Classes

(c) 2011 Haim Michael. All Rights Reserved.

Introduction

- The definition of a new class in C++ includes two steps. We first need to declare about the it and then we need to define it.
- The declaration about the class methods and members is done in a separated file, also known as the hearder file. Its extension is `h`.
- The definition is done in a source code file. Its extension is `cpp`.

(c) 2011 Haim Michael. All Rights Reserved.

Class Declaration

- The class declaration is placed within a header file. The extension should be `h`. We should terminate the declaration with a semicolon.

(c) 2011 Haim Michael. All Rights Reserved.

Class Declaration

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double w, double h);
    double area();
    void setWidth(double w);
    void setHeight(double h);
    double getWidth();
    double getHeight();
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Class Definition

- The class definition is placed within a source code file. The extension should be `cpp`.

(c) 2011 Haim Michael. All Rights Reserved.

Class Definition

```
#include "rectangle.h"


Rectangle::Rectangle()
{
    setWidth(0);
    setHeight(0);
}

void Rectangle::setHeight(double h)
{
    if(h>0)
    {
        height = h;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Class Definition

```
void Rectangle::setWidth(double w)
{
    if(w>0)
    {
        width = w;
    }
}

double Rectangle::getWidth()
{
    return width;
}

double Rectangle::getHeight()
{
    return height;
}

double Rectangle::area()
{
    return getWidth()*getHeight();
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Class Instantiating

- Unlike Java and C#, when declaring a class type variable the variable itself is already an object.

```
...
Rectangle rec;
rec.setWidth(4);
rec.setHeight(3);
std::cout << rec.area();
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Class Instantiating

- We can use the new operator for creating a new object of our class. Calling new returns the address of the new object. We can assign the address to a pointer type variable.

```
...
Rectangle* rec;
rec = new Rectangle();
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Class Instantiating

- Working with an object that was instantiated using new we should use the `->` (arrow) operator.

```
...
Rectangle* rec;
rec = new Rectangle();
rec->setWidth(4);
rec->setHeight(3);
std::cout << rec->area();
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Class Instantiating

```
#include <iostream>
#include "stdio.h"
#include "rectangle.h"

int main(int argc, char *argv[])
{
    Rectangle obA;
    obA.setWidth(4);
    obA.setHeight(3);
    std::cout << obA.area() << std::endl;
    Rectangle* obB;
    obB = new Rectangle();
    obB->setWidth(4);
    obB->setHeight(3);
    std::cout << obB->area() << std::endl;
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Access Control

- Each and every method and each and every member in our class is subject to one of the three possible access specifiers: `public`, `protected` or `private`.
- Unlike Java and C#, when specifying an access specifier it applies all methods and all members declaration that follows it. Unlike Java and C# we don't need to place separated access specifiers for each and every method or member.

(c) 2011 Haim Michael. All Rights Reserved.

Access Control

- The default access specifier is `private`. Each and ever method as well as each and every member their declaration is before the first access specifier shall have the `private` access specifier.
- C++ allows us to define methods both in classes and in structs. Unlike a class, the default access specifier for a struct is `public`.

(c) 2011 Haim Michael. All Rights Reserved.

Access Control

- We place the access specifiers within the declaration in the header file only. We don't need to repeat it within the source code file, where the implementation resides.

(c) 2011 Haim Michael. All Rights Reserved.

The `public` Access Specifier

- Any code can call a `public` method or access a `public` member.
- We will define `public` methods when we want to allow any code the possibility to call them.
- We will define `public` members when we want to allow any code a direct access to them.

(c) 2011 Haim Michael. All Rights Reserved.

The `private` Access Specifier

- Only methods that belong to the same class where the `private` member was declared can access that member. Only methods that belong to the same class where the `private` method was declared can call it.
- We will define `private` methods and `private` members when we want to restrict their accessibility.

(c) 2011 Haim Michael. All Rights Reserved.

The `protected` Access Specifier

- Only methods that belong to the same class where the `protected` member was declared or belong to a class that inherit it can access that member. Only methods that belong to the same class where the `protected` method was declared or belong to a class that inherit it can call it.
- We will define `protected` methods and `protected` members when we want to restrict their accessibility to the very same class they belong to as well as to other classes that inherit it.

(c) 2011 Haim Michael. All Rights Reserved.

Code Clarity

- It is a good practice to group the declaration into groups in the following order: public, protected and private.

```
class NameOfOurClass
{
    public:
        //methods declaration
        //members declaration
    protected:
        //methods declaration
        //members declaration
    private:
        //methods declaration
        //members declaration
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Calling Other Methods

- We can call methods from within the code of other methods in the same class.

```
...
double Rectangle::area()
{
    return getWidth()*getHeight();
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

The `this` Pointer

- Each method call passes a pointer to the object on which it was called. This pointer is passed as a hidden parameter. The name of this hidden parameter is `this`.

```
...
void Rectangle::setWidth(double wVal)
{
   if(wVal>0)
    {
        this->width = wVal;
    }
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

The `this` Pointer

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double w, double h);
    double area();
    void setWidth(double w);
    void setHeight(double h);
    double getWidth();
    double getHeight();
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

The `this` Pointer

```
#include "rectangle.h"


Rectangle::Rectangle()
{
    this->setWidth(0);
    this->setHeight(0);
}

void Rectangle::setHeight(double h)
{
    if(h>0)
    {
        this->height = h;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

The `this` Pointer

```
void Rectangle::setWidth(double w)
{
    if(w>0)
    {
        this->width = w;
    }
}

double Rectangle::getWidth()
{
    return this->width;
}

double Rectangle::getHeight()
{
    return this->height;
}

double Rectangle::area()
{
    return this->getWidth()*this->getHeight();
}
```

(c) 2011 Haim Michael. All Rights Reserved.

The `this` Pointer

```
#include <iostream>
#include "stdio.h"
#include "rectangle.h"

int main(int argc, char *argv[])
{
    Rectangle* ob;
    ob = new Rectangle();
    ob->setWidth(4);
    ob->setHeight(3);
    std::cout << ob->area() << std::endl;
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Objects on The Stack

- Declaring a simple class type variable will indirectly instantiate that class. The variable will actually be an object of that class.
- We will use the . (dot) operator for accessing the object members and for calling methods on it.

```
...
Rectangle rec;
rec.setWidth(4);
rec.setHeight(3);
std::cout << rec.area();
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Objects on The Heap

- Declaring a class type pointer variable willl allow us to assign it with an address for objecet we instantiate from that class.
- We will use the `->` (arrow) operator for accessing the object members and for calling methods on it.

```
...
Rectangle* rec;
rec = new Rectangle();
rec->setWidth(4);
rec->setHeight(3);
std::cout << rec->area();
...
```

(c) 2011 Haim Michael. All Rights Reserved.

The Object Life Cycle

- The object life cycle includes three activities: creation, assignment and destruction.
- While the first activity (creation) applies in all cases, the other two don't. Objects are not necessary assigned nor destructed.

(c) 2011 Haim Michael. All Rights Reserved.

Define Constructors

- The constructor responsibles for initializing the object. This initialization ensures that its members don't include any none valid values.
- The name of the constructor is the same as the name of the class.
- The constructor doesn't have a return type.
- Similarly to methods, we can declare more than one constructor as long as each and every one of them has a unique signature.

(c) 2011 Haim Michael. All Rights Reserved.

Define Constructors

- Calling a constructor is feasible both for objects created on the stack and for objects created on the heap.
- It is highly important to call delete on each and every object created on the heap.
- We cannot explicitly call one constructor from another. The result will be the creation of a new object.

(c) 2011 Haim Michael. All Rights Reserved.

Define Constructors

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double w, double h);
    double area();
    void setWidth(double w);
    void setHeight(double h);
    double getWidth();
    double getHeight();
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Define Constructors

```
#include "rectangle.h"


Rectangle::Rectangle()
{
    this->setWidth(0);
    this->setHeight(0);
}

Rectangle::Rectangle(double w, double h)
{
    this->setHeight(h);
    this->setWidth(w);
}

void Rectangle::setHeight(double h)
{
    if(h>0)
    {
        this->height = h;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

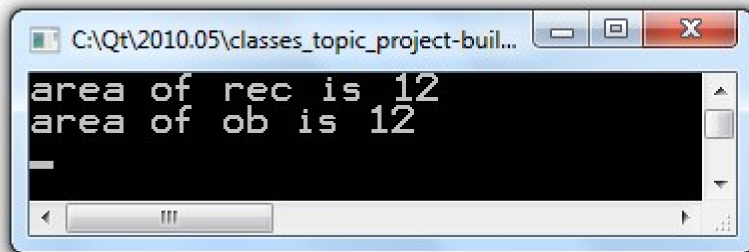
Define Constructors

```
void Rectangle::setWidth(double w)
{
    if(w>0)
    {
        this->width = w;
    }
}

double Rectangle::getWidth()
{
    return this->width;
}

double Rectangle::getHeight()
{
    return this->height;
}

double Rectangle::area()
{
    return this->getWidth()*this->getHeight();
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Define Constructors

```
#include <iostream>
#include "stdio.h"
#include "rectangle.h"

int main(int argc, char *argv[])
{
    Rectangle rec(3,4);
    std::cout << "area of rec is " << rec.area() << std::endl;
    Rectangle* ob;
    ob = new Rectangle(3,4);
    std::cout << "area of ob is " << ob->area() << std::endl;
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

Default Constructors

- The default constructor is the one that automatically exists in each and every class as long as we don't define constructors of our own.
- The default constructor takes no arguments and is also known as the zero arguments constructor.
- It is common to define a zero arguments constructor in order to take the place of the default constructor. This way we can properly initialize the object.

(c) 2011 Haim Michael. All Rights Reserved.

Initializer Lists

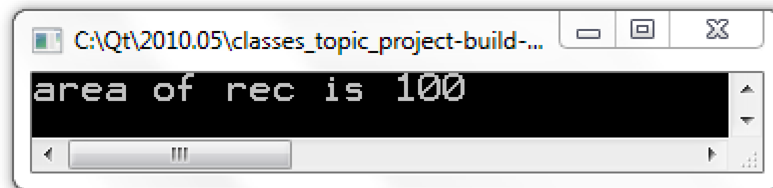
- C++ allows us to initialize the members of a new instantiated object through the initializer list.
- It is an alternative way for initializing using code written within the constructor.

```
...
Rectangle::Rectangle() :: width(0), height(0)
{
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Initializer Lists

```
#include <iostream>
#include "stdio.h"
#include "rectangle.h"

int main(int argc, char *argv[])
{
    Rectangle rec;
    std::cout << "area of rec is " << rec.area()
        << std::endl;
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

Initializer Lists

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double w, double h);
    double area();
    void setWidth(double w);
    void setHeight(double h);
    double getWidth();
    double getHeight();
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Initializer Lists

```
#include "rectangle.h"


Rectangle::Rectangle() : width(10), height(10)
{

}

double Rectangle::area()
{
    return this->width*this->height;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Initializer Lists

- The initializer list gets into action when the object is created. Initialization code within the constructor gets into action afterwards. This difference sets the initialization through the initializer list as the more efficient option.

(c) 2011 Haim Michael. All Rights Reserved.

Const Data Members Initialization

- The const data members can be initialized through the initializer list only. They cannot be initialized through code within the constructor because that code is executed after the variable was already created.

(c) 2011 Haim Michael. All Rights Reserved.

Reference Data Members Initialization

- The reference data members cannot be initialized within the constructor. They cannot exist without referring something.
- We can initialize reference data members using the initializer list only.

(c) 2011 Haim Michael. All Rights Reserved.

Object Data Members Initialization

- When having a member which is an object that should be instantiated from a class that doesn't have a zero argument constructor the only way to initialize it is using the initializer list.

(c) 2011 Haim Michael. All Rights Reserved.

Initialize List Initialization Order

- The members are initialized according to their order in the class definition. Their order in the list initializer has no effect.

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

- The copy constructor allows us to create an object which is an exact copy of another object.
- Unless we define our own copy constructor C++ will auto generate one. The autogenerated one simply copies one value into the other.
- When the original object holds addresses in its members the autogenerated copy constructor won't be sufficient.

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

geometry.h

```
class Point
{
public:
    Point();
    Point(const Point &src);
    Point(double x,double y);
    double x;
    double y;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

```
class Line
{
public:
    Line();
    Line(const Line &src);
    Line(Point* a,Point* b);
    void setPointA(Point* p);
    void setPointB(Point* p);
    Point* getPointA();
    Point* getPointB();
    void details();
    void triple();
private:
    Point* a;
    Point* b;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

geometry.cpp

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

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

Point::Point(const Point &src)
{
    this->x = src.x;
    this->y = src.y;
}

Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

```
Line::Line()
{
    a = new Point(10,10);
    b = new Point(10,10);
}

Line::Line(Point* a, Point* b)
{
    this->setPointA(a);
    this->setPointB(b);
}

Line::Line(const Line &src)
{
    a = new Point(src.a->x,src.b->y);
    b = new Point(src.a->x,src.b->y);
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

```
void Line::setPointA(Point* p)
{
    a = p;
}

void Line::setPointB(Point* p)
{
    b = p;
}

Point* Line::getPointA()
{
    return a;
}

Point* Line::getPointB()
{
    return b;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

```
void Line::details()
{
    std::cout << "point a  (" << a->x << "," << a->y << ")"
        << std::endl;
    std::cout << "point b  (" << b->x << "," << b->y << ")"
        << std::endl;
}

void Line::triple()
{
    a->x*=3;
    a->y*=3;
    b->x*=3;
    b->y*=3;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

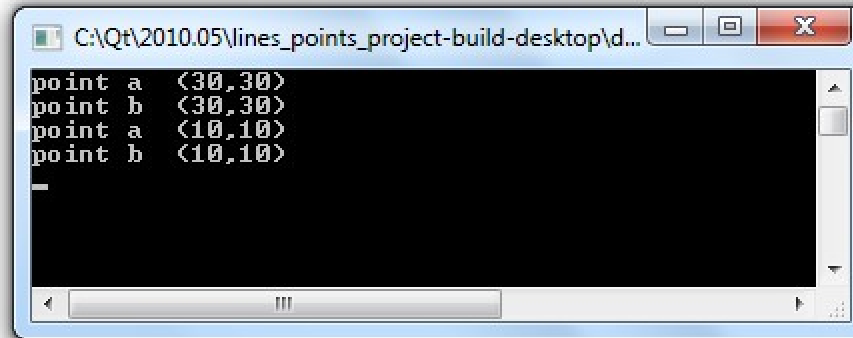
Copy Constructors

main.cpp

```
#include "stdio.h"
#include "geometry.h"

int main(int argc, char *argv[])
{
    Line one;
    Line two = one;
    one.triple();
    one.details();
    two.details();
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors



The Output

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

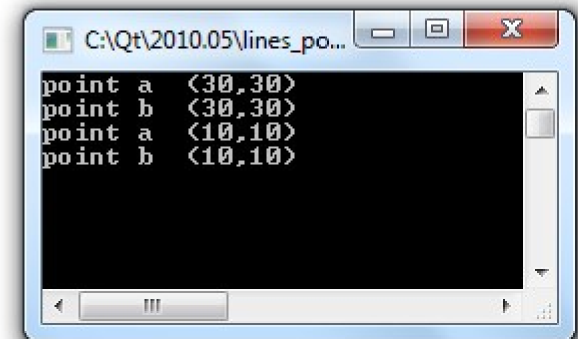
- When passing an argument to a function it is passed by value. This is the default behavior. The method receives a copy of the variable. It doesn't receive the variable itself.
- Thus, whenever we pass an object as argument to a function the compiler calls the copy constructor in order to initialize the new created object.
- The copy constructor is also called whenever a function returns an object.

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

- We can explicitly call the copy constructor. We will do so when there is a need to construct one object as an exact copy of another one.

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Line one;
    Line two(one);
    one.triple();
    one.details();
    two.details();
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

Passing Objects By Reference

- Passing objects by reference is usually more efficient comparing with passing them by value. The copy constructor is not invoked.
- Marking the parameter with `const` will ensure that the object is not changed during the execution of the function. Other developers won't need to worry about that possibility.

```
...
void Line::setPointA(const Point& p);
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Object Destruction

- When the object is destroyed the destructor method is called. The purpose of the destructor is to cleanup the memory that object was responsible for.
- Objects on the stack are automatically destroyed when the execution goes beyond their scope. In other words, whenever the code encounters an ending curly brace all objects that were created on the stack within those curly braces are destroyed.

(c) 2011 Haim Michael. All Rights Reserved.

Object Destruction

- Objects on the heap are not automatically destroyed. There is a need to `delete` them.

```
...
Rectangle* rec;
rec = new Rectangle(4,3);
...
delete rec;
...
```

- We will usually define the destructors for taking care of deleting objects on the heap.

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

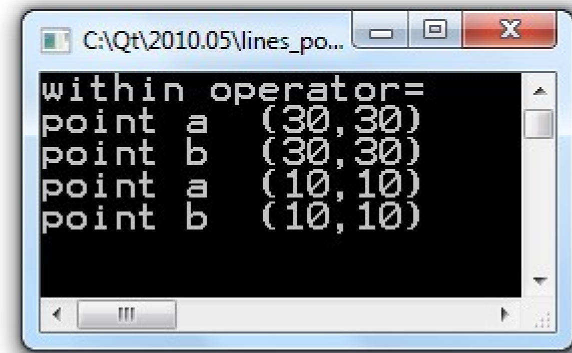
- The assignment operator gets into action when assigning one object into another. Unless we write one, C++ writes one for us.
- The default C++ assignment behavior is nearly identical to the default copy constructor behavior. Unlike the copy constructor, the assignment operator returns a reference to an object. That allows us to chain assignments with each other.

```
...
Rectangle rec;
rec = otherRec = goodRec = anotherRec;
...
```

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Line one;
    Line two;
    two = one;
    one.triple();
    one.details();
    two.details();
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
class Point
{
public:
    Point();
    Point(const Point &src);
    Point(double x,double y);
    double x;
    double y;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
class Line
{
public:
    Line();
    Line(const Line &src);
    Line(Point* a,Point* b);
    Line& operator=(const Line& other);
    void setPointA(Point* p);
    void setPointB(Point* p);
    Point* getPointA();
    Point* getPointB();
    void details();
    void triple();
private:
    Point* a;
    Point* b;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

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

Point::Point(const Point &src)
{
    this->x = src.x;
    this->y = src.y;
}

Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
Line::Line()
{
    a = new Point(10,10);
    b = new Point(10,10);
}

Line::Line(Point* a, Point* b)
{
    this->setPointA(a);
    this->setPointB(b);
}

Line::Line(const Line &src)
{
    a = new Point(src.a->x,src.b->y);
    b = new Point(src.a->x,src.b->y);
}

void Line::setPointA(Point* p)
{
    a = p;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
void Line::setPointB(Point* p)
{
    b = p;
}

Point* Line::getPointA()
{
    return a;
}

Point* Line::getPointB()
{
    return b;
}

void Line::details()
{
    std::cout << "point a  (" << a->x << "," << a->y << ")"
        << std::endl;
    std::cout << "point b  (" << b->x << "," << b->y << ")"
        << std::endl;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
void Line::triple()
{
    a->x*=3;
    a->y*=3;
    b->x*=3;
    b->y*=3;
}

Line& Line::operator=(const Line& other)
{
    std::cout << "within operator=" << std::endl;
    if(this==&other)
    {
        return (*this);
    }
    a = new Point(other.a->x,other.a->y);
    b = new Point(other.b->x,other.b->y);
    return (*this);
}
```

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

- The = operator does not always mean assignment. When placed on the same line where the variable is declared it functions as a shorthand for the copy constructor.

```
...
Rec rec = otherRec;
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Dynamic Memory Allocation

- When we don't know how much memory will be needed before the code actually runs we can dynamically allocate the require memory during the execution itself.
- When our object dynamically allocates the required memory we should pay attention to the copy constructor, assignment operator and the destructor defined in its class.

(c) 2011 Haim Michael. All Rights Reserved.

Freeing Memory with Destructors

- The destructor is executed when the object reaches the end of its life.
- The destructor has the same name as the name of the class preceeded by ~ (tilde).
- The destructor doesn't take any parameter and each class can include the definition for one destructor only.
- In general, the destructor frees the memory that was allocated in the constructor.

(c) 2011 Haim Michael. All Rights Reserved.

Freeing Memory with Destructors

```
class Line
{
public:
    Line();
    Line(const Line &src);
    Line(Point* a,Point* b);
    ~Line();
    Line& operator=(const Line& other);
    static Line getLine();
    void setPointA(Point* p);
    void setPointB(Point* p);
    Point* getPointA();
    Point* getPointB();
    void details();
    void triple();
private:
    Point* a;
    Point* b;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Freeing Memory with Destructors

```
...

Line::Line(const Line &src)
{
    std::cout << "within copy constructor" << std::endl;
    a = new Point(src.a->x,src.b->y);
    b = new Point(src.a->x,src.b->y);
}

Line::~Line()
{
    delete a;
    delete b;
}

...
```

(c) 2011 Haim Michael. All Rights Reserved.

Disabling Pass By Value

- We can disable passing by value and disable assignment by marking the copy constructor and the `operator=` definitions with the `private` access specifier.

- Doing so, it won't be possible to compile code that that tries to pass over the object by value, return it from a function or assign to it.

(c) 2011 Haim Michael. All Rights Reserved.

Static Data Members

- Static data member is a data member associated with a class instead of object.

```
...
class Rectangle
{
    public:
        static int sCounter;
        double width;
        double height;
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Static Data Members

- In order to access a static member we need to prefix it with the name of the class togeher with the `::` operator.

```
...
std::cout << Rectangle::sCounter;
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Const Data Members

- We can declare our members together with the `const` modifier. That will turn them into constants.
- The `const` data members are usually also static ones. Usually, it doesn't make sense to keep the same value in all objects.

```
...
class Car
{
    public:
        ...
        static const double maxSpeed = 180;
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Static Methods

- We can define static methods. Static methods don't apply specifically to each object. Static methods apply to the class as a whole.

```
...
class Rectangle
{
   public:
      Rectangle static unite(Rectangle a, Rectangle b);
      ...
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Static Methods

- Calling a static method is done similarly to the way in which we access static members.

```
...
Rectangle recA(4,3);
Rectangle recB(5,2);
Rectangle rec = Rectangle::unite(recA,recB);
...
```

- Static methods cannot access non static members.

(c) 2011 Haim Michael. All Rights Reserved.

Const Methods

- We can mark a method with `const` in order to ensure that it doesn't change any data member.

```
...
class Rectangle
{
   public:
      double area() const;
      ...
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Const Methods

- Objects marked with `const` can call `const` methods only. Others can call all sorts of methods. Both the ones marked with `const` and the ones that arenot.

(c) 2011 Haim Michael. All Rights Reserved.

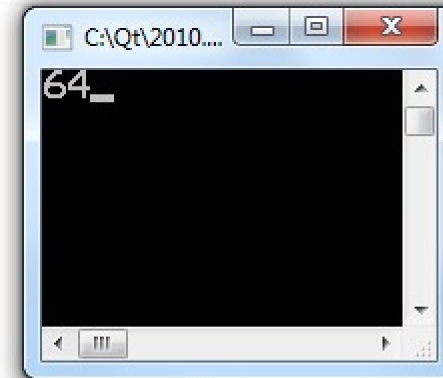
Functions Overloading

- The C++ programming language supports overloading. We can overload any function, method and constructor as many times as we want as long as the number and/or the types of the parameters differ.

(c) 2011 Haim Michael. All Rights Reserved.

Functions Overloading

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle rec(8);
    std::cout << rec.area();
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

Functions Overloading

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    double area();
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Functions Overloading

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Functions Overloading

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}

double Rectangle::area()
{
    return width*height;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Default Parameters

- We can specify default values for function and method parameters. If the user won't specify arguments for those parameters the defaults will be used.
- Setting defaults we can do it for a continuous list of parameters starting from the rightmost parameter only.

(c) 2011 Haim Michael. All Rights Reserved.

Default Parameters

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle a(4);
    std::cout << a.area();
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Default Parameters

```
class Rectangle
{
public:
    Rectangle(double w=10,double h=10);
    double area() {return width*height;}
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Default Parameters

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Inline Methods

- The C++ programming language allows us to define inline methods. Inline methods are methods the compiler insert their body directly into the code instead of each and every call for their execusion.
- The process involved with inline methods is just a simpler version for using the `#define` macro.
- We can specify an inline method or a function by placing the `inline` keyword infront of its name in the source code file where we define it.

(c) 2011 Haim Michael. All Rights Reserved.

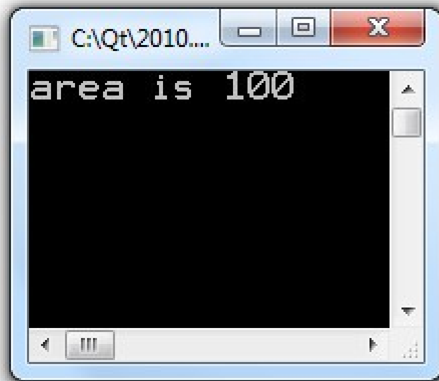
Inline Methods

- We can alternatively include the implementation of the method within the header file and taking it out from the source code file. Doing so will have the same result.

(c) 2011 Haim Michael. All Rights Reserved.

Inline Methods

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle* rec = new Rectangle(20,5);
    double total = rec->area();
    std::cout << "area is " << total << std::endl;
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

Inline Methods

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    double area() {return width*height;}
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Inline Methods

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Inline Methods

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Inline Methods

- When we define an inline method the compiler doesn't want to inline it might silently ignore our directive.
- Inline methods can lead to code bloat. We should use inline method and functions in a careful way.

(c) 2011 Haim Michael. All Rights Reserved.

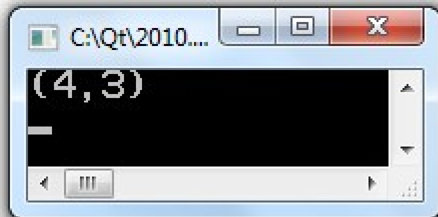
Nested Classes

- The class definition can include more than just methods and members. The class definition can include inner classes, inner structs and inner enums.
- Using the inner class (or the inner struct... or the inner enum) outside the scope of the outer class depends on their access specifiers.
- Referring an inner type should be done using its full qualified name.

(c) 2011 Haim Michael. All Rights Reserved.

Nested Classes

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle* rec = new Rectangle(20,5);
    rec->setLocation(Rectangle::Point(4,3));
    rec->getLocation().details();
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

Nested Classes

```
class Rectangle
{
public:
    class Point
    {
    public:
        Point();
        Point(double x, double y);
        void details();
        double x;
        double y;
    };
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    double area() {return width*height;}
    void setLocation(Point point);
    Point getLocation();
private:
    double width;
    double height;
    Point location;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Nested Classes

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Nested Classes

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}

Rectangle::Point Rectangle::getLocation()
{
    return location;
}

void Rectangle::setLocation(Point point)
{
    location = point;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Nested Classes

```
void Rectangle::Point::details()
{
    std::cout << "(" << x << "," << y << ")" <<
std::endl;
}

Rectangle::Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}

Rectangle::Point::Point()
{
    this->x = 10;
    this->x = 10;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

The `typedef` Keyword

- We can use the `typedef` keyword to create an alias for a data type.

```
typedef originalname newname;
```

(c) 2011 Haim Michael. All Rights Reserved.

The `typedef` Keyword

```
class Rectangle
{
public:
    class Point
    {
    public:
        Point();
        Point(double x, double y);
        void details();
        double x;
        double y;
    };
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    double area() {return width*height;}
    void setLocation(Point point);
    Point getLocation();
private:
    double width;
    double height;
    Point location;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

The `typedef` Keyword

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

The `typedef` Keyword

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}

Rectangle::Point Rectangle::getLocation()
{
    return location;
}

void Rectangle::setLocation(Point point)
{
    location = point;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

The `typedef` Keyword

```
typedef Rectangle::Point RecPoint;

//void Rectangle::Point::details()
void RecPoint::details()
{
    std::cout << "(" << x << "," << y << ")" << std::endl;
}

//Rectangle::Point::Point(double x, double y)
RecPoint::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}

//Rectangle::Point::Point()
RecPoint::Point()
{
    this->x = 10;
    this->x = 10;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

The `typedef` Keyword

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle* rec = new Rectangle(20,5);
    rec->setLocation(Rectangle::Point(4,3));
    rec->getLocation().details();
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Friends

- The C++ programming language allows us to declare that other classeses or nonmember functions are friends. Becoming friends they can access private and protected members and methods.

```
...
Class Car
{
    public:
        friend class Person;
        friend bool checkEngine(const Car& car);
}
...
```

The Person class is a friend of Car

We can declare a global function within our class together with the word friend. That will be sufficient both for declaring about the function and for making it a friend that can access Private and protected members of the class Car.

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

- The C++ programming language allows us to overload the operators we know with a specific definition for our class type.

```
...
Class Rectangle
{
    public:
        Rectangle();
        Rectangle operator+(Rectangle& other);
        double area();
    private:
        double width;
        double height;
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle a(2,3);
    Rectangle b(3,4);
    Rectangle c;
    c = a + b;
    std::cout << c.area();
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    Rectangle operator+(Rectangle& other);
    double area() {return width*height;}
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}

Rectangle Rectangle::operator+(Rectangle& other)
{
    return Rectangle(
        this->width+other.width,
        this->height+other.height);
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

- We can alternatively overload the operator by declaring a global function.

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle a(2,3);
    Rectangle b(3,4);
    Rectangle c;
    c = a + b;
    std::cout << c.area();
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    //Rectangle operator+(Rectangle& other);
    double area() {return width*height;}
    friend Rectangle operator+(Rectangle& a,Rectangle& b);
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}

Rectangle operator+(Rectangle& a, Rectangle& b)
{
    return Rectangle(a.width+b.width,a.height+b.height);
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    //Rectangle operator+(Rectangle& other);
    double area() {return width*height;}
    friend Rectangle operator+(Rectangle& a,Rectangle& b);
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

- We can overload all other operators by defining similar functions.

```
Rectangle operator+(Rectangle& a, Rectangle& b)
Rectangle operator-(Rectangle& a, Rectangle& b)
Rectangle operator*(Rectangle& a, Rectangle& b)
Rectangle operator\(Rectangle& a, Rectangle& b)
Rectangle operator+=(Rectangle& a, Rectangle& b)
Rectangle operator-=(Rectangle& a, Rectangle& b)
Rectangle operator*=(Rectangle& a, Rectangle& b)
Rectangle operator\=(Rectangle& a, Rectangle& b)
bool operator==(Rectangle& a, Rectangle& b)
bool operator<(Rectangle& a, Rectangle& b)
bool operator>(Rectangle& a, Rectangle& b)
bool operator!=(Rectangle& a, Rectangle& b)

...
```

(c) 2011 Haim Michael. All Rights Reserved.

Classes

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Introduction

- The definition of a new class in C++ includes two steps. We first need to declare about the it and then we need to define it.
- The declaration about the class methods and members is done in a separated file, also known as the hearder file. Its extension is `h`.
- The definition is done in a source code file. Its extension is `cpp`.

Class Declaration

- The class declaration is placed within a header file. The extension should be `h`. We should terminate the declaration with a semicolon.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Class Declaration

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double w, double h);
    double area();
    void setWidth(double w);
    void setHeight(double h);
    double getWidth();
    double getHeight();
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved. 4

Class Definition

- The class definition is placed within a source code file. The extension should be `cpp`.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Class Definition

```
#include "rectangle.h"


Rectangle::Rectangle()
{
    setWidth(0);
    setHeight(0);
}

void Rectangle::setHeight(double h)
{
    if(h>0)
    {
        height = h;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved. 6

Class Definition

```
void Rectangle::setWidth(double w)
{
    if(w>0)
    {
        width = w;
    }
}

double Rectangle::getWidth()
{
    return width;
}

double Rectangle::getHeight()
{
    return height;
}

double Rectangle::area()
{
    return getWidth()*getHeight();
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Class Instantiating

- Unlike Java and C#, when declaring a class type variable the variable itself is already an object.

```
...
Rectangle rec;
rec.setWidth(4);
rec.setHeight(3);
std::cout << rec.area();
...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Class Instantiating

- We can use the new operator for creating a new object of our class. Calling new returns the address of the new object. We can assign the address to a pointer type variable.

```
...
Rectangle* rec;
rec = new Rectangle();
...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Class Instantiating

- Working with an object that was instantiated using new we should use the -> (arrow) operator.

```
...
Rectangle* rec;
rec = new Rectangle();
rec->setWidth(4);
rec->setHeight(3);
std::cout << rec->area();
...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Class Instantiating

```
#include <iostream>
#include "stdio.h"
#include "rectangle.h"

int main(int argc, char *argv[])
{
    Rectangle obA;
    obA.setWidth(4);
    obA.setHeight(3);
    std::cout << obA.area() << std::endl;
    Rectangle* obB;
    obB = new Rectangle();
    obB->setWidth(4);
    obB->setHeight(3);
    std::cout << obB->area() << std::endl;
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Access Control

- Each and every method and each and every member in our class is subject to one of the three possible access specifiers: `public`, `protected` or `private`.
- Unlike Java and C#, when specifying an access specifier it applies all methods and all members declaration that follows it. Unlike Java and C# we don't need to place separated access specifiers for each and every method or member.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Access Control

- The default access specifier is `private`. Each and ever method as well as each and every member their declaration is before the first access specifier shall have the `private` access specifier.
- C++ allows us to define methods both in classes and in structs. Unlike a class, the default access specifier for a struct is `public`.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Access Control

- We place the access specifiers within the declaration in the header file only. We don't need to repeat it within the source code file, where the implementation resides.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The `public` Access Specifier

- Any code can call a `public` method or access a `public` member.
- We will define `public` methods when we want to allow any code the possibility to call them.
- We will define `public` members when we want to allow any code a direct access to them.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The `private` Access Specifier

- Only methods that belong to the same class where the `private` member was declared can access that member. Only methods that belong to the same class where the `private` method was declared can call it.
- We will define `private` methods and `private` members when we want to restrict their accessibility.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The `protected` Access Specifier

- Only methods that belong to the same class where the `protected` member was declared or belong to a class that inherit it can access that member. Only methods that belong to the same class where the `protected` method was declared or belong to a class that inherit it can call it.
- We will define `protected` methods and `protected` members when we want to restrict their accessibility to the very same class they belong to as well as to other classes that inherit it.

(c) 2011 Haim Michael. All Rights Reserved. 17

Code Clarity

- It is a good practice to group the declaration into groups in the following order: public, protected and private.

```
class NameOfOurClass
{
    public:
        //methods declaration
        //members declaration
    protected:
        //methods declaration
        //members declaration
    private:
        //methods declaration
        //members declaration
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Calling Other Methods

- We can call methods from within the code of other methods in the same class.

```
...
double Rectangle::area()
{
    return getWidth()*getHeight();
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The `this` Pointer

- Each method call passes a pointer to the object on which it was called. This pointer is passed as a hidden parameter. The name of this hidden parameter is `this`.

```
...
void Rectangle::setWidth(double wVal)
{
   if(wVal>0)
    {
        this->width = wVal;
    }
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The this Pointer

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double w, double h);
    double area();
    void setWidth(double w);
    void setHeight(double h);
    double getWidth();
    double getHeight();
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved. 21

(c) 2011 Haim Michael. All Rights Reserved.

The this Pointer

```
#include "rectangle.h"


Rectangle::Rectangle()
{
    this->setWidth(0);
    this->setHeight(0);
}

void Rectangle::setHeight(double h)
{
    if(h>0)
    {
        this->height = h;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The `this` Pointer

```
void Rectangle::setWidth(double w)
{
    if(w>0)
    {
        this->width = w;
    }
}

double Rectangle::getWidth()
{
    return this->width;
}

double Rectangle::getHeight()
{
    return this->height;
}

double Rectangle::area()
{
    return this->getWidth()*this->getHeight();
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The `this` Pointer

```
#include <iostream>
#include "stdio.h"
#include "rectangle.h"

int main(int argc, char *argv[])
{
    Rectangle* ob;
    ob = new Rectangle();
    ob->setWidth(4);
    ob->setHeight(3);
    std::cout << ob->area() << std::endl;
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Objects on The Stack

- Declaring a simple class type variable will indirectly instantiate that class. The variable will actually be an object of that class.
- We will use the . (dot) operator for accessing the object members and for calling methods on it.

```
...
Rectangle rec;
rec.setWidth(4);
rec.setHeight(3);
std::cout << rec.area();
...
```

(c) 2011 Haim Michael. All Rights Reserved. 25

Objects on The Heap

- Declaring a class type pointer variable willl allow us to assign it with an address for objecet we instantiate from that class.
- We will use the -> (arrow) operator for accessing the object members and for calling methods on it.

```
...
Rectangle* rec;
rec = new Rectangle();
rec->setWidth(4);
rec->setHeight(3);
std::cout << rec->area();
...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The Object Life Cycle

- The object life cycle includes three activities: creation, assignment and destruction.
- While the first activity (creation) applies in all cases, the other two don't. Objects are not necessary assigned nor destructed.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Define Constructors

- The constructor responsibles for initializing the object. This initialization ensures that its members don't include any none valid values.
- The name of the constructor is the same as the name of the class.
- The constructor doesn't have a return type.
- Similarly to methods, we can declare more than one constructor as long as each and every one of them has a unique signature.

(c) 2011 Haim Michael. All Rights Reserved. 28

Define Constructors

- Calling a constructor is feasible both for objects created on the stack and for objects created on the heap.
- It is highly important to call delete on each and every object created on the heap.
- We cannot explicitly call one constructor from another. The result will be the creation of a new object.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Define Constructors

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double w, double h);
    double area();
    void setWidth(double w);
    void setHeight(double h);
    double getWidth();
    double getHeight();
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved. 30

Define Constructors

```
#include "rectangle.h"


Rectangle::Rectangle()
{
    this->setWidth(0);
    this->setHeight(0);
}

Rectangle::Rectangle(double w, double h)
{
    this->setHeight(h);
    this->setWidth(w);
}

void Rectangle::setHeight(double h)
{
    if(h>0)
    {
        this->height = h;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved. 31

Define Constructors

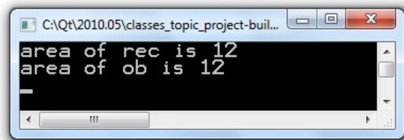
```
void Rectangle::setWidth(double w)
{
    if(w>0)
    {
        this->width = w;
    }
}

double Rectangle::getWidth()
{
    return this->width;
}

double Rectangle::getHeight()
{
    return this->height;
}

double Rectangle::area()
{
    return this->getWidth()*this->getHeight();
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Define Constructors

```
#include <iostream>
#include "stdio.h"
#include "rectangle.h"

int main(int argc, char *argv[])
{
    Rectangle rec(3,4);
    std::cout << "area of rec is " << rec.area() << std::endl;
    Rectangle* ob;
    ob = new Rectangle(3,4);
    std::cout << "area of ob is " << ob->area() << std::endl;
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved. 33

Default Constructors

- The default constructor is the one that automatically exists in each and every class as long as we don't define constructors of our own.
- The default constructor takes no arguments and is also known as the zero arguments constructor.
- It is common to define a zero arguments constructor in order to take the place of the default constructor. This way we can properly initialize the object.

(c) 2011 Haim Michael. All Rights Reserved. 34

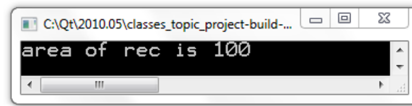
Initializer Lists

- C++ allows us to initialize the members of a new instantiated object through the initializer list.
- It is an alternative way for initializing using code written within the constructor.

```
...
Rectangle::Rectangle() :: width(0), height(0)
{
}
...
```

Initializer Lists

```
#include <iostream>
#include "stdio.h"
#include "rectangle.h"

int main(int argc, char *argv[])
{
    Rectangle rec;
    std::cout << "area of rec is " << rec.area()
        << std::endl;
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

Initializer Lists

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double w, double h);
    double area();
    void setWidth(double w);
    void setHeight(double h);
    double getWidth();
    double getHeight();
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Initializer Lists

```
#include "rectangle.h"


Rectangle::Rectangle() : width(10), height(10)
{

}

double Rectangle::area()
{
    return this->width*this->height;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Initializer Lists

- The initializer list gets into action when the object is created. Initialization code within the constructor gets into action afterwards. This difference sets the initialization through the initializer list as the more efficient option.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Const Data Members Initialization

- The const data members can be initialized through the initializer list only. They cannot be initialized through code within the constructor because that code is executed after the variable was already created.

(c) 2011 Haim Michael. All Rights Reserved. 40

Reference Data Members Initialization

- The reference data members cannot be initialized within the constructor. They cannot exist without referring something.
- We can initialize reference data members using the initializer list only.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Object Data Members Initialization

- When having a member which is an object that should be instantiated from a class that doesn't have a zero argument constructor the only way to initialize it is using the initializer list.

(c) 2011 Haim Michael. All Rights Reserved.

Initialize List Initialization Order

- The members are initialized according to their order in the class definition. Their order in the list initializer has no effect.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

- The copy constructor allows us to create an object which is an exact copy of another object.
- Unless we define our own copy constructor C++ will auto generate one. The autogenerated one simply copies one value into the other.
- When the original object holds addresses in its members the autogenerated copy constructor won't be sufficient.

(c) 2011 Haim Michael. All Rights Reserved. 44

Copy Constructors

geometry.h

```
class Point
{
public:
    Point();
    Point(const Point &src);
    Point(double x,double y);
    double x;
    double y;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

```
class Line
{
public:
    Line();
    Line(const Line &src);
    Line(Point* a,Point* b);
    void setPointA(Point* p);
    void setPointB(Point* p);
    Point* getPointA();
    Point* getPointB();
    void details();
    void triple();
private:
    Point* a;
    Point* b;
};
```

(c) 2011 Haim Michael. All Rights Reserved. 46

Copy Constructors

geometry.cpp

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

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

Point::Point(const Point &src)
{
    this->x = src.x;
    this->y = src.y;
}

Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

```
Line::Line()
{
    a = new Point(10,10);
    b = new Point(10,10);
}

Line::Line(Point* a, Point* b)
{
    this->setPointA(a);
    this->setPointB(b);
}

Line::Line(const Line &src)
{
    a = new Point(src.a->x,src.b->y);
    b = new Point(src.a->x,src.b->y);
}
```

(c) 2011 Haim Michael. All Rights Reserved. 48

Copy Constructors

```
void Line::setPointA(Point* p)
{
    a = p;
}

void Line::setPointB(Point* p)
{
    b = p;
}

Point* Line::getPointA()
{
    return a;
}

Point* Line::getPointB()
{
    return b;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

```
void Line::details()
{
    std::cout << "point a  (" << a->x << "," << a->y << ")"
        << std::endl;
    std::cout << "point b  (" << b->x << "," << b->y << ")"
        << std::endl;
}

void Line::triple()
{
    a->x*=3;
    a->y*=3;
    b->x*=3;
    b->y*=3;
}
```

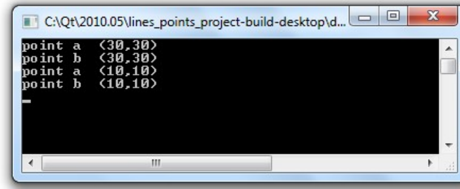
(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

main.cpp

```
#include "stdio.h"
#include "geometry.h"

int main(int argc, char *argv[])
{
    Line one;
    Line two = one;
    one.triple();
    one.details();
    two.details();
    getchar();
    return 0;
}
```

Copy Constructors



The Output

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Copy Constructors

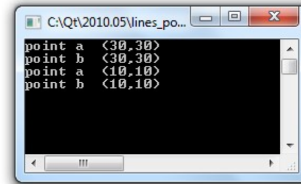
- When passing an argument to a function it is passed by value. This is the default behavior. The method receives a copy of the variable. It doesn't receive the variable itself.
- Thus, whenever we pass an object as argument to a function the compiler calls the copy constructor in order to initialize the new created object.
- The copy constructor is also called whenever a function returns an object.

(c) 2011 Haim Michael. All Rights Reserved. 53

Copy Constructors

- We can explicitly call the copy constructor. We will do so when there is a need to construct one object as an exact copy of another one.

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Line one;
    Line two(one);
    one.triple();
    one.details();
    two.details();
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Passing Objects By Reference

- Passing objects by reference is usually more efficient comparing with passing them by value. The copy constructor is not invoked.
- Marking the parameter with `const` will ensure that the object is not changed during the execution of the function. Other developers won't need to worry about that possibility.

```
...
void Line::setPointA(const Point& p);
...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Object Destruction

- When the object is destroyed the destructor method is called. The purpose of the destructor is to cleanup the memory that object was responsible for.
- Objects on the stack are automatically destroyed when the execution goes beyond their scope. In other words, whenever the code encounters an ending curly brace all objects that were created on the stack within those curly braces are destroyed.

(c) 2011 Haim Michael. All Rights Reserved. 56

Object Destruction

- Objects on the heap are not automatically destroyed. There is a need to `delete` them.

```
...
Rectangle* rec;
rec = new Rectangle(4,3);
...
delete rec;
...
```

- We will usually define the destructors for taking care of deleting objects on the heap.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

- The assignment operator gets into action when assigning one object into another. Unless we write one, C++ writes one for us.
- The default C++ assignment behavior is nearly identical to the default copy constructor behavior. Unlike the copy constructor, the assignment operator returns a reference to an object. That allows us to chain assignments with each other.

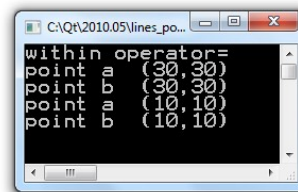
```
...
Rectangle rec;
rec = otherRec = goodRec = anotherRec;
...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Line one;
    Line two;
    two = one;
    one.triple();
    one.details();
    two.details();
    getchar();
    return 0;
}
```



The Assignment Operator

```
class Point
{
public:
    Point();
    Point(const Point &src);
    Point(double x,double y);
    double x;
    double y;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
class Line
{
public:
    Line();
    Line(const Line &src);
    Line(Point* a,Point* b);
    Line& operator=(const Line& other);
    void setPointA(Point* p);
    void setPointB(Point* p);
    Point* getPointA();
    Point* getPointB();
    void details();
    void triple();
private:
    Point* a;
    Point* b;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

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

Point::Point(const Point &src)
{
    this->x = src.x;
    this->y = src.y;
}

Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
Line::Line()
{
    a = new Point(10,10);
    b = new Point(10,10);
}

Line::Line(Point* a, Point* b)
{
    this->setPointA(a);
    this->setPointB(b);
}

Line::Line(const Line &src)
{
    a = new Point(src.a->x,src.b->y);
    b = new Point(src.a->x,src.b->y);
}

void Line::setPointA(Point* p)
{
    a = p;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

```
void Line::setPointB(Point* p)
{
    b = p;
}

Point* Line::getPointA()
{
    return a;
}

Point* Line::getPointB()
{
    return b;
}

void Line::details()
{
    std::cout << "point a  (" << a->x << "," << a->y << ")"
        << std::endl;
    std::cout << "point b  (" << b->x << "," << b->y << ")"
        << std::endl;
}
```

(c) 2011 Haim Michael. All Rights Reserved. 64

The Assignment Operator

```
void Line::triple()
{
    a->x*=3;
    a->y*=3;
    b->x*=3;
    b->y*=3;
}

Line& Line::operator=(const Line& other)
{
    std::cout << "within operator=" << std::endl;
    if(this==&other)
    {
        return (*this);
    }
    a = new Point(other.a->x,other.a->y);
    b = new Point(other.b->x,other.b->y);
    return (*this);
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The Assignment Operator

- The = operator does not always mean assignment. When placed on the same line where the variable is declared it functions as a shorthand for the copy constructor.

```
...
Rec rec = otherRec;
...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Dynamic Memory Allocation

- When we don't know how much memory will be needed before the code actually runs we can dynamically allocate the require memory during the execution itself.
- When our object dynamically allocates the required memory we should pay attention to the copy constructor, assignment operator and the destructor defined in its class.

(c) 2011 Haim Michael. All Rights Reserved. 67

Freeing Memory with Destructors

- The destructor is executed when the object reaches the end of its life.
- The destructor has the same name as the name of the class preceeded by ~ (tilde).
- The destructor doesn't take any parameter and each class can include the definition for one destructor only.
- In general, the destructor frees the memory that was allocated in the constructor.

(c) 2011 Haim Michael. All Rights Reserved. 68

Freeing Memory with Destructors

```
class Line
{
public:
    Line();
    Line(const Line &src);
    Line(Point* a,Point* b);
    ~Line();
    Line& operator=(const Line& other);
    static Line getLine();
    void setPointA(Point* p);
    void setPointB(Point* p);
    Point* getPointA();
    Point* getPointB();
    void details();
    void triple();
private:
    Point* a;
    Point* b;
};
```

(c) 2011 Haim Michael. All Rights Reserved. 69

Freeing Memory with Destructors

```
...

Line::Line(const Line &src)
{
    std::cout << "within copy constructor" << std::endl;
    a = new Point(src.a->x,src.b->y);
    b = new Point(src.a->x,src.b->y);
}

Line::~Line()
{
    delete a;
    delete b;
}

...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Disabling Pass By Value

- We can disable passing by value and disable assignment by marking the copy constructor and the `operator=` definitions with the `private` access specifier.
- Doing so, it won't be possible to compile code that that tries to pass over the object by value, return it from a function or assign to it.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Static Data Members

- Static data member is a data member associated with a class instead of object.

```
...
class Rectangle
{
   public:
      static int sCounter;
      double width;
      double height;
}
...
```

(c) 2011 Haim Michael. All Rights Reserved. 72

Static Data Members

- In order to access a static member we need to prefix it with the name of the class togeher with the `::` operator.

```
...
std::cout << Rectangle::sCounter;
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Const Data Members

- We can declare our members together with the `const` modifier. That will turn them into constants.
- The `const` data members are usually also static ones. Usually, it doesn't make sense to keep the same value in all objects.

```
...
class Car
{
    public:
        ...
        static const double maxSpeed = 180;
}
...
```

(c) 2011 Haim Michael. All Rights Reserved. 74

Static Methods

- We can define static methods. Static methods don't apply specifically to each object. Static methods apply to the class as a whole.

```
...
class Rectangle
{
   public:
      Rectangle static unite(Rectangle a, Rectangle b);
      ...
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Static Methods

- Calling a static method is done similarly to the way in which we access static members.

```
...
Rectangle recA(4,3);
Rectangle recB(5,2);
Rectangle rec = Rectangle::unite(recA,recB);
...
```

- Static methods cannot access non static members.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Const Methods

- We can mark a method with `const` in order to ensure that it doesn't change any data member.

```
...
class Rectangle
{
   public:
      double area() const;
      ...
}
...
```

(c) 2011 Haim Michael. All Rights Reserved.

Const Methods

- Objects marked with `const` can call `const` methods only. Others can call all sorts of methods. Both the ones marked with `const` and the ones that arenot.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Functions Overloading

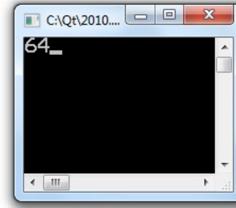
- The C++ programming language supports overloading. We can overload any function, method and constructor as many times as we want as long as the number and/or the types of the parameters differ.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Functions Overloading

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle rec(8);
    std::cout << rec.area();
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Functions Overloading

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    double area();
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Functions Overloading

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved. 82

Functions Overloading

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}

double Rectangle::area()
{
    return width*height;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

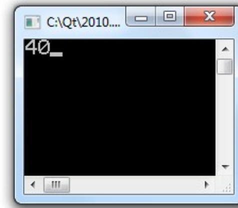
Default Parameters

- We can specify default values for function and method parameters. If the user won't specify arguments for those parameters the defaults will be used.
- Setting defaults we can do it for a continuous list of parameters starting from the rightmost parameter only.

(c) 2011 Haim Michael. All Rights Reserved.

Default Parameters

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle a(4);
    std::cout << a.area();
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Default Parameters

```
class Rectangle
{
public:
    Rectangle(double w=10,double h=10);
    double area() {return width*height;}
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Default Parameters

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved. 87

Inline Methods

- The C++ programming language allows us to define inline methods. Inline methods are methods the compiler insert their body directly into the code instead of each and every call for their execusion.
- The process involved with inline methods is just a simpler version for using the `#define` macro.
- We can specify an inline method or a function by placing the `inline` keyword infront of its name in the source code file where we define it.

(c) 2011 Haim Michael. All Rights Reserved. 88

Inline Methods

- We can alternatively include the implementation of the method within the header file and taking it out from the source code file. Doing so will have the same result.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Inline Methods

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle* rec = new Rectangle(20,5);
    double total = rec->area();
    std::cout << "area is " << total << std::endl;
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Inline Methods

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    double area() {return width*height;}
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved. 91

Inline Methods

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Inline Methods

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved. 93

Inline Methods

- When we define an inline method the compiler doesn't want to inline it might silently ignore our directive.
- Inline methods can lead to code bloat. We should use inline method and functions in a careful way.

(c) 2011 Haim Michael. All Rights Reserved. 94

Nested Classes

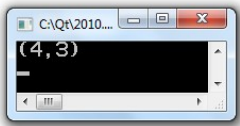
- The class definition can include more than just methods and members. The class definition can include inner classes, inner structs and inner enums.
- Using the inner class (or the inner struct... or the inner enum) outside the scope of the outer class depends on their access specifiers.
- Referring an inner type should be done using its full qualified name.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Nested Classes

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle* rec = new Rectangle(20,5);
    rec->setLocation(Rectangle::Point(4,3));
    rec->getLocation().details();
    getchar();
    return 0;
}
```



(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Nested Classes

```
class Rectangle
{
public:
    class Point
    {
    public:
        Point();
        Point(double x, double y);
        void details();
        double x;
        double y;
    };
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    double area() {return width*height;}
    void setLocation(Point point);
    Point getLocation();
private:
    double width;
    double height;
    Point location;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Nested Classes

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Nested Classes

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}

Rectangle::Point Rectangle::getLocation()
{
    return location;
}

void Rectangle::setLocation(Point point)
{
    location = point;
}
```

(c) 2011 Haim Michael. All Rights Reserved. 99

Nested Classes

```
void Rectangle::Point::details()
{
    std::cout << "(" << x << "," << y << ")" <<
std::endl;
}

Rectangle::Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}

Rectangle::Point::Point()
{
    this->x = 10;
    this->x = 10;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The `typedef` Keyword

- We can use the `typedef` keyword to create an alias for a data type.

```
typedef originalname newname;
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The `typedef` Keyword

```
class Rectangle
{
public:
    class Point
    {
    public:
        Point();
        Point(double x, double y);
        void details();
        double x;
        double y;
    };
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    double area() {return width*height;}
    void setLocation(Point point);
    Point getLocation();
private:
    double width;
    double height;
    Point location;
};
```

(c) 2011 Haim Michael. All Rights Reserved. 102

The `typedef` Keyword

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The `typedef` Keyword

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}

Rectangle::Point Rectangle::getLocation()
{
    return location;
}

void Rectangle::setLocation(Point point)
{
    location = point;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

The `typedef` Keyword

```
typedef Rectangle::Point RecPoint;

//void Rectangle::Point::details()
void RecPoint::details()
{
    std::cout << "(" << x << "," << y << ")" << std::endl;
}

//Rectangle::Point::Point(double x, double y)
RecPoint::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}

//Rectangle::Point::Point()
RecPoint::Point()
{
    this->x = 10;
    this->x = 10;
}
```

(c) 2011 Haim Michael. All Rights Reserved. 105

The `typedef` Keyword

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle* rec = new Rectangle(20,5);
    rec->setLocation(Rectangle::Point(4,3));
    rec->getLocation().details();
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Friends

- The C++ programming language allows us to declare that other classeses or nonmember functions are friends. Becoming friends they can access private and protected members and methods.

```
...
Class Car
{
    public:
        friend class Person;
        friend bool checkEngine(const Car& car);
}
...
```

The Person class is a friend of Car

We can declare a global function within our class together with the word friend. That will be sufficient both for declaring about the function and for making it a friend that can access Private and protected members of the class Car.

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

- The C++ programming language allows us to overload the operators we know with a specific definition for our class type.

```
...
Class Rectangle
{
    public:
        Rectangle();
        Rectangle operator+(Rectangle& other);
        double area();
    private:
        double width;
        double height;
}
...
```

(c) 2011 Haim Michael. All Rights Reserved. 108

Operators Overloading

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle a(2,3);
    Rectangle b(3,4);
    Rectangle c;
    c = a + b;
    std::cout << c.area();
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    Rectangle operator+(Rectangle& other);
    double area() {return width*height;}
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}

Rectangle Rectangle::operator+(Rectangle& other)
{
    return Rectangle(
       this->width+other.width,
       this->height+other.height);
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

- We can alternatively overload the operator by declaring a global function.

```
#include "stdio.h"
#include "geometry.h"
#include <iostream>

int main(int argc, char *argv[])
{
    Rectangle a(2,3);
    Rectangle b(3,4);
    Rectangle c;
    c = a + b;
    std::cout << c.area();
    getchar();
    return 0;
}
```

(c) 2011 Haim Michael. All Rights Reserved. 113

Operators Overloading

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    //Rectangle operator+(Rectangle& other);
    double area() {return width*height;}
    friend Rectangle operator+(Rectangle& a,Rectangle& b);
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved. 114

Operators Overloading

```
#include <iostream>
#include "stdio.h"
#include "geometry.h"

Rectangle::Rectangle()
{
    width = 10;
    height = 10;
}

Rectangle::Rectangle(double size)
{
    if(size>0)
    {
        width = size;
        height = size;
    }
}
```

(c) 2011 Haim Michael. All Rights Reserved. 115

Operators Overloading

```
Rectangle::Rectangle(double w, double h)
{
    if(w>0)
    {
        width = w;
    }
    if(h>0)
    {
        height = h;
    }
}

Rectangle operator+(Rectangle& a, Rectangle& b)
{
    return Rectangle(a.width+b.width,a.height+b.height);
}
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(double size);
    Rectangle(double w,double h);
    //Rectangle operator+(Rectangle& other);
    double area() {return width*height;}
    friend Rectangle operator+(Rectangle& a,Rectangle& b);
private:
    double width;
    double height;
};
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.

Operators Overloading

- We can overload all other operators by defining similar functions.

```
Rectangle operator+(Rectangle& a, Rectangle& b)
Rectangle operator-(Rectangle& a, Rectangle& b)
Rectangle operator*(Rectangle& a, Rectangle& b)
Rectangle operator\(Rectangle& a, Rectangle& b)
Rectangle operator+=(Rectangle& a, Rectangle& b)
Rectangle operator-=(Rectangle& a, Rectangle& b)
Rectangle operator*=(Rectangle& a, Rectangle& b)
Rectangle operator\=(Rectangle& a, Rectangle& b)
bool operator==(Rectangle& a, Rectangle& b)
bool operator<(Rectangle& a, Rectangle& b)
bool operator>(Rectangle& a, Rectangle& b)
bool operator!=(Rectangle& a, Rectangle& b)

...
```

(c) 2011 Haim Michael. All Rights Reserved.

(c) 2011 Haim Michael. All Rights Reserved.