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

with content from cplusplus.com, codeguru.com

1

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()

2

2

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
```

3

3

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
```

4

4

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;
}
catch(std::out_of_range o){
    std::cout<<o.what()<<std::endl;
}
```

5

5

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);
}
```

6

6

Vector Size

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

7

7

Vector Size

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

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

8

8

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`**)

9

9

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

10

10

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)

11

11

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

12

12

Classes

- example

```
// classes example
#include <iostream>
using namespace std;

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

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

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

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

- output

```
area: 12
```

13

13

Classes

- example with 2 variables

```
// example: one class, two objects
#include <iostream>
using namespace std;

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

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

int main () {
  Rectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

notes:
- each object has its own set of data/methods
- no parameters needed for call to area

- output

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

14

14

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

15

15

Classes

- example

```
// example: class constructor
#include <iostream>
using namespace std;

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

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

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

- output

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

16

16

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

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

17

17

Classes

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

```
class Rectangle {
    int width,height;
  public:
    Rectangle(int,int);
    int area() {return width*height;}
};
```

- 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) { }
```

18

18

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

19

19

Classes

- example

```
// member initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) : radius(r) { }
    double area() {return radius*radius*3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
  public:
    Cylinder(double r, double h) : base (r), height(h) {}
    double volume() {return base.area() * height;}
};

int main () {
  Cylinder foo (10,20);

  cout << "foo's volume: " << foo.volume() << '\n';
  return 0;
}
```

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

20

20

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[]								

21

21

Classes

- operator overloading example

```
// overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int a,int b) : x(a), y(b) {}
    CVector operator + (const CVector&);
};

CVector CVector::operator+ (const CVector& param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return temp;
}

int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}
```

example: equivalent

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

22

22

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;
}
```

23

23

Classes

- templates
  - parameterized class

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

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

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

  - or type **`float`**

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

24

24

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

25

25

Classes

- destructor example

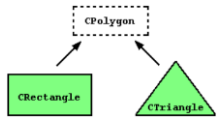
```
// destructors
#include <iostream>
#include <string>
using namespace std;

class Example4 {
    string* ptr;
  public:
    // constructors:
    Example4() : ptr(new string) {}
    Example4 (const string& str) : ptr(new string(str)) {}
    // destructor:
    ~Example4 () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example4 foo;
  Example4 bar ("Example");

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

26

26

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

27

27

Inheritance

- inheritance example
  - derived classes contain **`width, height, set_values`**
  - output

```
20
10
```

```
// derived classes
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

28

28

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

```
Polygon::width           // protected access
Rectangle::width         // protected access

Polygon::set_values()    // public access
Rectangle::set_values()  // public access
```

since

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

29

29

Virtual Methods

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

30

30

Virtual Methods

- virtual method example

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

area declared virtual – derived classes will redefine it

```
20
10
0
```

31

31

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

32

32

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

33

33

Classes

- abstract base class

```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

  - 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

```
Polygon * ppoly1;
Polygon * ppoly2;
```

34

34

Inheritance

- abstract base class example

```
// abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

```
20
10
```

35

35