C++: functions and operators

ens-lal

2013

Outline

- functions
  - arguments passing
  - overloading
- operators overloading

Arguments passing

Arguments can be given to functions:

- by value: the function is given a copy of the variable from the caller so that function can not modify the value of the original variable.
  Be wary of passing arguments *by-value* for complex types (*ie* `std::string`, `std::vector`) as the copy can and will degrade the application's performances.
- by address: addresses the cons of passing arguments *by-value* but adds some syntax noise.
- by reference: addresses the cons of passing *by-value* w/o the syntax noise (corollary: a bit "magical"). For clarity, one usually also adds the const keyword if the variable isn't meant to be modified by the function.

Call by-value

```
void fct(int a)
{ a = 4; }

#include <iostream>
int main(int argc, char **argv) {
  // calling fct
  int b = 3; fct(b);

  std::cout << "b= " << b << std::endl;
  return 0;
}
```

b 's value is NOT modified after the call of `fct`:

```
$ ./by-value
b= 3
```

Call by-address

```
void fct(int *a)
{ *a = 4; }

#include <iostream>
int main(int argc, char **argv) {
  // calling fct
  int b = 3; fct(&b);

  std::cout << "b= " << b << std::endl;
  return 0;
}
```

b 's value is modified after the call of `fct`:

```
$ ./by-addr
b= 4
```

Call by-reference

```
void fct(int &a)
{ *a = 4; }

#include <iostream>
int main(int argc, char **argv) {
  // calling fct
  int b = 3; fct(b);

  std::cout << "b= " << b << std::endl;
  return 0;
}
```

b 's value is modified after the call of `fct`:

```
$ ./by-ref
b= 4
```

Function overloading

Overloading a function is performed by implementing:

- multiple function members
  - with the same name
  - within a given class
- but with different arguments

vector.h

```
class Vector {
  int m_x, m_y;
public:
  Vector() : m_x(0), m_y(0) {}
  Vector(int x, int y) : m_x(x), m_y(y) {}

  void move(int x, int y);
  void move(const Vector &v);
};
```

vector.cxx

```
void Vector::move(int x, int y)
{ m_x += x; m_y += y; }

void Vector::move(const Vector &v)
{ m_x += v.m_x; m_y += v.m_y; }
```

main.cxx

```
#include "vector.h"
int main(int argc, char **argv) {
  Vector v1(3, 5);
  Vector v2(6, 7);

  v1.move(6, 7);
  v1.move(v2);

  return 0;
}
```

Operator overloading

Allows to redefine operators for the whole class.

unary

if Δ is a unary operator, Δ n can be interpreted as: `n.operator`Δ`()`
e.g: `n++` is interpreted as: `n.operator++()`

binary

if Δ is a binary operator, nΔ m can be interpreted as: `n.operator`Δ`(m)`
e.g: `n+m` is interpreted as: `n.operator+(m)`

- most of the operators can be redefined for any given class
- special case: the compiler will generate the assignment operator (=) for you if you don't declare+implement it. It is often needed to re-define it, though.

vector.h

```
#include <iosfwd>
class Vector {
  int m_x, m_y;
public:
  Vector() : m_x(0), m_y(0) {}
  Vector(int x, int y) : m_x(x), m_y(y) {}

  bool operator==(const Vector &v);
  Vector& operator=(const Vector &v);
  Vector& operator+(const Vector &v);

  friend
  std::ostream& operator<<(std::ostream &os, const Vector &v);
};
```

vector.cxx

```
#include "vector.h"
bool
Vector::operator==(const Vector &v)
{ return (m_x == v.m_x) && (m_y == v.m_y); }

Vector& Vector::operator=(const Vector &v) {
  if (&v != this) {    // protection against self-assignment
    this->m_x = v.m_x;
    this->m_y = v.m_y;
  }
  return *this;
}

Vector&
Vector::operator+(const Vector &v)
{ return Vector(m_x + v.m_x, m_y + v.m_y); }
```

vector.cxx (cont'd)

```
#include <ostream>
#include "vector.h"

std::ostream&
operator<<(std::ostream &os, const Vector &v)
{
  os << "Abscissa= " << v.m_x << std::endl
     << "Ordinate= " << v.m_y << std::endl;
  return os;
}
```

Note: this `operator<<` function can access the private datamembers of `Vector` because it was declared a `friend` of that class in the class declaration.

main.cxx

```
#include "vector.h"
#include <iostream>
#include <fstream>
int main(int argc, char **argv) {
  Vector v1(3, 5), v2(6, 8), v3, v4;

  v3 = v1 + v2; // =operator, +operator
  v4 = v3;      // =operator
  if (v3 == v4) { // comparison op.
    std::cout << "v3 and v4 are equal" << std::endl;
  }
  std::cout << v4 << std::endl; // print v4 on screen

  std::ofstream f("file.txt");
  f << v4 << std::endl;         // print v4 in a file
  f.close();
  return 0;
}
```

Overloading Operators

+	-	*	/	%	=	!
+=	-=	*=	/=	%=	++	--
<	>	<=	>=	==	!=	&&
||	&	|	^	<<	>>	
[]	new	delete				

Questions ?

Questions ?