Introduction to OOP.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand. (Martin Fowler)

Presented by:

Contents:

- History;
- C refrain;
- The OOP basis;
- Getting started.
The C and C++ development

- C – was developed from BCPL and B languages. The languages hasn’t different types definition, so each variable (char, short or long) is stored as one cell (word) in physical memory. However, the C language is a program language with different build-in types.

- C++ - C based program language which was developed in Bell laboratory in the beginning 80’s. The C++ is support OOP (Object Oriented Programming).

Why C++?

\[ S = a \times b \]

Where:
- \( a \) and \( b \) are variables.
- \( S \) is their product.

Diagram of a circuit with labeled components.
C vs. C++

- Same syntax, operators, expressions, data types, structures, arrays, cycles, functions and pointers;
- C++ has some new keywords:
  - class, delete, friend, inline, new, protected, public and et.
- C++ supports object oriented programming

Inline function

- In the C and C++ programming languages, an inline function is one qualified with the keyword inline; this serves two purposes. Firstly, it serves as a compiler directive that suggests (but does not require) that the compiler substitute the body of the function inline by performing inline expansion, i.e. by inserting the function code at the address of each function call, thereby saving the overhead of a function call. In this respect it is analogous to the register storage class specifier, which similarly provides an optimization hint. The second purpose of inline is to change linkage behavior; the details of this are complicated. This is necessary due to the C/C++ separate compilation + linkage model, specifically because the definition (body) of the function must be duplicated in all translation units where it is used, to allow inlining during compiling, which, if the function has external linkage, causes a collision during linking (it violates uniqueness of external symbols; in C++, the One Definition Rule). C and C++ (and dialects such as GNU C and Visual C++) resolve this in different ways.
Inline function

An inline function can be written in C++ like this:

```cpp
inline void swap(int & m, int & n)
{
    int temp = m;
    m = n;
    n = temp;
}
```

Then, a statement such as the following:

```cpp
swap(x, y);
```

will be translated into

```cpp
int temp = x;
x = y;
y = temp;
```

Introduction to OOP

We perceive the world around us as collection of objects, for example: a bird, a dog, a flower, a car, a house, a man and so on.

Each object has properties. For example: size, shape, color, weight and so on.

Each object has some operations (methods, functions) that characterize it. For example, an object, a person, may to move, talk, eat, etc.. object car can move, twitter, light the way and so on.

We learn about objects while viewing the properties and actions they can perform.
Introduction to OOP

- **Encapsulation (Information hiding)**: 모든 멤버로 구성된 객체가 외부로부터 접근이 제한되어야 한다. 이는 안전성을 높이고, 오류를 최소화하는 데 도움이 된다.

- **Inheritance**: 하나의 클래스가 다른 클래스의 속성을 상속받는 것을 말한다. 이는 코드 재사용을 가능하게 해준다.

- **Polymorphism**: 동일한 이름의 함수가 둘 이상의 객체에서 다른 행동을 할 수 있다. 이는 상속을 통해 클래스의 구현을 다양하게 할 수 있게 해준다.
Example 1: The class definition and the advantages of Encapsulation

- Class (class) is a template for objects. It can include attributes and functions of the object.

- Properties and functions are accessible throughout the class:
  - Public: accessible from within and outside the class.
  - Private: accessible only from within the class, but only to objects defined as friends.
  - Protected: accessible from within the class and its friends, and also from derived classes and their friends.

- Constructor is the class’s function that is called automatically when creating the object. Typically use this function to initialize the instance variables. Constructor function has the same name as the name of the class.

- Destructor is the class’s function that is called automatically when the object should be destroyed (Destructor occurs, for example, at the end of the program). Destructor function has the same name as the name of the class, but has a ~ character before the name.
Input/Output

```cpp
using namespace std;

int n, fact;
 cin >> n;
 fact = factorial(n);
 cout << n << "! = " << fact << endl;
```

**Notes:**
- cin & >> and cout << Both operators can be applied sequentially, and are left associative;
- One can send any type of basic data type to cout or obtain it from cin;
- It is possible to differentiate between normal program output, which is sent to cout, and error output, which is sent to cerr (similar to stdout and stderr in C);
- The endl is a global manipulator, and means end of line (endl = "\n")

Compound data types

It is possible to expand the data type repository by defining *structures*. A structure is used to pack together several fields that are conceptually related:

```cpp
struct Date
{
    int iDay;
    int iMonth;
    int iYear;
} DataStr;
```

```cpp
// Structure objects
definition
struct Date date;
    date.iDay = 31;
    date.iMonth = 12;
    date.iYear = 1999;
// Pointer to structure
struct Data *dateP = &date;
    (*dateP).iDay = 21;
// or easier
dateP->iDay = 21;
```
Structures vs. Classes

Working with complex numbers in C:

```c
struct _Complex
{
    float re; // The real part.
    float im; // The imaginary part.
};
typedef struct _Complex Complex;

#include <math.h>
/* Calculate the amplitude and argument of Complex number */
double magnitude (Complex z)
{
    return(sqrt(z.re*z.re+z.im*z.im));
}
double argument (Complex z)
{
    return(atan2(z.im, z.re));
}
```

```c
struct _Complex complex;
Complex.re = 3;
Complex.im = 4;
M = magnitude(complex);
Complex.re = 5;
```

Working with complex numbers in C++:

```cpp
class Complex
{
private:
    float Re; // The real part.
    float Im; // The imaginary part.
public:
    // Get the real and imaginary part
    float getReal() {return (Re);}
    float getImag() {return (Im);}
    // Set the complex number x + iy
    void set(float x, float y)
    {
        Re = x;
        Im = y;
    }
};

Complex z;
z.set(3, 4);
cout << z.getReal() << "i" << z.getImag() << endl;
z.Re = 5; // ERROR!!! in a private data
```

// Time class.
#include <iostream>
#include <iomanip>
using namespace std;

// Time class definition
class Time
{
public:
    Time(); // constructor
    void setTime( int, int, int ); // set hour, minute, second
    void printTime(); // print universal-time format

private:
    int hour; // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second; // 0 - 59
}; // end class Time
// Time constructor initializes each data member to zero and
// ensures all Time objects start in a consistent state
Time::Time()
{
    hour = minute = second = 0;
} // end Time constructor

// set new Time value using universal time, perform validity
// checks on the data values and set invalid values to zero
void Time::setTime( int h, int m, int s )
{
    hour = ( h >= 0 && h < 24 ) ? h : 0;
    minute = ( m >= 0 && m < 60 ) ? m : 0;
    second = ( s >= 0 && s < 60 ) ? s : 0;
} // end function setTime

// print Time
void Time::printTime()
{
    cout << setfill( '0' ) << setw( 2 ) << hour << ":" 
        << setw( 2 ) << minute << ":" 
        << setw( 2 ) << second;
} // end function printTime

int main()
{
    Time t; // instantiate object t of class Time
    // output Time object t's initial value
    cout << "The initial time is \n"
        << t.printTime(); // 00:00:00
    t.setTime( 13, 27, 6 ); // change time
    // output Time object t's new values
    cout << "The time after setTime is \n"
        << t.printTime(); // 13:27:06
    cout << endl;
    return 0;
} // end main

The initial time is 00:00:00
The time after setTime is 13:27:06
Function Overloading

- C++ supports writing more than one function with the same name but different argument lists. This could include:
  - different data types
  - different number of arguments

- The advantage is that the same apparent function can be called to perform similar but different tasks. The following will show an example of this.

```c
void swap (int *a, int *b) {
int temp;  temp = *a;  *a = *b;  *b = temp;
}

void swap (float *c, float *d) {
float temp;  temp = *c;  *c = *d;  *d = temp;
}

void swap (char *p, char *q) {
char temp;  temp = *p;  *p = *q;  *q = temp;
}

int main ( ) {
int a = 4, b = 6;
float c = 16.7, d = -7.89;
char p = 'M', q = 'n';
swap (&a, &b);
swap (&c, &d);
swap (&p, &q);
}
```
Operator Overloading

- C++ already has a number of types (e.g., int, float, char, etc.) that each have a number of built in operators. For example, a float can be added to another float and stored in yet another float with use of the + and = operators:
  
  ```
  floatC = floatA + floatB;
  ```

- In this statement, floatB is passed to floatA by way of the + operator. The + operator from floatA then generates another float that is passed to floatC via the = operator. That new float is then stored in floatC by some method outlined in the = function.

Operator Overloading (cont.)

- Operator overloading means that the operators:
  - Have multiple definitions that are distinguished by the types of their parameters
  - When the operator is used, the C++ compiler uses the types of the operands to determine which definition should be used.
Operator Overloading (cont.)

class Complex
{
private:
    double re, im;
public:
    Complex()
    {
        Set(0.0, 0.0);
    }
    void Set(double r, double i)
    {
        re = r;
        im = i;
    }
    Complex operator -(Complex z)    // Overloading '-'
    {
        Complex tmp;
        tmp.re = re - z.re;
        tmp.im = im - z.im;
        return tmp;
    }
}

Steps for defining an overloaded operator:

1. Name the operator being overloaded.
2. Specify the (new) types of parameters (operands) the operator is to receive.
3. Specify the type of value returned by the operator.
4. Specify what action the operator is to perform.
Example 2: Inheritance

- Inheritance allows you to build a hierarchical structure of objects.
  - **Base class** - it provides data or functions to other classes.
  - **Derived class** - inherits data or functions from other classes.

**Keywords: base class, derived class, inheritance, polymorphism.**

In C++, it is done by defining the function within the class.

**File point.h:**

```c++
#ifndef POINT_H
#define POINT_H

class Point {
  public:
    Point( int x = 0, int y = 0 ); // default constructor
    void setX( int ); // set x in coordinate pair
    int getX() const; // return x from coordinate pair
    void setY( int ); // set y in coordinate pair
    int getY() const; // return y from coordinate pair
    void print() const; // output Point object
  
  protected:
    int x; // x part of coordinate pair
    int y; // y part of coordinate pair
  };

#endif
```

**Definition:** public, protected.

**Note:** The keyword `const` will ban `getX()`, `getY()` and `print()` in class `Point` from being anything which can attempt to alter any member variables in the object.

**Hint:** In C++, a variable `x` in a class is a **mutable** variable and can be altered.

**Example:**

- **Base class:**
  - `class Point {
    ...}
  ```
- **Derived class:**
  - `class Ellipse : public Point {
    // ...}
  ```

**Example:**

- **Base class:**
  - `class Circle : public Point {
    // ...}
  ```
- **Derived class:**
  - `class Sphere : public Circle {
    // ...}
  ```
File point.cpp:

```cpp
#include <iostream>
#include "point.h" // Point class definition

// default constructor
Point::Point( int xValue, int yValue )
{
    x = xValue;
    y = yValue;
} // end Point constructor

// set x in coordinate pair
void Point::setX( int xValue )
{
    x = xValue; // no need for validation
} // end function setX

// return x from coordinate pair
int Point::getX() const
{
    return x;
} // end function getX

// set y in coordinate pair
void Point::setY( int yValue )
{
    y = yValue; // no need for validation
} // end function setY

// return y from coordinate pair
int Point::getY() const
{
    return y;
} // end function getY

// output Point object
void Point::print() const
{
    cout << '[' << x << ', ' << y << ']
}; // end function print
```

File circle.h:

```cpp
#ifndef CIRCLE_H
#define CIRCLE_H

#include "point.h" // Include point class definition

class Circle : public Point {

public:

    // default constructor
    Circle ( int = 0, int = 0, double r = 0.0 );
    void setRadius( double ); // set radius
    double getRadius() const; // return radius
    double getArea() const; // return area
    void print() const; // output Circle object

private:

    double radius; // Circle's radius

}; // end class Circle
#endif
```
Output after executing

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Area is 56.74
What are the memory segments?

- The **text segment** (often called **code segment**) is where the compiled code of the program itself resides.

- The two other sections from the code segment in the memory are used for **data**.

<table>
<thead>
<tr>
<th>heap</th>
<th>stack</th>
<th>static data</th>
</tr>
</thead>
<tbody>
<tr>
<td>calcSize()</td>
<td>main()</td>
<td></td>
</tr>
<tr>
<td>myobj = new Object()</td>
<td>int y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>code</td>
<td></td>
</tr>
</tbody>
</table>

What is stack?

- The **stack** is the section of memory that is allocated for **automatic variables within functions**.
  - Declaration with no static or external specifier defines an automatic variable.

- Data is stored in stack using the Last In First Out (LIFO) method. This means that storage in the memory is allocated and deallocated at only one end of the memory called the **top of the stack**.

- Stack is a section of memory and its associated registers that is used for temporary storage of information in which the most recently stored item is the first to be retrieved.
What is heap?

- **Heap** is an area of memory used for *dynamic memory allocation*. Blocks of memory are allocated and freed in this case in an arbitrary order. The pattern of allocation and size of blocks is not known until run time.

- Heap is usually being used by a program for many different purposes.

What is heap and stack?

- The **stack** is a place in the computer memory where all the variables that are declared and initialized *before* runtime are stored.

- The **heap** is the section of computer memory where all the variables created or initialized *at* runtime are stored.

- The stack is much faster than the heap but also smaller.
What is heap and stack?

```c
int x; // static stack storage
void main()
{
    int y; // dynamic stack storage
    char str; // dynamic stack storage
    str = malloc(50); // dynamic heap storage
    // if function defined
    // in DLL
    size = calcSize(10); // dynamic heap storage,
}
```

Arrays and Pointers

Static array:

- are allocated on the stack.

```c
int iFibArray[10];
int k;

iFibArray[0] = 1;
iFibArray[1] = 1;

for (k = 2; k < 10; k++)
    iFibArray[k] = iFibArray[k - 1] + iFibArray[k - 2];
```
A pointer is a variable whose size is the same as the size of `int` and can store the address of another variable.

```c
float fSum;       // definition of a variable
float *fSumP;    // definition of a pointer
fSumP = NULL;     // i.e., the pointer is currently not pointing on any variable
fSumP = &fSum;    // fSumP obtains the address of fSum
fSum = *fSumP;    // de-reference a pointer or i.e., obtain the variable in that address
```

Pointers play an important role when passing parameters to functions.

```c
void swap (int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

void main()
{
    int x = 3, y = 4;
    swap(x, y);
}
```

The function on the left don't really change the values of `x` and `y`.

- In C function arguments are passed by value – copy each function argument to stack.
- The solution is to pass the address of the variables to the function.
Dynamic memory in C++

++C - new – (פעולה) ומשחרר הזיכרון (פעולה).
delete – (פעולה).

למרות ש-C++ תומכת בהקצאת זיכרון דינامي also by function malloc and free, נהוג להשתמש באופרטורים new ו-delete; פונקציות malloc ו-free נכללות בגלל שפת C ש(fidשפת C++

יתרונות של new:
– בצורה אוטומטית מאתר מספיק זיכרון בכדי להחזיק אובייקט מסוג ספציפי.
– לא צריך להשתמש באופרטור sizeof, משום שהגודל מותאם אוטומטית.
– אוטומטית מחזיר פוינטר לאותו סוג ספציפי.

Dynamic memory in C++

```cpp
#include <iostream.h>
#include <new.h>

int main ( )
{
   int *ptr;
   // memory allocation for variable with size int
   ptr = new int;
   *ptr = 100;
   cout<<"At " << ptr << " is the value " << *ptr <<"\n";
   delete ptr;
   return 0;
}
```

At 0xabcdef00 is the value 100
Memory allocation for arrays

```cpp
#include <iostream.h>
#include <new.h>

int main ()
{
    int *array, i;
    // memory allocation for array of integers with length 10
    array = new int[10];
    for (i = 1; i < 10; i++)
    {
        *array++ = i;
        cout << array[i] << "\n";
    }
    delete [] array;
    return 0;
}
```

Memory allocation for 2d array

- Is it right? – NOT
- Should be:
  ```cpp
  int **array = new int*[sizeY];
  for(int i = 0; i < sizeY; ++i)
  {
      array[i] = new int[sizeX];
  }
  ```
- And then clean up would be:
  ```cpp
  for(int i = 0; i < sizeY; ++i)
  {
      delete [] array[i];
  }
  delete [] array;
  ```
Pass by Pointer and pass by Reference in C++

// Illustration of pass by pointer
#include <iostream.h>

void square (int *x)
{
    *x = (*x)*(*x);
}

int main ( )
{
    int num = 10;
    square(&num);
    // Value of num is 100
    cout << " Value of num is " << num;
    return 0;
}

// Illustration of pass by reference
#include <iostream.h>

// x becomes a reference parameter
void square (int &x)
{
    // no need to write *x = (*x)*(*x);
    x = x * x;
}

int main ( )
{
    int num = 10;
    square(num);
    // Value of num is 100
    cout << "Value of num is " << num;
    return 0;
}

More about reference variables

- Reference variables are not applicable only to function parameters. You can have them in other places of the program as well. Consider the following program:

```c
#include <iostream.h>

int main( )
{
    int x;
    int &ref = x; // ref is a reference variable
    x=5;
    cout << endl << x << " " << ref;
    x++; // 6 6
    cout << endl << x << " " << ref;
    ref++;
    cout << endl << x << " " << ref;
    cout << endl << &x << endl << &ref;
    return 0;
}
```

- The reference variables provides different names same variable. They both refers to the same memory address. The address of both ‘ref’ and ‘x’ is the same.
Polymorphic behavior of references

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

class A {
    public:
        A() {}
        virtual void print() { cout << "This is class A\n"; }
};
class B: public A {
    public:
        B() {}
        virtual void print() { cout << "This is class B\n"; }
};

int main() {
    A a;
    A& ref2a = a;
    B b;
    A& ref2b = b;
    ref2a.print();
    ref2b.print();
    return 0;
}
```

Virtual Functions

```cpp
#include <iostream.h>
#include <conio.h>

class base {
    public:
        virtual void show() {
            cout << "\n Base class show:"
        }
        void display() {
            cout << "\n Base class display:"
        }
};
class drive: public base {
    public:
        void show() {
            cout << "\n Drive class show:
        }
        void display() {
            cout << "\n Drive class display:
        }
};
```
Virtual Functions

```cpp
void main()
{
    clrscr();
    base obj1;
    base *p;
    cout <<"\n\t P points to base:\n";
    p = &obj1;
    p -> display();
    p -> show();
    cout <<"\n\n\t P points to drive:\n";
    drive obj2;
    p = &obj2;
    p -> display();
    p -> show();
    getch();
}
```

P points to Base
Base class display
Base class show
P points to Drive
Base class Display
Drive class Show

Virtual Functions

- Step 1: Start the program.
- Step 2: Declare the base class base.
- Step 3: Declare and define the virtual function show().
- Step 4: Declare and define the function display().
- Step 5: Create the derived class from the base class.
- Step 6: Declare and define the functions display() and show().
- Step 7: Create the base class object and pointer variable.
- Step 8: Call the functions display() and show() using the base class object and pointer.
- Step 9: Create the derived class object and call the functions display() and show() using the derived class object and pointer.
- Step 10: Stop the program.
Example 3: Polymorphism

- Use the polymorphism feature to extend Base Class functionality in different ways. A class can have different methods when used in different contexts.

- In C++, this is done using virtual functions and operator overloading (which define the polymorphic behavior of the object).

**Example 3:**

```
#include <iostream>
#include <string>

class Pet {
    public:
        // Constructors, Destructors
        Pet() {}  
        virtual ~Pet() {}  
        // General methods
        void breath();
        virtual void speak();
    };// end class pet

void Pet::breath() {
    cout << "Gasp" << endl;
}

void Pet::speak() {
    cout << "Growl" << endl;
}
```

- The class `Pet` includes a constructor, destructor, and two virtual methods: `breath` and `speak`.
- For the `Pet` class, the `breath` method is defined in the class, while the `speak` method is declared as virtual to allow polymorphism.
- `Dog` and `Cat` classes inherit from `Pet` and override the `speak` method to define their own behavior.

- Example usage:

```
Pet* pet1 = new Dog();
Pet* pet2 = new Cat();
pet1->speak();
pet2->speak();
```

- This demonstrates how polymorphism allows runtime method selection for objects of derived classes.
class Dog: public Pet
{
    public:
        Dog() {}  
        void speak();
};

void Dog::speak()
{
    cout << "Hav-Hav" << endl;
}

class Cat: public Pet
{
    public:
        Cat() {}  
        void speak();
};

void Cat::speak()
{
    cout << "Meow" << endl;
}

void chorus(Pet pet, Pet *petPtr, Pet &petRef)
{
    pet.speak();
    petPtr->speak();
    petRef.speak();
}

int main()
{
    Pet *ptr; // Pointer to base class
    ptr = new Pet; cout << "Pet Created" << endl;
    ptr->speak(); ptr->speak(); ptr->speak();
    delete ptr; // Prevent memory leaks

    ptr = new Dog; cout << "Dog Created" << endl;
    ptr->speak(); ptr->speak(); ptr->speak();
    delete ptr; // Prevent memory leaks

    ptr = new Cat; cout << "Cat Created" << endl;
    ptr->speak(); ptr->speak(); ptr->speak();
    delete ptr; // Prevent memory leaks

    return 0;
} // End main
Private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to "friends".

Friends are functions or classes declared with the friend keyword.

A non-member function can access the private and protected members of a class if it is declared a friend of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend.
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    Rectangle() {} 
    Rectangle (int x, int y) : width(x), height(y) {} 
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param) {
    Rectangle res;
    res.width = param.width*2;
    res.height = param.height*2;
    return res;
}

int main () {
    Rectangle foo;
    Rectangle bar (2,3);
    foo = duplicate (bar);
    cout << foo.area() << '\n';
    return 0;
}

The duplicate function is a friend of class Rectangle. Therefore, function duplicate is able to access the members width and height (which are private) of different objects of type Rectangle. Notice though that neither in the declaration of duplicate nor in its later use in main, function duplicate is considered a member of class Rectangle. It isn't! It simply has access to its private and protected members without being a member.

Typical use cases of friend functions are operations that are conducted between two different classes accessing private or protected members of both.
Friend classes

Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class:

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

class Square;

class Rectangle {
  int width, height;
  public:
    int area ()
      { return (width * height); }
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  return 0;
}
```
In this example, class Rectangle is a friend of class Square allowing Rectangle's member functions to access private and protected members of Square. More concretely, Rectangle accesses the member variable Square::side, which describes the side of the square.

There is something else new in this example: at the beginning of the program, there is an empty declaration of class Square. This is necessary because class Rectangle uses Square (as a parameter in member convert), and Square uses Rectangle (declaring it a friend).

Friendships are never corresponded unless specified: In our example, Rectangle is considered a friend class by Square, but Square is not considered a friend by Rectangle. Therefore, the member functions of Rectangle can access the protected and private members of Square but not the other way around. Of course, Square could also be declared friend of Rectangle, if needed, granting such an access.

Another property of friendships is that they are not transitive: The friend of a friend is not considered a friend unless explicitly specified.

---

Function

```matlab
function y = mean(x)
% MEAN Average or mean value.
% For vectors, MEAN(x) returns the mean value.
% For matrices, MEAN(x) is a row vector containing the mean value of each column.
[m,n] = size(x);
if m == 1
    m = n;
end
y = sum(x)/m;
```

---

**Structure of a Function M-file**

- **Keyword**: function
- **Function Name (same as file name .m)**
- **Input Argument(s)**
- **Output Argument(s)**

```
function y = mean(x)
% MEAN Average or mean value.
% For vectors, MEAN(x) returns the mean value.
% For matrices, MEAN(x) is a row vector containing the mean value of each column.
[m,n] = size(x);
if m == 1
    m = n;
end
y = sum(x)/m;
```
What are MEX-files?

- MEX stands for MATLAB Executable.
- MEX-files are a way to call your custom C/C++ routines directly from MATLAB as if they were MATLAB built-in functions.
- Mex-files can be called exactly like M-functions in MATLAB.
- Here, all code examples will be presented in C.

Reasons for MEX-files

- The ability to call large existing C/C++ routines directly from MATLAB without having to rewrite them as M-files.
- Speed: you can rewrite bottleneck computations (like for-loops) as a MEX-file for efficiency.
Components of a MEX-file

- A gateway routine, **mexFunction**, that interfaces C/C++ and MATLAB data
- A computational routine, called from the gateway routine. This routine performs the computations that the MEX-file should implement
- Preprocessor macros, for building platform-independent code

The gateway routine

- The name of the gateway routine must be **mexFunction**.
  - **prhs** - an array of right-hand input arguments.
  - **plhs** - an array of left-hand output arguments.
  - **nrhs** - the number of right-hand arguments, or the size of the prhs array.
  - **nlhs** - the number of left-hand arguments, or the size of the plhs array.

```c
void mexFunction (int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    /* more C/C++ code ... */
}
```
Some important points

- The parameters `prhs`, `plhs`, `nrhs` and `nlhs` are required.
- The header file, `mex.h`, that declares the entry point and interface routines is also required.
- The name of the file with the gateway routine will be the command name in MATLAB.
- The file extension of the MEX-file is platform dependent.
  - The `mexext` function returns the extension for the current machine
- MATLAB is 1-based and C is 0-based

The computational routine

- The computational routine is called from the gateway routine
- It is a good idea to place the computational routine in a separate subroutine although it can be included in the gateway routine
**EXAMPLE: hellomatlab.c**

- The function description:
  - If a string is passed as an input, it displays the string as output
  - If no inputs are passed, a default string is passed as an output
  - An improper call (improper number of inputs/outputs) returns an error message as the output

```c
/* hellomatlab.c */
#include "mex.h"
void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
  if (nrhs < 1)
    plhs[0] = mxCreateString("Default String: Hello Users");
  else if (nrhs > 1)
    mexErrMsgTxt("Improper Inputs argument: Please pass 1 or no input arguments");
  else
    plhs[0] = mxDuplicateArray(prhs[0]);
}
```

**C compiler**

EXAMPLE: hellomatlab.c

- Setup the C-compiler to generate MEX-files
- Compile the hellomatlab.c and build the binary MEX-file
- Call the hellomatlab function with one input argument. See the results.
- Call the hellomatlab function without inputs.
- Call the hellomatlab function with improper number of inputs

```c
/* hellomatlab.c */
1.>> mex -setup
2.>> mex hellomatlab.c
3.>> hellomatlab (['My first MEX-file works!'])
   ans = My first MEX-file works!
4.>> hellomatlab
   ans = Default String: Hello Users
5.>> hellomatlab (['My first MEX-file works!'],['aaa'])
   Error using hellomatlab
   Improper Inputs argument: Please pass 1 or no input arguments
```

Compiler setup

`>> mex -setup`

1) Welcome to mex -setup. This utility will help you set up a default compiler. For a list of supported compilers, see [list of supported compilers](http://www.mathworks.com/support/compilers/R2013b/win64.html)

Please choose your compiler for building MEX-files:

Would you like mex to locate installed compilers [y]/n?

- put option - y
2) Please choose your compiler for building MEX-files:

Would you like mex to locate installed compilers [y]/n? y

Select a compiler:
[1] Microsoft Software Development Kit (SDK) 7.1 in C:\Program Files (x86)\Microsoft Visual Studio 10.0

[0] None

➤ put option - 1

Please verify your choices:

Compiler: Microsoft Software Development Kit (SDK) 7.1
Location: C:\Program Files (x86)\Microsoft Visual Studio 10.0

Are these correct [y]/n?

➤ put option - y
Working with mxArrays

For on-line help support it need to create an m-file with the same name as mex-file.

The function should be decelerated and help body should be written, but no function body.

The file should be saved as m file with same name as function.

% timestwo.m
function [out1,out2] = timestwo(in1,in2)
% timestwo takes 2 arguments and returns 2 ...
An additional examples

```c
#include "mex.h"
void timestwo(double y[], double x[], int m, int n)
{
    int j;
    for (j = 0; j < m*n; j++)
        y[j] = 2.0 * x[j];
}
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double *x, *y;
    int mrows, ncols;  /* Check for proper number of arguments. */
    if (nrhs != 1)
        mexErrMsgTxt("One input required.");
    else if (nlhs > 1)
        mexErrMsgTxt("Too many output arguments");
    mrows = mxGetM(prhs[0]);
    ncols = mxGetN(prhs[0]);
    plhs[0] = mxCreateDoubleMatrix(mrows,ncols, mxREAL);
    x = mxGetPr(prhs[0]); /* Assign pointers to each input and output. */
    y = mxGetPr(plhs[0]);
    timestwo(y,x,mrows,ncols); /* Call the timestwo subroutine. */
}
```

An additional examples

```c
#include "mex.h"
void timestwo_alt(double *y, double x)
{
    *y = 2.0 * x;
}
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double *y;
    double x;
    /* Create a 1-by-1 matrix for the return argument */
    plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
    /* Get the scalar value of the input x */
    x = mxGetScalar(prhs[0]);
    /* Get the scalar value of the input x */
    y = mxGetPr(plhs[0]);
    /* Assign a pointer to the output */
    timestwo_alt(y, x); /* Call the timestwo_alt subroutine */
}
```
Working with two cpp files

// timestwo.h must be in the current directory
#ifdef EXTFUNC_H
#define EXTFUNC_H

void extFunc(double y[], double x[], int m, int n);
#endif

- All CPP files should be added to compiled line
- For example:
  - mex C:\Work\MEX\Test2\timestwo.cpp
  - C:\Work\MEX\Test2\extFunc.cpp

Working with two cpp files

- External function definition

// extFunc.cpp file
#include "extFunc.h"
#include <iostream>
#include <iomanip>
using namespace std;

void extFunc(double y[], double x[], int m, int n)
{
    int j;
    for (j = 0; j < m*n; j++)
        y[j] = 2.0 * x[j];
    cout << "OK";
}

- Operator cout print “OK” in MATLAB console and helps to debug MEX functions
Working with two cpp files

```cpp
// timestwo.cpp file
#include "mex.h"
#include "extFunc.cpp"

void mexFunction int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]
{
    double *x, *y;
    int mrows, ncols; /* Check for proper number of arguments. */
    if (nrhs != 1)
        mexErrMsgTxt("One input required.");
    else if (nlhs > 1)
        mexErrMsgTxt("Too many output arguments");
    mrows = mxGetM(prhs[0]);
    ncols = mxGetN(prhs[0]);
    plhs[0] = mxCreateDoubleMatrix(mrows, ncols, mxREAL);
    x = mxGetPr(prhs[0]); /* Assign pointers to each input and output. */
    y = mxGetPr(plhs[0]);
    extFunc(y, x, mrows, ncols); /* Call the timestwo subroutine. */
}
```

Simple GUI examples

Simple GUI examples


MATLAB GUI

OBJECT ORIENTED PROGRAMMING LAB

1. The first OOP experiment (Exp. 1) is an obligation for all students in the advanced course computer lab.
2. The second and third OOP experiments (Exp. 2 & 3), respectively, are only for those students who choose to specialize in OOP.
3. Instead of doing experiments 5 & 6, students who choose to specialize in OOP can choose to do a final OOP project. In order to do this, the student has to send their idea for this project in an email attachment to Prof. A. and must be an approved project. Only then they can start working on it.

C++ tutorials

TUTORIALS AND EXPLANATIONS

Visual C++ / MFC

Online tutorials (*full in Russian)

- C++ Real World Web site
- Another C++ Real World Web site
- Internet Introduction to C++
- Internet C++ / Fast quizzes
- Internet Web site: Basic C to C++
- Internet Web site: C++ and OOP
- Internet MFC Web site
- DOP/History of C++
- Creating a simple MFC program
- Teach Yourself Visual C++ in 24 Hours
- VB6 Visual C++ Tutorials
- Visual C++/MFC Tutorial
- Borland C++ Builder group programming
- Advanced Programming Courses (C++ language)
The END