

    Complex operator*(Complex rhs) const;
    Complex operator*(float k) const;

    ...
};
Implementation of Complex Class

Complex Complex::operator*(Complex rhs) const
{
    Complex prod;
    prod.re = (re*rhs.re - im*rhs.im);
    prod.im = (re*rhs.im + im*rhs.re);
    return prod;
}

Complex Complex::operator*(float k) const
{
    Complex prod;
    prod.re = re * k;
    prod.im = im * k;
    return prod;
}
#include <iostream>
#include "Complex.h"

int main()
{
    Complex c1, c2(1), c3(1,2);

    c1 = c2 * c3 * c2;
    c1.print();

    c1 = c1 * 5; // translated to c1.operator*(5)
    c1.print();

    // How about this?
    c1 = 5 * c1;

    return 0;
}
#include <iostream>
#include "Complex.h"

int main()
{
    Complex c1, c2(1), c3(1,2);

    c1 = c2 * c3 * c2;
    c1.print();

    c1 = c1 * 5; // translated to c1.operator*(5)
    c1.print();

    // How about this?
    c1 = 5 * c1; // CANNOT translate to 5.operator*(c1)
    c1.print();

    return 0;
}
Friend Classes

A class may declare another class as a friend as well. In that case all member functions of the “befriended” class can access the private members of its friend class.

```
class A
{
    ...
};

class B
{
    ...
    friend A;
};
```

“A” can access private members of “B” (but not vice versa!!) B is giving authority to class A to access its private members.
Friend Classes

```cpp
#include<iostream>

using namespace std;

class A
{
    int vala;

    public:
    void set(int tvala)
    {
        vala=tvala;
    }
    void display()
    {
        cout<<"\n The Value of a is:"<vala;
    }

    friend class B;
};
class B
{
    int valb;

    public:
    void set(int tvalb)
    {
        valb=tvalb;
    }
    void display()
    {
        cout<<"\n The Value of b is:"<valb;
    }

    void sum(A ta)
    {
        valb=valb+ta.vala;
    }

    int main()
    {
        A aobj;
        B bobj;
        aobj.set(100);
        bobj.set(200);
        aobj.display();
        bobj.display();
        bobj.sum(aobj);
        bobj.display();
        return 0;
    }
};
```
Friend Classes

class B {
   friend class A; // A is a friend of B
private:
   int i;
};

class A {
public:
   A(B b) {
      b.i = 0; // legal access due to friendship
   }
};

class Node {
private:
   int data;
   int key;
   // ...
   friend int BinaryTree::find(); // Only BinaryTree's find function has access
};
References

• References are a type of C++ variable that act as an alias to another variable.
• A reference variable acts just like the original variable it is referencing.
• References are declared by using an ampersand (&) between the reference type and the variable name.
Example

```cpp
int n = 5, m = 6;
int &rn = n;

n = 6;
rn = 7,
cout << n << rn << m << endl; //776
rn = m ;
cout << n << rn << m << endl; //666
```

You cannot declare a reference without giving a value.

Makes n equal to m (doesn't make rn refer to m)
**const Reference**

- A `const` reference will not let you change the value it references:

  - **Example:**
    ```cpp
    int n = 5;
    const int & rn = n;
    
    rn = 6;  // error!!
    ```

- `const` reference is like a `const` pointer to a `const` object.
References vs Pointers

Everything that is accomplished by references can be accomplished by pointers but the syntax of references is simpler:

Example

```c
int n = 5;
int &rn = n;
int *const p = &n;
*p = 6;
rn = 6;
```

Pointer fun video: [https://youtu.be/i49_SNt4yfk](https://youtu.be/i49_SNt4yfk)
Pointers and \texttt{const}

There are two different ways that pointers and \texttt{const} can be intermixed:

1. Constant pointer
2. Pointer to a constant variable
Constant Pointer

- A `const` pointer must be initialized to a value upon declaration, and its value can not be changed.
- However, because the value being pointed to is still non-`const`, it is possible to change the value being pointed to via dereferencing the pointer:

```c
int *const p = &i;
*p = 6;  // it is O.K.
p = &j;  // NOT O.K.
```
**Pointer to a const variable**

- It is also possible to declare a pointer to a constant variable by using the `const` before the data type:
  ```
  int i;
  const int * p = &i;
  *p = 6;  // it is NOT O.K., because i is not treated as constant when accessed by p.
  ```

- However, it can be changed independently:
  ```
  i = 6;  // It is O.K.
  ```

- It is also possible to declare a `const` pointer to a constant value:
  ```
  const int n = 5;
  const int * const p = &n;
  ```
Parameter Passing

In C, all parameters are passed by value (call by value). But C++ offers three options:

• Call by value
  – Copy of data passed to function
  – Changes to copy do not change original

• Call by reference
  – Uses &
  – Avoids a copy and allows changes to the original

• Call by constant reference
  – Uses const &
  – Avoids a copy and guarantees that actual parameter will not be changed
Example

```cpp
#include <iostream>
using std::cout;
using std::endl;

int squareByValue( int ); // pass by value
void squareByReference( int & ); // pass by reference
int squareByConstReference( const int & ); // const ref.

int main()
{  int x = 2, z = 4, r1, r2;

    r1 = squareByValue(x);
    squareByReference( z );
    r2 = squareByConstReference(x);

    cout << "x = " << x << " z = " << z << endl;
    cout << "r1 = " << r1 << " r2 = " << r2 << endl;
    return 0;
}
```
int squareByValue( int a )
{
    return a *= a;  // caller's argument not modified
}
void squareByReference( int &cRef )
{
    cRef *= cRef;    // caller's argument modified
}
int squareByConstReference (const int& a )
{
    // a *= a;     not allowed (compiler error)
    return a * a;
}
Improving the Complex Class

```cpp
#ifndef _Complex_H
#define _Complex_H

using namespace std;

class Complex
{
    float re, im; // by default private
public:
    Complex(float x = 0, float y = 0)
    : re(x), im(y) { }

    Complex operator*(const Complex & rhs) const;
    float modulus() const;
    void print() const;
};
#endif
```

Old class:
```cpp
class Complex
{
    float re, im; // by default private
public:
    Complex(float x = 0, float y = 0)
    : re(x), im(y) { }

    Complex operator*(Complex rhs) const;
    float modulus() const;
    void print() const;
};
```
#include <iostream>
#include <cmath>
#include "Complex.h"

Complex Complex::operator*(const Complex& rhs) const
{
    Complex prod;
    prod.re = (re*rhs.re - im*rhs.im);
    prod.im = (re*rhs.im + im*rhs.re);
    return prod;
}

float Complex::modulus() const
{
    return sqrt(re*re + im*im);
}

void Complex::print() const
{
    std::cout << "(" << re << "," << im << ")" << std::endl;
}
The uses of keyword `const`

We may encounter `const` in the following cases:

1. Const reference parameter:
   ```cpp
   Complex operator*(const Complex& rhs) const;
   ```
   In this case it means the parameter cannot be modified.

2. Const member function:
   ```cpp
   Complex operator*(const Complex& rhs) const;
   ```
   In this case it means the function cannot modify class members.

3. Const object/variable:
   ```cpp
   const Complex c1(3, 4);
   ```
   In this case it means the object cannot be modified.
Memory Management

In C++ we use `new` and `delete` instead of `malloc` and `free` used in C

- `new` - automatically creates object of proper size, calls constructor, returns pointer of the correct type
- `delete` - destroys object (calls the destructor) and frees space

• Example:

```cpp
int* pi = new int(6);
Complex *pc = new Complex(3, 5);
delete pi;
delete pc;
```
// Allocate an array of complex objects (calls the default constructor for each object).
Complex *ptr1 = new Complex [10];

for (int i = 0; i < 10; ++i)
    ptr[i]->print();

delete[] ptr1; // note the delete[] syntax

// similar for int
int* ptr2 = new int[12];

// free up the dynamically allocated array
delete [] ptr2; //use [] iff [] is used in allocation
Default Arguments Revisited

• In C++ functions can have default arguments
• This is specified in the function declaration (not the definition):

```cpp
int foo(int x = 1, int y = 2, int z = 3);
```

```cpp
foo(); // all parameters use the default value
foo(5); // y and z use the default value
foo(5, 8); // z uses the default value
foo(5, 8, 9); // default values are not used
```
Default Arguments Revisited

• Note that it is impossible to supply a user-defined value for z without also supplying a value for x and y. That is the following does not work:

```c
foo(,,9); // compile error
```

• For this reason the default parameters must be the rightmost ones:

```c
int foo(int x = 1, int y = 2, int z); // WRONG
int foo(int z, int x = 1, int y = 2); // CORRECT
```
Function Overloading Revisited

- Functions with the same name and different parameters
- Overloaded functions should perform similar tasks (otherwise it would be confusing):

  Function to square ints and function to square floats

  ```
  int square(int x) {return x * x;}
  float square(float x) { return x * x;}
  ```

- Compiler chooses based on the actual parameter types:

  ```
  square(4); // calls the integer version
  square(4.0f); // calls the float version
  ```
Function Overloading Revisited

- Functions that only differ by return type cannot be overloaded:

  int square(int x);
  float square(int x); // Compile error
Operator Overloading Revisited

• Remember that we overloaded the * operator for the Complex class.

• Operator overloading allows us to use existing operators for user-defined classes.

• The following operators can be overloaded:

```
+    -    *    /    %    ^    &    |
~    !    ,    =    =    =    &&   ||
++   --   <<=   >>=   ==   !=   &&   ||
+=   -=   /=   %=   ^=   &=   |=   *=
<<=  >>==  [ ]   ( )   ->   -*    new    delete
```

• Note that the precedence, associativity, and arity of the operators cannot be changed!
Copy Constructor

➢ The copy constructor for a class is responsible for creating copies of objects of that class type whenever one is needed. This includes:
1. when the user explicitly requests a copy of an object,
2. when an object is passed to function by value, or
3. when a function returns an object by value.
Example

// The following is a copy constructor
// for Complex class. Since it is same
// as the compiler’s default copy
// constructor for this class, it is
// actually redundant.

Complex::Complex(const Complex & C )
{
    re = C.re;
    im = C.im;
}
Example

class MyString
{
    public:
        MyString(const char* s = " ");
        MyString(const MyString& s);
        ...
    private:
        int length;
        char* str;
};
Example (cont.)

```cpp
MyString::MyString(const MyString& s)  
{
 length = s.length;
 str = new char[length + 1];
 strcpy(str, s.str);
}
```
Calling the copy constructor

• Examples:

  MyObject a; // default constructor call
  MyObject b(a); // copy constructor call
  MyObject bb = a; // identical to bb(a) : copy
                  //constructor call
  MyObject c;    // default constructor call
  c = a;         // assignment operator call
this Pointer

• Each class object has a pointer which automatically points to itself. The pointer is identified by the keyword **this**.

```cpp
public:
  void printThisPointer() const
  {
    cout << this << endl;
  }

void main()
{
  Cow betsy;
  cout << &betsy << endl;
  betsy.printThisPointer();
  Cow georgy;
  cout << &georgy << endl;
  georgy.printThisPointer();
}
```

```cpp
class Cow
{
  int maxHeartbeats;
  Cow(int maxHeartbeats)
  {
    this->maxHeartbeats = maxHeartbeats;
  }
}
```
Example: overloading operator=

// defining an overloaded assignment operator
Complex Complex :: operator=(const Complex & rhs )
{
    // don't assign to yourself!
    if ( this != &rhs ) // note the "address of" rhs
    {
        this -> Re = rhs.Re; // correct but
        // redundant: means Re = rhs.Re
        this -> Imag = rhs.Imag;
    }
    return *this;     // return the calling class object
                      // enables cascading
}
Example

MyString MyString::operator=(const MyString& rhs) {
    if (this != &rhs) {
        delete[] this->str; // donate back useless memory
        this->length = rhs.length;

        // allocate new memory
        this->str = new char[this->length + 1];

        strcpy(this->str, rhs.str); // copy characters
    }
    return *this;    // return self-reference
}
Copy constructor and assignment operator

• Note that the copy constructor is called when a new object is being created

• The assignment operator is called when an existing object is assigned to a new object

class MyObject {
public:
    MyObject();       // Default constructor
    MyObject(const MyObject &a);  // Copy constructor
    MyObject& operator=(const MyObject& a) // Assignment op.
};

MyObject a; // constructor called
MyObject b = a; // copy constructor called
b = a; // assignment operator called
Destructor

- For classes with pointers we also need to define a destructor to avoid memory leaks

```cpp
class MyString {
    public:
    MyString(const char* s = "") ;
    MyString(const MyString& s);

    ~MyString(); // destructor
    MyString& operator=(const MyString& s);

    ...

    private:
    int length;
    char* str;
};
```
Destructor

- For classes with pointers we also need to define a destructor to avoid memory leaks

```cpp
MyString::~MyString()
{
    delete[] str;
}
```
static Class Members

• Shared by all objects of a class
  – Normally, each object gets its own copy of each variable

• Efficient when a single copy of data is enough
  – Only the static variable has to be updated

• May seem like global variables, but have class scope
  – Only accessible to objects of same class

• Exist even if no instances (objects) of the class exist

• Can be variables or functions
  • public, private, or protected
Example

In the interface file:

private:
    static int count;
...

public:
    static int getCount();
...

int Complex::count = 0;

int Complex::getCount()
{
    return count;
}

Complex::Complex()
{
    Re = 0;
    Imag = 0;
    count ++;
}
Driver Program

cout << Complex :: getCount() << endl;
Complex c1;
cout << c1.getCount();
Templates

• The template allows us to write routines that work for arbitrary types without having to know what these types will be.

• Two types:
  – Function templates
  – Class templates
Function Templates

• A function template is not an actual function; instead it is a design (or pattern) for a function.

• The compiler creates the actual function based on the actual types used in the program.

```cpp
// swap function template.

template < class T>
void swap( T &lhs, T &rhs )
{
    T tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

The swap function template
Using a template

- Instantiation of a template with a particular type, logically creates a new function.
- Only one instantiation is created for each parameter-type combination.

```c
int main()
{
    int x = 5, y = 7;
    double a = 2, b = 4;
    swap(x, y); // instantiates an int version of swap
    swap(x, y); // uses the same instantiation
    swap(a, b); // instantiates a double version of swap

    cout << x << " " << y << endl;
    cout << a << " " << b << endl;

    // swap(x, b); // Illegal: no match
    return 0;
}
```
Class templates

- Class templates are used to define generic classes:
  - You can parameterize a class definition with a type, to allow you to write generic type-independent code, e.g., Stack of integers, floats, characters, pointers, or some arbitrary data structure:

```cpp
template <class T>
class Stack {
  public:
    Stack(int sz);  // Constructor: initialize variables, allocate space.
    ~Stack();      // Destructor: deallocate space allocated above.
    void Push(T value); // Push an integer, checking for overflow.
    bool Full();     // Returns TRUE if the stack is full, FALSE otherwise.
  private:
    int size;       // The maximum capacity of the stack.
    int top;        // Index of the lowest unused position.
    T *stack;       // A pointer to an array that holds the contents.
};
```
Class templates

• Class templates are used to define generic classes:
  – You can parameterize a class definition with a type, to allow you to write generic type-independent code, e.g., Stack of integers, floats, characters, pointers, or some arbitrary data structure:

```cpp
// template version of Stack::Stack
template <class T>
Stack<T>::Stack(int sz) {
    size = sz;
    top = 0;
    stack = new T[size];  // Let's get an array of type T
}

// template version of Stack::Push
template <class T>
void Stack<T>::Push(T value) {
    ASSERT(!Full());
    stack[top++] = value;
}
```
Class templates

- Class templates are used to define generic classes:
  - You can parameterize a class definition with a type, to allow you to write generic type-independent code, e.g., Stack of integers, floats, characters, pointers, or some arbitrary data structure:

```c
void test() {
    Stack<int> s1(17);
    Stack<char> *s2 = new Stack<char>(23);

    s1.Push(5);
    s2->Push('z');
    delete s2;
}
```
Class templates

- Class templates are used to define generic classes:
  - You can parameterize a class definition with a type, to allow you to write generic type-independent code, e.g., Stack of integers, floats, characters, pointers, or some arbitrary data structure:

```cpp
template <class T>
class TemplateTest
{
    // this class can use T as a generic type
    public:
    void f(T a);
    T g();
    ...
    private:
    T x, y, z;
    ...
};
```
Implementation

• Each member function must be declared as a template.

// Typical member implementation.

```cpp
template <class T>
void TemplateTest<T>::f(T a)
{
  // Member body
}
```
Form:

\[ \text{class-name} \ <\text{type}> \ \text{an-object}; \]

Interpretation:

– Type may be any defined data type. **Class-name** is the name of a template class. The object **an-object** is created when the arguments specified between < > replace their corresponding parameters in the template class.
Example

// Memory cell interface (MemoryCell.h)

template <class T>
class MemoryCell
{
    public:
        MemoryCell(const T& initVal = T());
        const T& read() const;
        void write(const T& x);

    private:
        T storedValue;
};
Class template implementation

// Implementation of class members

template <class T>
MemoryCell<T>::MemoryCell(const T& initVal) :
    storedValue(initVal){ }

template <class T>
const T& MemoryCell<T>::read() const
{
    return storedValue;
}

template <class T>
void MemoryCell<T>::write(const T& x)
{
    storedValue = x;
}
A simple test routine

```cpp
int main()
{
    MemoryCell<int> m; // instantiate int version
    MemoryCell<float> f; // instantiate float ver.
    MemoryCell<int> m2; // use the previously created class

    m.write(5);
    m2.write(6);
    f.write(3.5);
    cout << "Cell content: " << m.read() << endl;
    cout << "Cell content: " << m2.read() << endl;
    cout << "Cell content: " << f.read() << endl;
    return 0;
}
```
C++ Error Handling

• In C, errors are reported by returning error codes from functions:

```c
int read(const char* filename, char data[]) {
    FILE* fp = fopen(filename, "r");
    if (fp == NULL) {
        return -1; // indicate error
    }
    // read file contents into data
    ...
}
```
C++ Error Handling

- In C++, we have a more advanced mechanism called exceptions
- It uses three keywords: `throw`, `catch`, `try`
- The function that encounters an error throws an exception:

```cpp
int read(const char* filename, char data[]) {
    FILE* fp = fopen(filename, "r");
    if (fp == NULL)
        throw "file open error"; // indicate error

    // read file contents into data
    ...
}
```
C++ Error Handling

- This exception must be caught, otherwise the program will abnormally terminate:

```cpp
int main()
{
    char data[128];
    try {
        read("test.txt", data);
        ... // some other code
    }
    catch(const char* error) {
        // if read throws an exception,
        // program will continue executing from here
        cout << "Error message:" << error << endl;
    }
}
```
C++ Error Handling

- Note that we throw an object or a variable, and we catch an object or a variable. These types should match for the exception to be caught.

- In the previous example we threw a `const char*` and caught a `const char*`, so it was correct.
Another Example

• We can also throw an object of a user defined class:

```c
#include <iostream>

class FileReadError
{
};

int read(const char* filename, char data[])
{
    FILE* fp = fopen(filename, "r");
    if (fp == NULL)
        throw FileReadError(); // indicate error

    // read file contents into data
    ...
}
```
C++ Error Handling

• Then we must update the catch code as well:

```cpp
int main()
{
    char data[128];
    try {
        read("test.txt", data);
    }
    catch(FileReadError error) {
        // if read throws an exception,
        // we will come here
    }
}
```
C++ Error Handling

• There are many details of exception handling
• In this class, you should only know that the destructors of the local objects will be called when an exception is thrown:

```cpp
class A {
public:
    ~A() { cout << "destructor called" << endl; }
};

int read(const char* filename, char data[]) {
    A a;
    FILE* fp = fopen(filename, "r");
    if (fp == NULL)
        throw "file open error"; // a's destructor will be called
}
```
Example of a try-catch Statement

```cpp
try
{
    // Statements that process personnel data and may throw
    // exceptions of type int, string, and SalaryError
}
catch ( int )
{
    // Statements to handle an int exception
}
catch ( string s )
{
    cout << s << endl; // Prints "Invalid customer age"
    // More statements to handle an age error
}
catch ( SalaryError )
{
    // Statements to handle a salary error
}
```
Standard Template Library

• I/O Facilities: iostream
• Garbage-collected String class
• Containers
  – vector, list, queue, stack, map, set
• Numerical
  – complex
• General algorithms
  – search, sort
Using the vector

- Vector: Dynamically growing, shrinking array of elements
- To use it include library header file: 
  #include <vector>
- Vectors are declared as
  vector<int> a(4); //a vector called a,  
                  //containing four integers
  vector<int> b(4, 3); //a vector of four  
                      // elements, each initialized to 3.
  vector<int> c; // 0 int objects
- The elements of an integer vector behave just like ordinary integer variables
  a[2] = 45;
Manipulating vectors

• The size() member function returns the number of elements in the vector. 
  \texttt{a.size()} returns a value of 4.

• The = operator can be used to assign one vector to another.

• e.g. \texttt{v1 = v2}, so long as they are vectors of the same type.

• The push\_back\() member function allows you to add elements to the end of a vector.
push_back() and pop_back()

```cpp
vector<int> v;
 v.push_back(3);
 v.push_back(2);
// v[0] is 3, v[1] is 2, v.size() is 2
 v.pop_back();
 int t = v[v.size()-1]; //t=3
 v.pop_back();
```