Chapter 9

Object-Oriented programming

9.1 Intro

The "object approach", which is the fundamental idea in the conception of `C++` programs, consists in building the programs as an interaction between objects :

1. For all part of the program that use a given object, it is defined by the **methods** you can use on it ;
2. you can take an existing object and add data inside and methods to manipulate it, this is call inheritance.

The gains of such an approach are :

1. Modularity : each object has a clear semantic (`Employer` or `DrawingDevice`), a clear set of methods (`getSalary(), getAge()`, or `drawLine(), drawCircle()` ;
2. Less bugs : the data are accessed through the methods and you can use them only the way to object's creator wants you to :
3. Re-use : you can extend an existing object, or you can build a new one which could be use in place of the first one, as long as it has all the methods required (for example the `Employer` could be either the CEO or a worker, both of them having the required methods but different data associated to them. `DrawingDevice` could either be a window, a printer, or anything else).

9.2 Vocabulary

- A **class** is the definition of a data structure and the associated operations that can be done on it ;
- an **object** (equivalent to a variable) is an **instanciation** of the class, i.e. an existing set of data build upon the model described by the class ;
- a **data field** is one of the variable internal to the object containing a piece of data ;
- a **method** is a special function associated to a class.

9.3 Protected fields

Some of the data fields of a class can be hidden. By default, they are, and it's why we have used the `public` keyword in preceding examples. You can specify explicitly some fields to be "hidden" with the `private` keywords :

```
class Yeah {
  int a;
public:
  int b;
  double x;
private:
  double z;
};

int main(int argc, char **argv) {
  Yeah y;
  y.a = 5;
  y.b = 3;
  y.x = 2.3;
  y.z = 10.0;
}
```

```
/tmp/chose.cc: In function ‘int main(int, char **)’:
/tmp/chose.cc:2: ‘int Yeah::a’ is private
/tmp/chose.cc:12: within this context
/tmp/chose.cc:7: ‘double Yeah::z’ is private
/tmp/chose.cc:15: within this context
```

9.4 Methods

The `class` keyword allows you to associate to the data type you create a set of **methods** with privileged access to the inner structure of the object. Those functions must be seen as the *actions* you can do on your object. They are very similar to standard functions, except that they are associated to a class and can be called only for a given object.

```
class Matrix {
  int width, height;
  double *data;
public:
  void init(int w, int h) {
    width = w; height = h;
    data = new double[width * height];
  }

  void destroy() { delete[] data; }

  double getValue(int i, int j) {
    return data[i + width*j];
  }

  void setValue(int i, int j, double x) {
    data[i + width*j] = x;
  }
};
```

9.5 Calling methods

As for fields, the syntax is either the dot-notation `.` or the arrow-notation `->` :

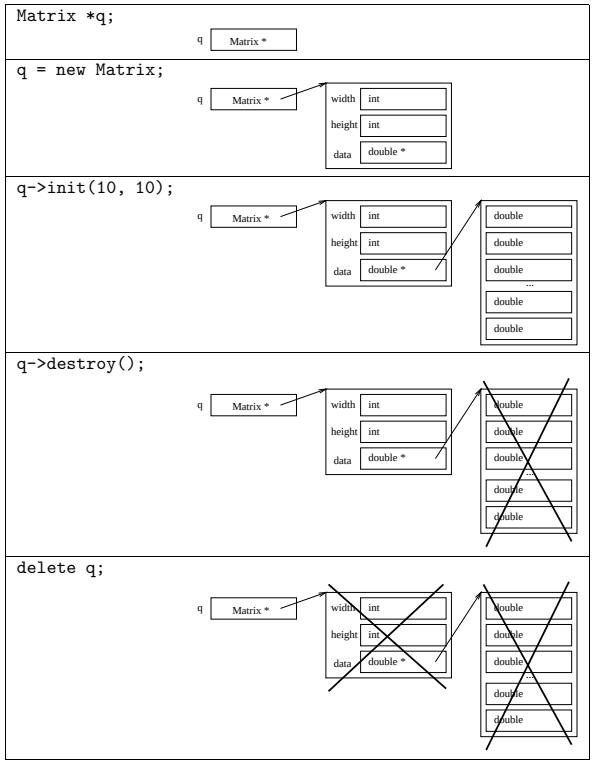
```
int main(int argc, char **argv) {
  Matrix m;
  m.init(20, 20);
  for(int i = 0; i<20; i++) m.setValue(i, i, 1.0);
  m.destroy();

  Matrix *q;
  q = new Matrix;
  q->init(10, 10);
  for(int i = 0; i<10; i++) q->setValue(i, i, 1.0);
```



```
  q->destroy(); // here we deallocate q->data but not q itself
  delete q;     // here we deallocate q itself
}
```

9.6 Some memory figures



9.7 Separating declaration and definition

We have seen that we can separate the declaration (i.e. giving the name of the function, its return type and the number and types of its parameters) and the definition (i.e. the code itself).

For methods it's the same, but we need a syntax to specify the class a function belongs to (the same name can be used for member functions of different classes). The syntax is `<class name>::<function name>`.

The methods identifier can be used alone in the member functions statement.

```
class Small {
  int x;
public:
  void setValue(int a);
};

class Bigger {
  int x, y;
public:
  void setValue(int a);
};

void Small::setValue(int a) { x = a; }
void Bigger::setValue(int a) { x = a; y = a*a; }
```

9.8 Protection of data integrity

This access through methods is very efficient to protect the integrity of data and control the out of bounds errors :

```
class Matrix {
  int width, height;
  double *data;
public:
  void init(int w, int h) {
    width = w; height = h;
    data = new double[width * height];
  }

  void destroy() { delete[] data; }
```



```
  double getValue(int i, int j) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     return data[i + width*j];
  }

  void setValue(int i, int j, double x) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     data[i + width*j] = x;
  }
};
```

9.9 Abstraction of concepts

This notion of matrix, and the associated method can also be used for a special class of matrix with only ONE non-null coefficient. This matrix would allow you to store one value at one location.

```
class MatrixAlmostNull {
  int width, height;
  int x, y;
  double v;
public:
  void init(int w, int h) { width = w; height = h; v = 0.0; }
  void destroy() { }

  double getValue(int i, int j) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     if((i == x) && (j == y)) return v; else return 0.0;
  }

  void setValue(int i, int j, double vv) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     if((v == 0.0) || ((x == i) && (y == j))) {
         x = i;
         y = j;
         v = vv;
     } else abort();
  }
};
```

9.10 Constructors

In the preceding examples, we have used each time one function to initialize the object and another one to destroy it. We know that for any object those two tasks have to be done.

The C++ syntax defines a set of special methods called **constructors**. Those methods have the same name as the class itself, and do not return results. The are called when the variable of that type is defined :

```
#include <iostream>
#include <cmath>

class NormalizedVector {
  double x, y;
public:
  NormalizedVector(double a, double b) {
    double d = sqrt(a*a + b*b);
    x = a/d;
    y = b/d;
  }
  double getX() { return x; }
  double getY() { return y; }
};

int main(int argc, char **argv) {
  NormalizedVector v(23.0, -45.0);
  cout << v.getX() << ' ' << v.getY() << '\n';
  NormalizedVector *w;
  w = new NormalizedVector(0.0, 5.0);
  cout << w->getX() << ' ' << w->getY() << '\n';
  delete w;
}
```

The same class can have many constructors :

```
#include <iostream>
#include <cmath>

class NormalizedVector {
  double x, y;
public:
  NormalizedVector(double theta) {
    x = cos(theta);
```



```
    y = sin(theta);
  }

  NormalizedVector(double a, double b) {
    double d = sqrt(a*a + b*b);
    x = a/d;
    y = b/d;
  }
  double getX() { return x; }
  double getY() { return y; }
};
```

9.11 Default constructor

A default constructor can be called with no parameters, and is used if you define a variable with no initial value.

```
class Something {
public:
  Something() {};
};

class SomethingElse {
public:
  SomethingElse(int x) {};
};

int main(int argc, char **argv) {
  Something x;
  SomethingElse y;
}
```

compilation returns

```
/tmp/chose.cc: In function `int main(int, char **)':
/tmp/chose.cc:13: no matching function for call to
                  `SomethingElse::SomethingElse ()'
/tmp/chose.cc:8: candidates are:
                   SomethingElse::SomethingElse(int)
/tmp/chose.cc:9:
                   SomethingElse::SomethingElse(const
                                        SomethingElse &)
```

9.12 Destructor

The symmetric operation is the destruction of objects. This is required as soon as the object dynamically allocates other objects.

The special method defined to do that is called the **destructor**, and is called as soon as the compiler need to deallocate an instance of the class. There is only one destructor per class, which return no value, and has no parameter. The name of the destructor is the class name prefixed with a ~.

We can now re-write our matrix class :

```
class Matrix {
  int width, height;
  double *data;
public:
  Matrix(int w, int h) {
    width = w; height = h;
    data = new double[width * height];
  }

  ~Matrix() { delete[] data; }

  double getValue(int i, int j) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     return data[i + width*j];
  }

  void setValue(int i, int j, double x) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     data[i + width*j] = x;
  }
};
```

9.13 Tracing precisely what is going on

```
#include <iostream>

class Something {
  char *name;
public:
  Something(char *n) {
    name = n; cout << "Creating " << name << '\n';
  }
  ~Something() { cout << "Destroying " << name << '\n'; }
};

int main(int argc, char **argv) {
  Something x("x"), y("y");
  Something *z = new Something("z");
  Something w("w");
  { Something v("v"); }
  delete z;
}
```

```
Creating x
Creating y
Creating z
Creating w
Creating v
Destroying v
Destroying z
Destroying w
Destroying y
Destroying x
```


9.14 The member operators

We have seen that we can define our own operators. We can also define class operators. Here we redefine the bracket operator, with one integer parameter. By returning a reference to a value, the result of the `[]` operator is a lvalue, and finally we can use those new arrays like standard arrays!

```
#include <iostream>

class OneDArray {
  int size;
  double *data;
public:
  OneDArray(int s) { size = s; data = new double[size]; }
  ~OneDArray() { delete[] data; }
  double &operator [] (int k) {
    if((k < 0) || (k >= size)) abort();
    return data[k];
  }
```

```
};

int main(int argc, char **argv) {
  OneDArray a(10);
  for(int i = 0; i<10; i++) a[i] = 1.0/i;
  for(int i = 0; i<10; i++)
    cout << "a[" << i << "] = " << a[i] << '\n';
  a[14] = 1.0;
}
```

displays :

```
a[0] = inf
a[1] = 1
a[2] = 0.5
a[3] = 0.333333
a[4] = 0.25
a[5] = 0.2
a[6] = 0.166667
a[7] = 0.142857
a[8] = 0.125
a[9] = 0.111111
Aborted
```

A simple vector class to illustrate the `+` operator redefinition. The passing by reference is just used here to increase the performances by avoiding a copy. Note that the precise meaning of the operation `v + w` is here `v.(operator +)(w)`.

The `=` operator is implicitly defined by the compiler and just copies the two field.

```
#include <iostream>

class TwoDVector {
  double x, y;
public:
  TwoVector() { x = 0; y = 0; }
  TwoDVector(double a, double b) { x = a; y = b; }
  TwoDVector operator + (TwoDVector &v) {
    return TwoDVector(x + v.x, y + v.y) ;
  }
  void print() { cout << x << ' ' << y << '\n'; }
};
```



```
int main(int argc, char **argv) {
  TwoDVector v(2, 3);
  TwoDVector w(4, 5);
  TwoDVector z;
  z = v+w;
  z.print();
}
```

displays `6 8`.

9.15 Summary for classes

Properties of a class :

- Corresponds to a data-structure, defined with several **data fields** ;
- each data field has a **type** and an **identifier** ;
- data fields can be **public** or **private** ;
- a instantiation of a class is called an **object** and is the same as a variable ;
- **methods** are functions that can be applied to an object and have privileged access to the data fields ;
- methods are called with either the `.` operator or the `->` operator if we use a pointer to an object ;
- **constructors** are special functions called when creating an instance of the class, they do not return types and have for identifier the same identifier as the class itself ;
- the **destructor** is a special method called when an object is destructed, is has no return value and has for identifier the class name prefixed by a `~` ;
- we can also define **member operators** ;
- we can define method out of the class statement by using the `<class name>::<member name>` syntax.