More C++ :
Vectors, Classes, Inheritance, Templates

with content from cplusplus.com, codeguru.com

Vectors

- vectors in C++
  - basically arrays with enhancements
    - indexed similarly
    - contiguous memory
  - some changes
    - defined differently
    - can be resized without explicit memory allocation
    - contains methods, such as size()

Vectors

- using vectors
  - must include **<vector>**
  - template, so must be instantiated with type
  - qualified with **std::**

```
std::vector<int> v;    // declares a vector of integers
```

  - can be simplified in small projects

```
#include <vector>
using namespace std;
//...
vector<int> v;         // no need to prepend std:: any more
```

C++ Standard Arrays vs. Vectors

```
size_t size = 10;
int sarray[10];
int *darray = new int[size];
// do something with them:
for(int i=0; i<10; ++i){
    sarray[i] = i;
    darray[i] = i;
}
// don't forget to delete darray when you're done
delete [] darray;
```

```
#include <vector>
//...
size_t size = 10;
std::vector<int> array(size);    // make room for 10 integers,
                                 // and initialize them to 0
// do something with them:
for(int i=0; i<size; ++i){
    array[i] = i;
}
// no need to delete anything
```

Vector Length

- previous program does not check for valid index, which enhances performance
- using **`at`** function will check index

```
    std::vector<int> array;
    try{
        array.at(1000) = 0;
    }
5.  catch(std::out_of_range o){
        std::cout<<o.what()<<std::endl;
    }
```

Vector Length

- vectors can grow
  - certain amount of space allocated initially
  - once that space runs out, new space is allocated and the values are copied over

```
#include <vector>
#include <iostream>
//...
std::vector<char> array;
char c = 0;
while(c != 'x'){
  std::cin>>c;
  array.push_back(c);
}
```

Vector Size

- use **`pushback(el)`** to grow the size dynamically
- use **`resize`** to set or reset the size of the array

Vector Size

- use the **`size()`** method for loops

```
for (i = 0; i < array.size(); i++)
   array[i] = 0;
```

Classes

- classes
  - fancy struct's
  - expanded concept of data structures
    - data
    - methods (functions)
  - object
    - instantiation of a class
    - type/variable ⇔ class/object
  - defined with keyword **`class`** (or **`struct`**)

Classes

- members are listed under access specifiers
  - **`private`**
    - members accessible only from within the class
  - **`protected`**
    - members accessible to class or derived classes
  - **`public`**
    - members accessible anywhere the object is visible
- by default, access is private

Classes

- example

```
1 class Rectangle {
2     int width, height;
3   public:
4     void set_values (int,int);
5     int area (void);
6 } rect;
```

  - declares a class, **`Rectangle`**
  - declares an object, **`rect`**
  - class contains 4 members
    - 2 private data
    - 2 public methods (declarations only, not definitions)

Classes

- members are accessed through objects

```
1 rect.set_values (3,4);
2 myarea = rect.area();
```

- public methods can be accessed directly using . operator
  - similar to struct's

Classes

– example

```
1  // classes example
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      void set_values (int,int);
9      int area() {return width*height;}
10 };
11
12 void Rectangle::set_values (int x, int y) {
13   width = x;
14   height = y;
15 }
16
17 int main () {
18   Rectangle rect;
19   rect.set_values (3,4);
20   cout << "area: " << rect.area();
21   return 0;
22 }
```

notes:

- declaration vs. definition
- inline function
- encapsulation
- data hiding

– output

```
area: 12
```

Classes

- example with 2 variables

```
1  // example: one class, two objects
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      void set_values (int,int);
9      int area () {return width*height;}
10 };
11
12 void Rectangle::set_values (int x, int y) {
13   width = x;
14   height = y;
15 }
16
17 int main () {
18   Rectangle rect, rectb;
19   rect.set_values (3,4);
20   rectb.set_values (5,6);
21   cout << "rect area: " << rect.area() << endl;
22   cout << "rectb area: " << rectb.area() << endl;
23   return 0;
24 }
```

notes:

each object has its own set of data/methods

no parameters needed for call to area

- output

```
rect area: 12
rectb area: 30
```

Classes

- what would happen if we called **`area`** before setting values?
  - undetermined result
- constructors
  - automatically called when a new object is created
  - initializes values, allocates memory, etc.
  - constructor name same as class name
  - no return type
  - cannot be called explicitly

Classes

- example

```
1  // example: class constructor
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      Rectangle (int,int);
9      int area () {return (width*height);}
10 };
11
12 Rectangle::Rectangle (int a, int b) {
13   width = a;
14   height = b;
15 }
16
17 int main () {
18   Rectangle rect (3,4);
19   Rectangle rectb (5,6);
20   cout << "rect area: " << rect.area() << endl;
21   cout << "rectb area: " << rectb.area() << endl;
22   return 0;
23 }
```

notes:
- results same as before
- set_values omitted
- values passed to constructor

- output

```
rect area: 12
rectb area: 30
```

Classes

- constructors can be overloaded
  - different number of parameters
  - different parameter types
- implicit default constructor defined if no other constructor defined
  - takes no parameters
  - called when object is declared but no parameters are passed to the constructor
  - cannot call default constructor with parentheses
    - represents a function declaration

```
1 Rectangle rectb;   // ok, default constructor called
2 Rectangle rectc(); // oops, default constructor NOT called
```

Classes

- member initialization
  - can be done in constructor body or member initialization

```
1 class Rectangle {
2     int width,height;
3   public:
4     Rectangle(int,int);
5     int area() {return width*height;}
6 };
```

- constructor can be defined normally

```
Rectangle::Rectangle (int x, int y) { width=x; height=y; }
```

- or with member initialization

```
Rectangle::Rectangle (int x, int y) : width(x) { height=y; }
```

```
Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
```

Classes

- for simple types, doesn't matter if initialization is defined or by default
- for member objects (whose type is a class)
  - if not initialized after the colon, they are default-constructed
  - default construction may not be possible if no default constructor defined for class
  - use member initialization list instead

Classes

- example

```
1  // member initialization
2  #include <iostream>
3  using namespace std;
4
5  class Circle {
6      double radius;
7    public:
8      Circle(double r) : radius(r) { }
9      double area() {return radius*radius*3.14159265;}
10 };
11
12 class Cylinder {
13     Circle base;
14     double height;
15   public:
16     Cylinder(double r, double h) : base (r), height(h) {}
17     double volume() {return base.area() * height;}
18 };
19
20 int main () {
21   Cylinder foo (10,20);
22
23   cout << "foo's volume: " << foo.volume() << '\n';
24   return 0;
25 }
```

Cylinder class has member of type class Circle and needs to call Circle constructor in member initialization list

Classes

- operator overloading
  - allows operators, such as + or *, to be defined for user-defined types
  - defined like member functions, but prepended with keyword **`operator`**

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

Classes

- operator overloading example

```
1  // overloading operators example
2  #include <iostream>
3  using namespace std;
4
5  class CVector {
6    public:
7      int x,y;
8      CVector () {};
9      CVector (int a,int b) : x(a), y(b) {}
10     CVector operator + (const CVector&);
11 };
12
13 CVector CVector::operator+ (const CVector& param) {
14   CVector temp;
15   temp.x = x + param.x;
16   temp.y = y + param.y;
17   return temp;
18 }
19
20 int main () {
21   CVector foo (3,1);
22   CVector bar (1,2);
23   CVector result;
24   result = foo + bar;
25   cout << result.x << ',' << result.y << '\n';
26   return 0;
27 }
```

example: equivalent

```
1 c = a + b;
2 c = a.operator+ (b);
```

Classes

- **`this`**
  - pointer to current object
  - used within a class method to refer to the object that called it

- example

```
Rectangle::Rectangle (int width, int height)  {
    this -> width = width;
    this -> height = height;
}
```

Classes

- templates
  - parameterized class

```
1 template <class T>
2 class mypair {
3     T values [2];
4   public:
5     mypair (T first, T second)
6     {
7       values[0]=first; values[1]=second;
8     }
9 };
```

  - can be used to store elements of type **`int`**

```
mypair<int> myobject (115, 36);
```

  - or type **`float`**

```
mypair<double> myfloats (3.0, 2.18);
```

Classes

- destructor
  - opposite of constructor
  - called when an object's lifetime ends
  - performs cleanup, such as memory deallocation
  - returns nothing, not even void
  - name same as class name, but preceded by ~
  - implicit default destructor provided if none defined

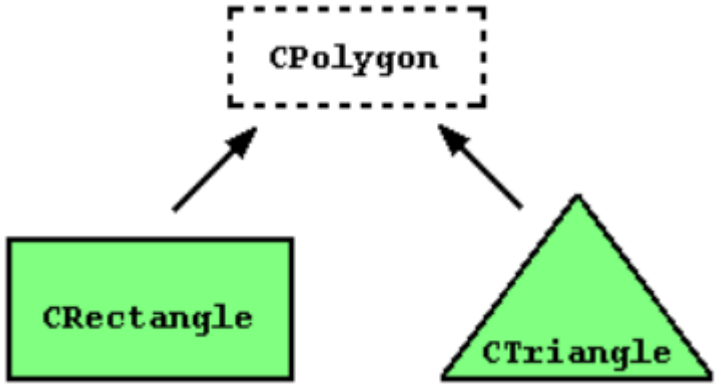
Classes

- destructor example

```
1  // destructors
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  class Example4 {
7      string* ptr;
8    public:
9      // constructors:
10     Example4() : ptr(new string) {}
11     Example4 (const string& str) : ptr(new string(str)) {}
12     // destructor:
13     ~Example4 () {delete ptr;}
14     // access content:
15     const string& content() const {return *ptr;}
16 };
17
18 int main () {
19   Example4 foo;
20   Example4 bar ("Example");
21
22   cout << "bar's content: " << bar.content() << '\n';
23   return 0;
24 }
```

Inheritance

- inheritance
  - allows classes to be extended
  - derived classes retain characteristics of the base class
  - avoids replicated code by allowing common properties to be contained in one class and then used by other classes



  - **`Polygon`** contains common members; **`Rectangle`** and **`Triangle`** access common members plus add specific features

Inheritance

- inheritance example
  - derived classes contain

    `width, height,`

    `set_values`
  - output

```
20
10
```

```
1  // derived classes
2  #include <iostream>
3  using namespace std;
4
5  class Polygon {
6    protected:
7      int width, height;
8    public:
9      void set_values (int a, int b)
10       { width=a; height=b;}
11 };
12
13 class Rectangle: public Polygon {
14   public:
15     int area ()
16       { return width * height; }
17 };
18
19 class Triangle: public Polygon {
20   public:
21     int area ()
22       { return width * height / 2; }
23   };
24
25 int main () {
26   Rectangle rect;
27   Triangle trgl;
28   rect.set_values (4,5);
29   trgl.set_values (4,5);
30   cout << rect.area() << '\n';
31   cout << trgl.area() << '\n';
32   return 0;
33 }
```

Inheritance

- inheritance
  - access types and inheritance

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

  - inherited members have same access permissions as in base class in this example

```
1 Polygon::width              // protected access
2 Rectangle::width            // protected access
3
4 Polygon::set_values()       // public access
5 Rectangle::set_values()     // public access
```

since

```
class Rectangle: public Polygon { /* ... */ }
```

Virtual Methods

- virtual methods
  - can be redefined in derived classes, while preserving its calling signature
  - declared with keyword **`virtual`**

Virtual Methods

– virtual method example

```
1  // virtual members
2  #include <iostream>
3  using namespace std;
4
5  class Polygon {
6    protected:
7      int width, height;
8    public:
9      void set_values (int a, int b)
10       { width=a; height=b; }
11     virtual int area ()
12       { return 0; }
13 };
14
15 class Rectangle: public Polygon {
16   public:
17     int area ()
18       { return width * height; }
19 };
20
21 class Triangle: public Polygon {
22   public:
23     int area ()
24       { return (width * height / 2); }
25 };
```

```
27 int main () {
28   Rectangle rect;
29   Triangle trgl;
30   Polygon poly;
31   Polygon * ppoly1 = &rect;
32   Polygon * ppoly2 = &trgl;
33   Polygon * ppoly3 = &poly;
34   ppoly1->set_values (4,5);
35   ppoly2->set_values (4,5);
36   ppoly3->set_values (4,5);
37   cout << ppoly1->area() << '\n';
38   cout << ppoly2->area() << '\n';
39   cout << ppoly3->area() << '\n';
40   return 0;
41 }
```

`area` declared virtual – derived classes will redefine it

```
20
10
0
```

Virtual Methods

- virtual methods
  - if **`virtual`** keyword removed, all derived class calls to **`area`** method through pointers to base class would return 0
  - virtual methods redefined in derived classes
    - non-virtual methods can also be redefined in derived classes
    - but, if virtual, a pointer to the base class can access the redefined virtual method in the derived class
  - a class that declares or inherits a virtual function is polymorphic
  - note that **`Poly`** is a class, too, and objects can be declared with it

Virtual Methods

- abstract base class
  - similar to base class in previous example
  - can only be used as base class
  - can have virtual methods without definition
    - pure virtual function
    - appended with **=0**

Classes

- abstract base class

```
1 // abstract class CPolygon
2 class Polygon {
3   protected:
4     int width, height;
5   public:
6     void set_values (int a, int b)
7       { width=a; height=b; }
8     virtual int area () =0;
9 };
```

  - cannot be used to declare objects

```
Polygon mypolygon;   // not working if Polygon is abstract base class
```

  - can be used to create pointers to it and take advantage of polymorphic features

```
1 Polygon * ppoly1;
2 Polygon * ppoly2;
```

Inheritance

- abstract base class example

```
1  // abstract base class
2  #include <iostream>
3  using namespace std;
4
5  class Polygon {
6    protected:
7      int width, height;
8    public:
9      void set_values (int a, int b)
10       { width=a; height=b; }
11     virtual int area (void) =0;
12 };
13
14 class Rectangle: public Polygon {
15   public:
16     int area (void)
17       { return (width * height); }
18 };
19
20 class Triangle: public Polygon {
21   public:
22     int area (void)
23       { return (width * height / 2); }
24 };
```

```
26 int main () {
27   Rectangle rect;
28   Triangle trgl;
29   Polygon * ppoly1 = &rect;
30   Polygon * ppoly2 = &trgl;
31   ppoly1->set_values (4,5);
32   ppoly2->set_values (4,5);
33   cout << ppoly1->area() << '\n';
34   cout << ppoly2->area() << '\n';
35   return 0;
36 }
```

```
20
10
```