Operator overloading

- The operators are treated by the compiler as special functions, where the calling syntax is different of that of common functions, the operator and the parameters (operands) respecting the rules as in algebra

- The C++ language allows the overloading by the programmers the C++ language operators. These C++ operators can be redefined such that they perform some operations defined by programmers.

- The operator overloading, regardless of the way it is realized, must be associated to classes, in the sense that at least one of the parameters must be a class object (explicit or implicit by the hidden parameter *this*).

A. Defining and using overloaded operators

Example. In the case of designing a class for polynomial operations, the operation of adding two polynomials can be described by a common function, or by the overloading of the adding operator:

I.

```
class polynomial {
  double *a;
  int n;
public:
  polynomial();
  // ...
  friend polynomial sum(const polynomial& a,
                        const polynomial& b);
  // ...
};
```

```
void processing() {
  polynomial p, q, r;
  // ...
  r = sum(p, q);
  // ...
}
```

II.

```
class polinom {
  double *a;
  int n;
public:
  polinom();
  friend polinom operator+(const polinom& a,
                           const polinom& b);
  // ...
};

void prelucrare() {
  polinom p, q, r;
  // ...
  r = p + q;
}
```

- ***The definition*** of an overloaded operator represents the definition of a function, which has the name of a predefined form:

  `operator <op>`

  where `<op>` represents the operator which is overloaded.

  The call of an overloaded operator is realized respecting the predefined C++ syntax for the respective operator.

- The operator function can be defined as *friend* function of the respective class, or also as a *member* function. In the case of member functions, the number of parameters is always ***less with 1*** than the arity of the corresponding operator (the first parameter is the current object of the class, represented by the hidden parameter *this* of the function).

- So, it results a third variant for the *polynomial* class:

III.

```
class polynomial {
  double *a;
  int n;
public:
  polynomial();
  // ...
  polynomial operator+(const polynomial& a);
  // ...
};
void processing() {
  polynomial p, q, r;
  // ...
  r = p + q;
  // ...
}
```

- The *syntax generated by the compiler* for the operator call is different in the last two cases. In the case of a *friend* functions, the effective calling syntax for *p+q* is:

  ```
  operator+(p, q)
  ```

  while in the case of a *member* function, the syntax is:

  ```
  p.operator+(q)
  ```

- The great majority of the C++ language operators can be overloaded:

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

- For determining the number of arguments of the operator function it must be taken in consideration the *arity* of the original operator from the C++ language, and also the fact if the operator function is a *member* of a class, or a *friend* one.

- The next table presents the calling syntax and also the form of the effective call for unary and binary operators (*X* and *Y* denote the operands and *Op* the operator) in the case of *member* functions, and also of the *friend* functions:

Call Syntax	**Effective Call**
`Op X`	`X.operatorOp()`
`X Op`	`X.operatorOp()`
`X Op Y`	`X.operatorOp(Y)`

Call Syntax	**Effective Call**
`Op X`	`operatorOp(X)`
`X Op`	`operatorOp(X)`
`X Op Y`	`operatorOp(X,Y)`

- The use of overloaded operators in a C++ program, must respect some restrictions imposed by the uniform treatment of the compiler for these operators:

1) It is not allowed to define new operators, others than the operators presented in the last table;
2) It is not allowed to change the arity, neither the precedence of an operator;
3) The operators cannot be combined for forming new operators. For example, if the operators **+** and **=** have been defined, then the call **+=** does not mean the call of **+**, followed by the call of **=**;
4) The operators **=**, **[]**, and **()** must be non-static member functions

- The operators =, (), [], ->, and ->*, must always be defined as member functions. For all the others operators can be taken in consideration the following tips for operators' definition:
  - It is indicated as all unary operators to be defined as member functions;
  - The binary composed operators of assignment (**+=**, **-=**, **/=**, ***=**, **^=**, **&=**, **\=**, **%=**, **>>=**, **<<=**) is indicated to be defined as member functions;
  - For the others binary operators it is indicated to be defined as friend functions.

B. Unary operators

- From the set of C++ unary operators, the only two operators that can present problems are the operators for ***incrementing*** and ***decrementing***, because these can have two different forms of calling syntax: *prefix* and *postfix*.

- The result returned by the operator is the same, but its ***side effect*** is different (the evaluation value of an expression containing such operators is different in the case of the two calling forms).

Example.

```
int n = 4;
if (n++ < 6) cout << ”OK\n”;     //Eval(n++)=4
cout << n << endl;               //n=5
if (++n < 6) cout << ”KO\n”;     //Eval(++n)=6
cout << n << endl;               //n=6
```

- It results that there exists two different operators for the incrementing and decrementing operations, the compiler generating different calls for the two forms of them.

- In the case of the incrementing operator **++** defined as a friend function of a class, for the prefix form **++a**, the following call will be generated:

  ```
  operator++(a)
  ```

  while for the postfix form **a++**, will be generated the following call:

  ```
  operator++(a,int)
  ```

  The auxiliary parameter is used only for distinguish between the two operators.
- A usual example of using the incrementing and decrementing operators is the case of classes representing ***containers*** and ***iterators***.
  - A ***container*** is a collection of many objects of the same type, which allows being accessed only an object at a time, in a similar way as the lists and the vectors;
  - An ***iterator*** is always associated to a container, being responsible with the access to the current object from the container, but it does not allow the access to the container implementation

Example. Implementation of a list as a container. The unary operator **->** is used to access the current object from the container; the operator **++** is used to pass to the next element. Both operators belong to iterator class, and not to the container.

```
// the class of the elements from the container
class Node {
  int val;
  Node *next;
public:
  Node(int v, Node *p = 0): val(v), next(p) {}
  ~Node() { next = 0; }
  int Val() const { return val; }
  void Print() const { cout << val << endl; }
  friend class List;
  friend class ListIterator;
};

// the class of the container
class List {
  Node* first;
  void Copy(Node* p);
  void Delete();
```

```
public:
  List() { first = 0; }
  List(const List& l): first(0) { Copy(l.first); }
  ~List() { Delete(); first = 0; }
  void Add(int k) {     //insertion in front of the list
    Node* p = new Node(k);
    p->next = first;
    first = p;
  }
  friend class ListIterator;
};

// the class of the iterator
class ListIterator {
  List l;               // It matters the declaration order
  Node* current;        // of the two data members !!
public:
  ListIterator(List& ll): l(ll), current(l.first) {}
  Node* operator->() const { return current; }
  // prefix operator
  Node* operator++() {     // prefix operator
    current = current->next;
    return current;
  }
```

```
  Node* operator++(int) {   // postfix operator
    Node* p = current;
    current = current->next;
    return p;
  }
  // re-initialization of the iteration
  void BeginIterator() { current = l.first; }
};

int main() {
  List l;
  l.Add(3);
  l.Add(7);
  l.Add(5);
  ListIterator it(l);
  do   // Erorr!! After the display of 3, it++ becomes NULL
    it->Print();
  while(it++);
  it.BeginIterator();
  do   // Correct. It is displayed 5,7,3.
    it->Print();
  while(++it);
  return 0;
}
```

C. The assignment operator

- The ***(copy) assignment operator*** is one of the most important and most used overloaded operators, being the only operator, which can by default generated by the compiler in the case of missing its explicit definition in a class.

- The assignment operator must be defined as member function. This restriction is natural, because of the assignment operation:

  <*variable*> = <*expression*>

  The assignment operator is strong related to the <*variable*> (it is a *l-value*), which represents the first operand

- There are similarities between the *assignment operation* and the *initialization of a variable*, and thus there are similarities between the *(copy) assignment operator* and the *copy-constructor*.

Example:

```
class ComplexNo {
  double x,y;
public:
  ComplexNo(double a=0, double b=0) { x=a; y=b; }
  ComplexNo(NumarComplex& c) { x=c.x; y=c.y; }
  void operator=(ComplexNo& c) { x=c.x; y=c.y; }
  void Print() { cout << x << y << endl; }
};

int main() {
  ComplexNo z1(2, 7);
  ComplexNo z2 = z1;    // copy-constructor
  ComplexNo z3;
  z3 = z1;              // assignment operator
  z1.Print();
  z2.Print();
  z3.Print();
  return 0;
}
```

- One of the problems which may appear is related to the values returned by an assignment operator. In the case when the result type is `void`, as in the previous example, it cannot be realized a ***multiple assignment***. For example, for the assignment:

  ```
  z2 = z1 = z;
  ```

  the compiler should generate:

  ```
  z2.operator=(z1.operator=(z));
  ```

  But the sub-expression **`z1.operator=(z)`** has the type **`void`**, while the parameter of the function **`z2.operator=()`** must be also a reference to an object of the type *ComplexNo*.

- Usually the assignment operator returns an instance object of the respective class or a reference to such an object.

Example: The redefinition of the assignment operator from the last example:

```
ComplexNo& operator=(ComplexNo& c) {
  x = c.x;
  y = c.y;
  return *this;
}
```

- In the case when a class does not have its own assignment operator, the compiler will generate a *default* operator, similarly as in the case of copy-constructor: it will generate an assignment member by member of the component data.

Example:

```
class A {
public:
  A& operator=(const A&) {
    cout << "class A; operator=" << endl;
    return *this;
  }
};

class B {
  A a;
};

int main() {
  // the default constructors for A and B are generated
  B b1, b2;
  b1 = b2;   // default operator= for class B
  return 0;
}
```

The program displays the message **`Class A; operator=,`** what means that a default assignment operator for the member *a* is generated

- The observations from the copy-constructor in the case of derived classes are also valid in this case:
  - if a class is derived from one or more base classes and, in addition, it contains also member data which are instances of other classes, *the assignment operator default generated by the compiler will call, first, the assignment operators of the base classes, and then the ones from the classes to which the member objects belong, in the specification order in the derived class*.
  - if an instance of a component class, or a base class is on its turn a derived class, *the calling rule of the assignment operators is applied recursively*.

- If a class uses pointers, it is not indicated to leave the assignment operator to be automatically generated by the compiler.

Example: An assignment operator for the polynomial class previous defined:

```
class polynomial {
  double *a;
  int n;
public:
  polynomial(int k = 0) { a = new double [1+n=k]; }
  polynomial& operator=(polynomial& p);
  // ...
};

polynomial& polynomial::operator=(polynomial& p) {
  a = new double [1+n=p.n];
  // All the coefficients are copied
  for (int i=0; i<=n; i++)
    a[i] = p.a[i];
  return *this;
}
```

- In the previous implementation of the assignment operator there exists a subtle error: in the case of an assignment, the memory zone occupied by the initial polynomial coefficients will remain blocked until the end of the program execution.

```
polynomial& polynomial::operator=(polynomial& p) {
  delete[] a;
  a = new double[1+n=p.n];
  for (int i=0; i<=n; i++)
    a[i] = p.a[i];
  return *this;
}
```

- The previous assignment operator contains also a hidden error. In the case when it is used an assignment as:

```
p = p;
```

after freeing the memory for the current object, its information it is not longer available and the copy of the member data cannot be realized.

Example:

```
polynomial& polynomial::operator=(polynomial& p) {
  if (&p == this)
    return *this;
  delete[] a;
  a=new double[1+n=p.n];
  for(int i=0; i<=n; i++)
    a[i] = p.a[i];
  return *this;
}
```

- There exist two important restrictions concerning the overloading of the assignment operator, which is not applied to the most of the others operators:
  - the `operator=` function must be non-static, for assuring that the left operand of the assignment is always an object;
  - the `operator=` function cannot be inherited in a class hierarchy (as the copy-constructor), because each derived class from a hierarchy can contain also specific members, but for these members the assignment operator from the base class cannot be used.

Example:

```
class B {
  int n;
public:
  B(int k=0) { n=k; }
  B& operator=(B& b) { n=b.n; return *this; }
};

class D: public B {
  int k;
public:
  D(int a, int b):B(a), k(b) {}
  D& operator=(D& d) {
    // the additional member is copied
    k = d.k;
    // the base part is copied
    B::operator=(d);
    return *this;
  }
};
```

D. Binary operators

- Most of the overloaded operators are binary operators.
- The operators `=`, `[]`, `()`, `->`, and `->*` are necessary to be defined as *member* functions, while for the rest of the binary operators is indicated to be defined as *friend* functions.
- The advantage of using binary operators as *friend* functions consists in the possibility of *automatic conversion* of operands, unlike in the case of *member* operators, when the left operand of the operator must have a pre-defined data type (e.g. the class where the operator was defined).

Example:

```
class Number {
  int n;
public:
  Number(int k=0): n(k) {}
  const Number operator+(const Number& k) const
    { return n + k.n; }
  friend const Number operator-(const Number&, const Number&);
};

const Number operator-(const Number& n1, const Number& n2)
  { return Number (n1.n-n2.n); }

int main() {
  Number a(7), b(3);
  a+b; // OK
  a+1; // OK: the second argument is converted to Number
  1+a; // Error: the first argument must be Number
  a-b; // OK
  a-1; // OK: the second argument is converted to Number
  1-a; // OK: the first argument must be Number
  return 0;
}
```

- Because the class *Number* has a conversion constructor from *int* to *Number*, when calling an operator as:

  operator-(<arg1>, <arg2>)

  an object of *Number* type can be created starting from an integer value, both for the first argument and also for the second one. In the case of a call as:

  <arg1>.operator-(<arg2>)

  this conversion can be realized only for the second argument.

Example. It is defined the class polynomial which contains overloaded operators:

```
class polynomial {
  double *a;
  int n;
public:
  polynomial(int k) { a = new double[n=k]; }
  polynomial() { n=0; a=0; }
  polynomial(polynomial&);
  ~polynomial() { delete[] a; a=0; n=0; }
  polynomial& operator=(polynomial &);
  friend polynomial operator*(polynomial&, polynomial&);
  int operator<(polynomial& p) { return n<p.n; }
  double& operator[](int i) { return a[i]; }
```

```
  // ...
};

polynomial::polynomial(polynomial& p): n(p.n) {
  if (&p == this)
    return *this;
  delete[] a;
  a = new double[1+n=p.n];
  for (int i=0; i<=n; i++)
    a[i] = p.a[i];
  return *this;
}

polynomial operator*(polynomial &p1, polynomial &p2) {
  polynomial p(p1.n+p2.n);
  for(int k=0; k<=p.n; k++) {
    p.a[k] = 0;
    for(int i=0; i<=p1.n; i++)
      p.a[i] += p1.a[i]*p2.a[k-i];
  }
  return p;
}
```

E. Move semantics

- The C language use ***copy semantics*** for assignments and parameter passing

- Example:

```
int n = 5, m = n, p; /* a copy of n is assigned to m */
p = m;               /* a copy of m is assigned to p */
```

- Initially, the C++ language used the same copy semantics

- Example:

```
class C {
public:
   C(int k=0) {…}            // conversion constructor
   C(const C& x) {…}         // copy constructor
   C& operator=(const C& x) {…}  // operator =
   ...
};
```

```
C c1(2), c2 = c1, c3;
c3 = c2;              // a copy of c2 is assigned to c3

C func(C x) {
   /* … */
   return x;
}

c3 = func(c2);        // a copy of c2 is passed to x
                      // a copy of x is passed to c3
```

- If a program contains several assignments and several function calls, there are many temporary objects that are constructed (and then destroyed) by copy assignments or copy constructors
  - These operations may affect the execution time of the program

- In general, in programming languages there are 3 types of semantics for implementing such operations:

1. ***Share semantics*** (or ***Reference semantics***), where the ***reference*** of the first object is copied to the second one.
    - In this case both objects refer the same memory zone
    - Both objects can be used
    - A garbage collector is needed
    - Some language that implements share semantics: Java, Python

2. ***Copy semantics***.
    - In this case the two objects have distinct zones of memory
    - Both objects can be used
    - Languages: C, C++

3. ***Move semantics***, where the ***reference*** of the first object is copied to the second one.
    - The second object refers to the same zone of memory as the first one
    - The first object cannot be used
    - Languages: Rust, C++ (starting to C++11)

- The C++98 standard defines ***references*** and ***passing objects by reference***
  - In this case, the copy constructor is not called

Example:

```
void func(C& x) {
   /* … */
   return x;
}

c3 = func(c2);    // a reference of c2 is passed to x
```

- ***Limitations*** of passing by reference:
  - ***Non-const references*** can only reference ***non-const l-values***, so a reference parameter cannot accept an argument that is a ***const l-value*** or an ***r-value***

Example:

```
C createC() {
   return C();
}

void func(C& x) { /*…*/ }

func(createC());  // error: parameter is a r-value
```

- In the last case, only ***copy semantics*** can be used (before C++11)
- Starting to C++11 the C++ language have also a ***move semantics***, that allows programs to run faster
- The C++ compiler determines from the context when copy semantics have to be used, or move semantics have to be used

- The ***move semantics*** uses two elements:
  - ***r-value references***
  - ***move constructor*** and ***move operator=***

- ***References*** from C++98 become ***l-value references*** in C++11
  - ***l-value***: an address of a (semi)permanent object
  - ***r-value***: an address of a temporary object

Example:

```
int n = 7;
int& m = n; // a l-value

double x, y;
double& X1() {
   return x;
}
double X2() {
   return x;
}
```

```
X1() = 3.7; // X1: a l-value
y = X2();   // X2: a r-value
```

- Prior to C++11, a l-value can be bind to a r-value, only it is a const reference

Example:

```
const double& z = X2();   // OK
double& z = X2();         // error
```

- A “mutable” reference cannot be bind to a temporary object that can disappear

- In C++11 a ***r-value reference*** is introduced

- A “mutable” ***r-value reference*** can be bound to a ***r-value***, but not to a ***l-value***

- Syntax: **`T&&`** is a ***r-value reference*** to a temporary object of the type **`T`**
  - **`T&&`** is not a *reference to a reference* to the type **`T`** (as in the case of pointers)

Example:

```
double&& z = X2();          // OK
double x;
void printX(const double& v) {
   cout << v
}
void printX(double&& v) {
   cout << v
}
double X() {
   return x;
}
printX(x);      // the first overloaded function
printx(X());    // the second overloaded function
```

- The ***second component*** of ***move semantics*** is represented by the ***move constructor*** and the ***move operator==***

Example:

```
class X {
   int* v;
   int len;
public:
   X(int k) { v = new int[len = k]; }
   ~X() { delete[] v; }
   X(const X& o) {                  // copy constructor
      v = new int[len = o.len];
      std::memcpy(v, o.v, o.len);
   }
   X& operator=(const X& o) {     // copy operator =
      if (this == &o) return *this;
      delete[] v;
      v = new int[len = o.len];
      std::memcpy(v, o.v, o.len);
      return *this;
   }
};
```

```
X createX(int k)
   return X(k);
}

int main() {
   X o = createX(100);      // (1) copy constructor
   o = createX(50);         // (2) copy operator =
}
```

- In (1) 2 temporary objects are created that are expensive copies (similar in (2)):
  - Returning from **`createX()`**
  - In the copy-constructor

- The main idea of the copy semantics:
  - To add new versions of the copy constructor and the assignment operator so that it can take a temporary object in input for modifying data from it

- This is a problem for C++98: how to modify a temporary object?
  - Solution in C++11: by using r-values references

Example. The before class `X` rewritten with *move semantics*

```
class X {
   int* v;
   int len;
public:
   X(int k) { v = new int[len = k]; }
   ~X() { delete[] v; }
   X(const X& o) {                  // copy constructor
      v = new int[len = o.len];
      std::memcpy(v, o.v, o.len);
   }
   X(X&& o):v(o.v), len(o.len) { // move constructor
      o.v = NULL;
      o.len = 0;
   }
   X& operator=(const X& o) {    // copy operator =
      if (this == &o) return *this;
      delete[] v;
      v = new int[len = o.len];
      std::memcpy(v, o.v, o.len);
```

```
      return *this;
   }
   X& operator=(X&& o) {       // move operator =
      if (this == &o) return *this;
      delete[] v;
      v = o.v;
      len = o.len;
      o.v = NULL;
      o.len = 0;
      return *this;
   }
   int getLen() { return len; }
};

X createX(int k) { return X(k); }

int main() {
   X x1(100);     // default constructor
   X x2 = x1;     // copy constructor (l-value in input)
   X x3 = createX(50);     // move constructor
                           // (r-value in input)
```

```
    x2 = x3;           // copy operator = (l-value in input)
    x2 = createX(50); // move operator = (l-value in input)
    return 0;
}
```

- Using r-values and l-values, overloaded functions can be written to determine whether function parameters are l-values or r-values

Example:

```
void printLen(X& x) {
    cout << x.getLen() << endl;
}
void printLen(X&& x) {
    cout << x.getLen() << endl;
}
…
printLen(x2);
printLen(createX(30));
```

- In C++11 all the algorithms and containers were extended to support move semantics.

- There is an utility function **`std::move`** from the Standard Library that can be used ***to convert an l-value into an r-value***

Example:

```
X x1(100);
X x2(x1);                    // copy constructor
X x3(std::move(x2));         // move constructor
```

- **`std::move`** has converted the l-value **`x2`** into an r-value

- However, after the construction of the object **`x3`**, **`x2`** becomes a hollow object and it cannot be used after that

F. Type conversion

- The C++ language allows two ways of type conversion for data:
  - ***explicit conversion*** of types
  - ***default conversion*** of types.

Explicit conversion of types

- The explicit conversion does not involve operator overloading.

- The C++ language supports the explicit conversion of type as the C language, but it has in addition a series of ***specific operators of explicit conversion***.

- The main operators of explicit conversion are: **`static_cast, const_cast, dynamic_cast, and reinterpret_cast`**.

- For these operators, instead of the traditional syntax (the `cast` operator):
  `(<type>) <expression>`
  it is used the following syntax:
  `<operator-cast> ‘<’ <type> ‘>’ <expression>`
  where **`<operator-cast>`** can be one of the operators previous mentioned

- The ***`static_cast` operator*** is the main operator for explicit conversion and it is used, in general, for well defined conversion, for which can be used the cast operator of the C language.

Example.

```
//A. Conversions for an appropriate type
int a, b;
double x = static_cast<double>(a)/b;
//B. Pointer conversion from the void* type
double *p = static_cast<double*>malloc(sizeof(double));
//C. Conversion of implicit type, which are done by the compiler
void f(double z);
int k = 3;
f(static_cast<double>(k));
```

- The ***const_cast operator*** is used in the case of **const** and **volatile** modifiers. Noting with *T* the data type, this operator allows the conversion from the type *const T*, or *volatile T* to the *T* type, or to a derived type from *T* (*T** for example).

Example.

```
class A {
  // ...
};
class B: public A {
  // ...
};
void g(B *pb);
B b;
const B& cb = b;

// Error! Must be B*, not const B*
g(&cb);

// OK.
g(const_cast<B*>(&cb));
```

```
// OK. Same thing, but using the C style
g((B*)&cb);
A *pa = new A;

// Error! it must be B*, not A*
g(pa);

// Error! const_cast cannot be used
// It is a downcasting conversion
g(const_cast<B*>(pa));
```

- The ***dynamic_cast operator*** is used only in the case of class hierarchies, for realizing the conversion from a class situated at the top of the class hierarchy to a class situated at the bottom of it. The operation is called, usually, ***downcasting*** and will be discussed in the chapter related to the polymorphism.

Remark. The operator **`dynamic_cast`** can be used only in the case of class hierarchies that use the ***polymorphism*** (based on the ***virtual functions***), because it uses information from runtime.

Exemple.

```
class A {
  // It does not contain virtual functions
  // ...
};
class B: public A { /* ... */ };
class A1 {
public:
  virtual ~A1() {}
  // ...
};
class B1: public A1 { /* ... */ };
// ...
A1* pa1 = new B1;   // Upcasting
// OK. Downcasting
B1* pb1 = dynamic_cast<B1*>(pa1);
int a, b;
```

```
// Error! it does not exists inheritance
double x = dynamic_cast<double>(a)/b;
A* pa = new B;       // Upcasting
// Error. It does not exists virtual functions
B* pb = dynamic_cast<B*>(pa);
```

- The ***operator reinterpret_cast*** is used in the cases when an object is seen as a structure of bits and it is desired to be interpreted as an object having a completely different structure. Usually, the result of the conversion is dependent of implementation, what means that this type of conversion it is not in general, portable.

- A usual use of the operator ***reinterpret_cast*** consists in conversion between different types of pointers.

Example. It is considered an array of pointers to functions:

```
typedef void (*FuncPtr)();
FuncPtr funcPtrArray[10];
```

In the case when it is desired to use a pointer to a function with another prototype, for example:

```
int Func();
```

one can be proceed as follows:

```
// Error! Different types
funcPtrArray[0] = &Func;
// OK.
funcPtrArray[0] = reinterpret_cast<FuncPtr>(&Func);
```

- A special attention is imposed when using this kind of conversion type, because the compiler does not perform any verification of types and this thing can generate errors.

- For correct use of the operator **`reinterpret_cast`** it is imposed a re-conversion to the initial type of the converted type.

Example.

```
#include <iostream>
using namespace std;
const int sz = 100;
struct X {
  int a[sz];
};
void print(X* x) {
  for(int i=0; i<sz; i++)
    cout << x->a[i] << ' ';
  cout << endl;
}
```

```
int main() {
  X x;
  print(&x);
  int* xp = reinterpret_cast<int*>(&x);
  for(int* i = xp; i < xp + sz; i++)
    *i = 0;
  // It can not be used xp as X*
  // if it is not converted back
  print(reinterpret_cast<X*>(xp));
  // The x identifier can be used without conversion
  print(&x);
  return 0;
}
```

Automatic conversion of types

- An example of automatic conversion of data types in the C language is the expression evaluation. For example, in the case of the following function definition:

```
void f(double x) { cout << x << endl; }
f(5);
```

- In the case of C++ language it is possible to do an automatic conversion also for the data types defined by the user. There are two possibilities of conversion:
  - using ***conversion constructs***,
  - using ***conversion operators***.

- **First method**. Using *conversion constructors*, an automatic conversion can be realized, *from the type of the parameter of the constructor to the type of the class where the constructor is defined*.

Example:

```
class A {
  int n;
public:
  // conversion constructor from int to the type A
  A(int k = 0) { n = k; }
  void Print() { cout << n << endl; }
};

void f(A a) {
  a.Print();
}

int main () {
  A a(10);
  f(a);  // the type conversion is not performed
  f(5);  // it is performed the conversion int -> A
  return 0;
}
```

- There are also cases when it is not desired to perform an automate conversion of types. In these situations can the type conversion can be prevent by putting the keyword ***explicit*** before the respective constructor.

Example:

```
class B1 {
public:
  double x;
  B1(double z = 0.0) { x = z; }
};

class B2 {
public:
  double y;
  B2(double z): y(z) { }
  // performs the automatic conversion B1 -> B2
  B2(const B1& b): y(b.x) { }
  void Print() { cout << n << endl; }
};

class B3 {
public:
```

```
  double k;
  B3(double z = 0.0) { k = z; }
   // the conversion B1 -> B3 is not allowed
  explicit B3(const B1& b): y(b.x) { }
  void Print() { cout << n << endl; }
};

void f(B3 b) { b.Print(); }

void g(B2 b) { b.Print(); }

int main () {
  B1 b1;
  B2 b2;
  B3 b3;
  g(b1);  // OK! It exist a conversion operator B1 -> B2
  f(b2);  // Error! Does not exist the conversion B2 -> B3
  f(b1);  // Error! It is not allowed the conversion B1 -> B3
  f(b3);  // OK! It is not needed the conversion
  f(B3(b1)); // OK! The conversion is explicitly
  return 0;
}
```

- **The second method** of automatic type conversion uses a *conversion operator*. This is an operator with a special syntax which allows the conversion from the type of the class where the operator is defined to a data type desired by the programmer.

- The prototype of such an operator is (the operator's name is the name of the destination class of the conversion):

```
operator <class_name> ()
```

Example.

```
class C1 {
      int n;
   public:
     // conversion constructor
     C1(int k): n(k) {}
     void Print() { cout << n << endl; }
   };
```

```
class C2 {
  int m;
public:
  // conversion constructor
  C2(int k): m(k) {}
  // conversion operator
  operator C1() const { return C1(m); }
  void Print() { cout << m << endl; }
}

void h(C1 c) {
  c.Print();
}

int main {
  C2 c;
  C1 c1;
  h(c);   // the conversion operator is called
  h(1);   // the conversion constructor is called
  h(c1);
  return 0;
}
```

- The *difference* between the two methods is that in the case of the *conversion constructor*, the conversion operation between the source type and the destination type *is performed by the objects of destination type*, while in the case of the *conversion operator* the *objects of source type* are responsible to this conversion.

- Usually, it is allowed only one automatic conversion type between the two data types. In the case when would be allowed more types of conversions between the same types (both the conversion operator and the conversion constructor), a confusion may occur at the selection of the conversion type.

Example.

```
class B;

class A {
public:
  operator B() const ; // conversion A -> B
};
```

```
class B {
public:
  B(A);   // conversion A -> B
}

void f(B) { }

int main {
  A a;
  f(a);   // Error!! Call ambiguity
  return 0;
}
```

- An *error* which can appear is referred strictly at the *conversion operator*. In a certain class it is allowed *only one type conversion* to another class by using the conversion operator; otherwise, confusion might appear at the selection of the desired conversion operator.

Example.

```
class A;
class B;

class C {
public:
  operator A() const ;  // conversion C -> A
  operator B() const ;  // conversion C -> B
};

void f(A);

void f(B);      // overloaded function

int main {
  C c;
  f(c);         // Error!!
  return 0;
}
```

G. Smart pointers. Overloading the operator ->

- ***Smart pointers*** are objects which behave like pointers but do more than a pointer. These objects are flexible as pointers and have the advantage of being an object.

- A smart pointer is designed to handle the problems caused by using normal pointers

Example.

```
class Person {
  int age;
  char* pName;
public:
  Person(): pName(0),age(0) {}
  Person(char* name, int age): pName(name), age(age)
   { }
  ~Person() { }
  void Display() { cout << pName << age; }
};
```

```
int main() {
  Person* pPerson  = new Person("Scott", 25);
  pPerson->Display();
  delete pPerson;
  delete pPerson;    // Error !!
  return 0;
}
```

- Since the smart pointer should behave like a pointer, it should support the same interface as pointers do; i.e., they should support the following operations: two possibilities of conversion:
  - Dereferencing (**`operator *`**),
  - Indirection (**`operator ->`**).

Example. A ***smart pointer*** class related to the class *Person*.

```
class SP {
private:
    Person*    pData; // pointer to the Person class
public:
```

```
    SP(Person* pValue) : pData(pValue) { }
    ~SP() { delete pData; }
    Person& operator* () { return *pData; }
    Person* operator-> () { return pData; }
};
```

- The main responsibility of the *SP* class is to hold a pointer to the *Person* class and then delete it when its destructor is called. It should also support the interface of the pointer.

- One problem is that we can use this smart pointer class for a pointer of the *Person* class only. This problem can be solved by making use of *templates* (smart pointer class generic)

Example.

```
template <class T>
class TSP {
private:
   T*    pData;
public:
```

```
    TSP(T* pValue) : pData(pValue) { }
    ~TSP() { delete pData; }
    T& operator* () { return *pData; }
    T* operator-> () { return pData; }
};

int main () {
    TSP<Person> p(new Person("Scott", 25));
    p->Display();
    {
        TSP<PERSON> q = p;
        q->Display();
        // Destructor of q will be called here
    }
    p->Display();
    return 0;
}
```

G1. Implementing a garbage collector mechanism in C++

- One important application of smart pointers is ***memory leaks***. One can develop a simple ***garbage colector*** by using smart pointers (C++ does not has support for garbage collecting):

1. ***First step***. Developing a *reference counting class*.

```
template <class T>
class RefCount {
protected:
  int refVal;
  T* p;
public:
  RefCount(){ refVal=0; p=(T*)this;}
  void AddRef() { refVal++; }
  void ReleaseRef() {
    refVal--;
    if(refVal == 0)
      delete [] this;
```

```
  }
  T* operator->(void) { return p; }
};
```

2. ***Second step***. Develop a template class using smart pointers which encapsulates all garbage collection processes.

```
template <class T>
class gcPtr {
  T* ptr;
  char c;
public:
  // called for pointer declaration
  // without memory allocation (gcPtr<X> a;)
  gcPtr() {
    c='0';
  }
  // called for pointer with memory allocation
  // gcPtr<X> a = new X;
  gcPtr(T* ptrIn) {
    ptr=ptrIn;
```

```
  ptr->AddRef();
  c='1';
}
// called for arrays x[]
operator T*(void) { return ptr; }
// called for pointers *x
T& operator*(void) { return *ptr; }
// called for selection from pointer x->
T* operator->(void){ return ptr; }
// called for assignments x=y
gcPtr& operator=(gcPtr<T> &pIn) {
  return operator=((T *) pIn);
}
gcPtr& operator=(T* pIn) {
  if(c=='1') {   // variables have memory allocation
    // Decrease refcount for left hand side operand
    ptr->ReleaseRef();
    ptr = pIn;
   // Increase refcount for right hand operand
    pIn->AddRef();
    return *this;
  } else {       // variables do not have memory allocation
    ptr=pIn;
```

```
            ptr->AddRef();
            return *this;
          }
        }
        ~gcPtr(void) { ptr->ReleaseRef(); }
};
```

3. ***Third step***. Derive the desired class from *RefCount*, passing the name of the class to *RefCount* as template parameter.

```
class X: public RefCount<X> {
    public:
    void f() { std::cout << "f"; };
};
```

Remark. The class `X` is a *derived class* and in the same time a *type argument of the base class*, `RefCount`. This pattern is known as **Curiously Recurring Template Pattern** (CRTP). This is no recursion, because the methods of the base class will be instantiated when they are called. CRTP is an often-used

idiom in C++ to implement ***static polymorphism*** (will be discussed in a later chapter).

4. ***Fourth step.*** Declare a pointer to the desired class, use *gcPtr<T>* class, passing the name of the class as template parameter.

```
gcPtr<X> o1;
o1 = new X;
o1->f();
gcPtr<X> o2=new X;
o2->f();
```

G2. Smart pointers in modern C++

- C++ does not have a *garbage collector* mechanism (like Java, or Python)

- The main advantage of garbage collection is its simplicity, because destructors no longer have to be written and explicitly called by programmers

- There are however two main drawbacks:
  - It is performed at runtime, when a lot of time is used cleaning up the all garbage
  - In Java, or Python, destructors are present (provided by the compiler/interpreter), but they are not called when an object goes out of scope, which can be dangerous when the destructor have to release a critical resource like a lock

- Instead, C ++ has an important idiom, called **Resource Acquisition Is Initialization** (RAII), when a resource should be acquired in the constructor of the object and released in the destructor of the object

- Using RAII, C++ has a kind of garbage collection:
  - By use of smart pointers explicitly
  - By the memory management, which is designed to no overhead in performance or memory compared to a raw pointer

- ***Smart pointers*** are standard template classes, similar cu the above *TSP* class, which allow to use objects, and not pointers, in a similar way
  - The main advantage is the fact that for these objects, the constructors and destructors are automatically called (objects are stored in the stack zone, not in the heap)
  - They are declared in the ***`memory`*** header

1. The first smart pointer, called ***`auto_ptr`***, was present in the classical C++90
   - It is based on exclusive ownership model, i.e. two pointers of the same type cannot point to the same resource at the same time

- It is deprecated in C++11, because the assignment operator and the copy constructor transfers ownership and resets the ***rvalue*** auto pointer to a null pointer (in C++ there is not *move semantics*)
- **Example**:

```
#include <iostream>
#include <memory>

class A {
public:
  void print() { std::cout << "\n"; }
};

int main() {
  std::auto_ptr<A> p1(new A);
  p1->print();       // Class A
  // the address of p1
  std::cout << p1.get() << std::endl;
  // copy constructor -> p1 will be empty.
  std::auto_ptr<A> p2(p1);
  p2->print();   // Class A
  // address 00000000 - p1 is empty
  std::cout << p1.get() << std::endl;
```

```
    // p2 has the address of p1
    std::cout << p2.get() << std::endl;
    return 0;
}
```

2. In C++11, ***auto_ptr*** was replaced by ***unique_ptr***
   - Allows exactly one owner of the underlying pointer
   - Can be instantiated with and without resource
   - Offers the interface of the underlying resource
   - Can be moved to a new owner, but not copied or shared (using *move semantic*)
   - Its size is one pointer and it supports **rvalue** references for fast insertion and retrieval from C++ Standard Library collections
   - Starting to C++14, it can be created with the helper function ***make_unique***
   - **Example**:

```
#include <iostream>
#include <memory>
#include <utility>
```

```
struct Int {
  int i;
  Int(int i_):i(i_){}
  ~Int(){ std::cout << "Destructor: " << i << std::endl; }
};

int main() {
  std::unique_ptr<Int> uPtr1{ new Int(2020) };
  std::cout << uPtr1.get() << std::endl;
  std::unique_ptr<Int> uPtr2;
  uPtr2= std::move(uPtr1);
  std::cout << uPtr1.get() << std::endl;
  std::cout << uPtr2.get() << std::endl;
  {
    std::unique_ptr<Int> localPtr{ new Int(2030) };
  }
  uPtr2.reset(new Int(2021));
  Int* pi = uPtr2.release();
  delete pi;
  std::unique_ptr<Int> uPtr3{ new Int(2031) };
  std::unique_ptr<Int> uPtr4{ new Int(2032) };
  std::swap(uPtr3, uPtr4);
  std::cout << uPtr3.get() << std::endl;
  std::cout << uPtr4.get() << std::endl;
```

```
  auto uPtr5 = std::make_unique<Int>(2026);
  return 0;
}
```

3. ***shared_ptr*** is a reference-counted smart pointer, used to assign one raw pointer to multiple owners
   - The raw pointer is not deleted until all ***shared_ptr*** owners have gone out of scope or have otherwise given up ownership
   - The size is two pointers; one for the object and one for the shared control block that contains the reference count
   - **Example**:

```
#include <iostream>
#include <memory>

struct Int {
  int i;
  Int(int i_):i(i_){}
  ~Int(){ std::cout << "Destructor: " << i << std::endl; }
};

int main() {
```

```
    std::shared_ptr<Int> shPtr(new Int(2020));
    std::cout<< shPtr->i << std::endl;
    std::cout<< shPtr.use_count() << std::endl;
    {
      std::shared_ptr<Int> localPtr(shPtr);
      std::cout << locSharPtr.use_count() << std::endl;
    }
    std::cout <<  shPtr.use_count() << std::endl;
    std::shared_ptr<Int> globalPtr= shPtr;
    std::cout <<  shPtr.use_count() << std::endl;
    globalPtr.reset();
    std::cout <<  shPtr.use_count() << std::endl;
    shPtr = std::shared_ptr<Int>(new Int(2021));
    auto shPtr1 = std::make_shared<Int>(2030);
    std::cout<< shPtr1->i << std::endl;
    return 0;
}
```

4. ***weak_ptr*** is a special-case smart pointer, required in some cases to break circular references between ***shared_ptr*** instances
    - It provides access to an object that is owned by one or more ***shared_ptr*** instances but does not participate in reference counting

- **Example**:

```
#include <iostream>
#include <memory>

class B;

class A {
  std::weak_ptr<B> sP1; // used to avoid circular dependency
public:
  A() { std::cout << "A()" << std::endl; }
  ~A() { std::cout << "~A()" << std::endl; }
  void setShared(std::shared_ptr<B>& p) { sP1 = p; }
};

class B {
  std::shared_ptr<A> sP1;
public:
  B() { std::cout << "B()" << std::endl; }
  ~B() { std::cout << "~B()" << std::endl; }
  void setShared(std::shared_ptr<A>& p)  { sP1 = p; }
};

int main() {
  std::shared_ptr<A> aPtr(new A);
```

```
  std::shared_ptr<B> bPtr(new B);
  aPtr->setShared(bPtr);
  bPtr->setShared(aPtr);
  return 0;
}
```

H. Functors (Function objects). Overloading the operator ()

- A ***function object*** extends the characteristics of regular functions by using object-oriented C++. Thus, they can be referred to as ***smart functions*** that provide several advantages over regular functions:
  - Function objects can have member functions as well as member attributes. Thus, they can carry a state that even can be different at the same time,
  - Due to their nature, function objects can be initialized before their usage.
  - Function objects are usually faster *than regular functions.*

- To create an object that behaves just like a function, it is neccessary to provide a way to call this object by name using parentheses and (optionally) passing arguments.

- The solution to the problem is the ***function call operator*** **`()`.** For example:

```
class function_object {
public:
   // ...
   int operator()(int i) { return i; }
   // ...
};
int main() {
   function_object fo;
   cout << fo(5) << endl;
}
```

- Classification of Function Objects:

1. ***Stateless*** function objects are the closest correspondent to a regular function. The function object doesn't have data members, or, if it does, they have no impact whatsoever on the function call operator.

```
class stateless_function_object {
public:
   int operator()(int i) const { return i * i; }
};
int main() {
   stateless_function_object f1;
   cout << f1(2) << endl;     // will output 4
   return 0;
}
```

2. ***Invariable stateful*** function objects do have a state, but the function call operator is declared constant. Semantically, this means that the operation will use the state, but won't change it.

```
class invariant_function_object {
public:
   invariant_function_object(int state):
      state_(state) {}
   int operator()(int i) const
      { return i * state_; }
```

```
private:
   int state_;
};
int main() {
   invariant_function_object f2(3);
   cout << f2(2) << endl;    // will output 6
   return 0;
}
```

3. ***Variable stateful*** functions objects not only have a state, but also can change this state with each operation

```
class stateful_function_object {
public:
   stateful_function_object(): state_(0) {}
   int operator()(int i) { return state_ += i; }
   int get() const { return state_; }
private:
   int state_;
};
```

```
int main() {
    int v[] = {1, 2, 3, 4, 5};
    stateful_function_object f3;
    for (int k = 0; k < 4; k++)
     f3(v[k]);
    // will output 15 (1 + 2 + 3 + 4 + 5)
    cout << f3.get() << endl;
    return 0;
}
```

I. Pointers to members. Overloading the operator `->*`

- ***Pointers to members*** allow to refer to non-static members of class objects. A pointer to member cannot be used to point to a static class member because the address of a static member is not associated with any particular object. (in this case one can use a normal pointer to function).

- A class can have two general categories of members: functions and data members. Similarly, there are two categories of pointers to members: ***pointers to member functions***, and ***pointers to data members***

- The syntax for ***pointers to data members*** resembles the form of ordinary pointers to functions, with the addition of the class name followed by the operator `::`

Example.

```
class A {
public:
  int num;
  int x;
  int func ();
};

int A::*pmi; // pmi is a pointer to an int member of A
pmi = &A::num;
```

- Manipulating a data member through an object

```
A a1;
A a2;
int n = a1.*pmi;      // copy the value of a1.num to n
a1.*pmi = 5;          // assign the value to a1.num
a2.*pmi = 6;          // assign the value 6 to a2.num
```

- Manipulating a data member through an object's pointer

```
A * pa = new A;
int n = pa->*pmi;     // assign to n the value of pa->num
pa->*pmi = 5;         // assign the value 5 to pa->num
```

- A ***pointer to a member function*** consists of the member function's return type, the class name followed by `::`, the pointer's name, and the function's parameter list

```
class A {
public:
  int num;
  int x;
  int func ();
};

int (A::*pmf) ();
// pmf is a pointer to some member function of class A
// that returns int and takes no arguments
```

- Invokation of the member function to which *pmf* points:

```
pmf = &A::func;    //assign pmf
A a;
A *pa = &a;
(a.*pmf)();        // invoke a.func()
// call through a pointer to an object
(pa->*pmf)();      // calls pa->func()
```

- To reduce complex syntax, you can declare a **`typedef`** to be a pointer to a member. For example:

```
typedef int A::*my_pointer_to_member;
typedef int (A::*my_pointer_to_function) ();

my_pointer_to_member pdptr = &A::x;
my_pointer_to_function pfptr = &A::func;

A a;
a.*pdptr = 10;
cout << a.*pdptr << endl;
(a.*pfptr) ();
```

- Because the precedence of **`()`** (function call operator) is higher than **`.*`** and **`->*`**, parentheses must used to call the function pointed to by pointer to function.

- Just like operator **`->`**, the pointer-to-member dereference operator **`->*`** is generally used with some kind of object that represents a *smart pointer*.

- When defining, the operator **`->*`** must return an object for which the **`operator()`** can be called.

- Before to create an operator **`->*`**, first a class with an **`operator()`** have to be created. The operator **`->*`** will return an object of that class.
    - This class must retrieve the necessary information so that when the **`operator()`** is called, the pointer-to-member will be dereferenced for the current object

- In the following example, the constructor of the class **`FuncionObject`** stores both the pointer to the object and the pointer to the member function.
    - the **`operator()`** uses those to make the actual pointer-to-member call

Example:

```
class C {
public:
    int f1(int i) const {
        cout << "f1";
        return i;
    }
    int f2(int i) const {
        cout << "f2";
        return i;
    }
    int f3(int i) const {
        cout << "f3";
        return i;
    }
};

typedef int (C::*PMF)(int) const;

class FunctionObject {
    C* pobj;
    PMF pf;
```

```
public:
    // Save the object pointer and member pointer
    FunctionObject(C* wp, PMF pmf): pobj(wp), pf(pmf) {
        cout << "FunctionObject constructor\n";
    }
    int operator()(int i) const {
        return (pobj->*pf)(i); // Make the call
    }
};

class SPC {
    C* pc;
public:
    SPC(C* p) : pc(p) { }
    ~SPC() { delete pc; }
    C& operator* () { return *pc; }
    C* operator-> () { return pc; }
    // operator->* must return an object
    // that has an operator():
    FunctionObject operator->*(PMF pmf) {
        return FunctionObject(pc, pmf);
    }
};
```

```
int main() {
    char ch;
    C *p = new C();
    SPC w(p);
    PMF pmf = &C::f1;
    cout << (w->*pmf)(1) << endl;
    pmf = &C::f2;
    cout << (w->*pmf)(2) << endl;
    pmf = &C::f3;
    cout << (w->*pmf)(3) << endl;
    cin >> ch;
    return 0;
}
```

- The **operator->*** creates and returns a **FunctionObject**

- When **operator->*** is called, the compiler immediately calls **operator()** for the return value of **operator->***, passing to the **operator()** the arguments of the **operator->***

- The **`FunctionObject::operator()`** takes these arguments and then dereferences the “real” pointer-to-member using its stored object pointer and pointer-to-member

Overloading operators in Python

- In Python operator overlaoding has a has a broader meaning:
  - Classes can overload all Python ***expression operators***
  - In addition, classes can oveload ***built-in operations*** such as printing, function calls, attribute access, etc

- Overloading is implemented by using ***special methods***: class functions that begins and ends with double underscore __

A. String representation

- There are two special methods for displaying the content of objetcs:
  - **`__str__`** , which is used in **`print`** operation
  - **`__repr__`**, which is used in used in all other contexts

- **Example**:

```
class Point:
   def __init__(self, x = 0, y = 0):
      self.x = x
      self.y = y
   def __str__(self):
      return "({0},{1})".format(self.x,self.y)
   def __repr__(self):
      return "[{0},{1}]".format(self.x,self.y)
p = Point(1,2)
print(p)      # (1,2)
p             # [1,2]
```

B. Overloading algebraic operators

- Common algebraic operators

Operator	*Expression*	*Meaning*
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`
Negative	`-p`	`p.__neg__()`
Positive	`+p`	`p.__pos__()`

Example. A class defining vectors of the form $\overline{v} = x\overline{i} + y\overline{j}$

```
class Vector:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
    def __add__(self, v):
        x = self.x + v.x
        y = self.y + v.y
        return Vector(x,y)
    def __sub__(self, v):
        x = self.x - v.x
        y = self.y - v.y
        return Vector(x,y)
    def __mul__(self, v):
        z = self.x * v.x + self.y * v.y
        return z

v1 = Vector(1,3)
v2 = Vector(2,8)
```

```
print(v1+v2)          # (3,11)
z = v1*v2
print(z)              # 26
```

- Binary operators can use another form of special methods in the case when the first operand is not compatible with the class containing the overloaded operator: **`__radd__`**, **`__rsub__`**, **`__rmul__`**

- In these cases the two operands are reversed

Example:

```
class Number:
    def __init__(self, a = 0):
        self.val = a
    def __str__(self):
        return "{0}".format(self.val)
    def __add__(self, v):
        print('add')
        return self.val + v
    def __radd__(self, v):
        print('radd')
        return v + self.val
x = Number(5)
print(x+1)          # add 6
print(1+x)          # radd 6
```

C. Comparison operators

- Comparison operators in Python are similar to comparison operators of the C language

Operator	*Expression*	*Meaning*
Less than	`p1 < p2`	`p1.__lt__(p2)`
Less than or equal to	`p1 <= p2`	`p1.__le__(p2)`
Equal to	`p1 == p2`	`p1.__eq__(p2)`
Not equal to	`p1 != p2`	`p1.__ne__(p2)`
Greater than	`p1 > p2`	`p1.__gt__(p2)`
Greater than or equal to	`p1 >= p2`	`p1.__ge__(p2)`

Example:

```
class Vector:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
    def __lt__(self, v):
        self_mag = (self.x ** 2) + (self.y ** 2)
        v_mag = (v.x ** 2) + (v.y ** 2)
        return self_mag < v_mag
print(Vector(1,3) < Vector(2,5))      # True
print(Vector(3,6) < Vector(2,5))      # False
```

D. Bitwise operators

- Bitwise operators in Python are similar to bitwise operators of the C language

Operator	*Expression*	*Meaning*
Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Example. A class that implements a circular list.

```
class CList:
  def __init__(self, elems = []):
    self.elem = elems
  def __str__(self):
    return "({0})".format(self.elem)
  def __add__(self, l):
    return CList(self.elem + l.elem)
  def __lshift__(self, k):
    n = len(self.elem)
    return CList(self.elem[k:] + self.elem[:k])
  def __rshift__(self, k):
    n = len(self.elem)
    return CList(self.elem[n-k:] + self.elem[:n-k])
l1 = CList([2,3,4])
l2 = CList([11,12,13])
l = l1+l2
print(l)            # ([2, 3, 4, 11, 12, 13])
print(l << 2)       # ([4, 11, 12, 13, 2, 3])
print(l >> 2)       # ([12, 13, 2, 3, 4, 11])
```

D. Augmented assignment operators

- In Python, augmented assignment operators are similar to the compound assignment operators of the C language and they have the same meaning.

- The corresponing operation is perfomed „*in-place*”, when the left-hand side object supports it, and that the right-hand side is only evaluated once.

Operator	***Expression***	***Meaning***
In-place addition	`p += v`	`p.__iadd__(v)`
In-place subtraction	`p -= v`	`p.__isub__(v)`
In-place multiplication	`p *= v`	`p.__imul__(v)`
In-place power	`p /= v`	`p.__itruediv__(v)`
In-place modulo	`p %= v`	`p.__imod__(v)`
In-place power	`p **= v`	`p.__ipow__(v)`
In-place Left Shift	`p <<= v`	`p.__ilshift__(v)`
In-place Right Shift	`p >>= v`	`p.__irshift__(v)`
In-place AND	`p &= v`	`p.__iand__(v)`

In-place OR	`p |= v`	`p.__ior__(v)`
In-place XOR	`p ^= v`	`p.__ixor__(v)`

- The expression `x += y` tries to call `x.__iadd__(y)`. If `__iadd__` is not present, `x.__add__(y)` is attempted.

Example.

```
class Number:
    def __init__(self, a = 0):
        self.val = a
    def __str__(self):
        return "({0})".format(self.val)
    def __iadd__(self, v):
        return Number(self.val + v)
x = Number(7)
x += 2
print(x)    # (9)
```

E. Indexing and slicing

- ***Indexing*** an ***slicing*** are two useful operations performed on sequence types (strings, tuples, lists, ...)
  - An ***indexing*** oepration returns an element of the sequence
  - A ***slicing*** operation returns a slice (a group) of elements

Example:

```
v = [2, 4, 6, 8, 10, 12, 14, 16]
print(v[1])         # 4
print(v[2:6:2])     # [6, 10]
```

- Python has two special methods for overlaoding indexing and slicing operation:
  - **`__getitem__`** for retrieving elements
  - **`__setitem__`** for modifying elements

- **`__getitem__`** and **`__setitem__`** are called automaticallyu both for indexing operations and for slice expressions, in function of the second parameter (which can be an index or a slice)

Example:

```
class CList:
  def __init__(self, elems = []):
    self.elem = elems
  def __str__(self):
    return "({0})".format(self.elem)
  def __lshift__(self, k):
    n = len(self.elem)
    return CList(self.elem[k:] + self.elem[:k])
  def __getitem__(self, index):
    print('getitem: ', index)
    return self.elem[index]     # indexing or slicing
  def __setitem__(self, index, value):
    print('setitem: ', index, value)
    self.elem[index] = value    # assign index or slice
```

```
l = CList([2, 4, 6, 8, 10, 12, 14])
print(l[2])
# getitem: 2
# 6
print(l[2:6])
# getitem: slice(2, 6, None)
# [6, 8, 10, 12]
l[2] = 5
# setitem: 2 5
print(l)
# ([2, 4, 5, 8, 10, 12, 14])
l[3:] = [7, 9]
# setitem: slice(3, None, None) 7
print(l)
# ([2, 4, 5, 7, 9])
```

F. Iterable objects

- All iteration contexts in Python use two special methods:
  - **`__iter__`**, which returns an iterator object
  - **`__next__`**, which produces items (until a StopIteration exception is raised)

- An object is called **iterable** if there exists an iterator for it. Most of built-in containers in Python are iterables.

- The **`iter()`** function (which in turn calls the **`__iter__()`** method) returns an iterator of the iterable

- The **`next()`** function to manually iterate through all the items of an iterator

- When the end of the iterable has been reached and there is no more data to be returned, it will raise **`StopIteration`**

Example: For the class `Clist`:

```
class CList:
def __init__(self, elems = []):
   self.elem = elems
def __str__(self):
   return "({0})".format(self.elem)
def __lshift__(self, k):
   n = len(self.elem)
   return CList(self.elem[k:] + self.elem[:k])
def __iter__(self):
   self.current = -1
   return self
def __next__(self):
   if self.current == len(self.elem)-1:
      raise StopItyeration
   else:
      self.current += 1
      return self.elem[self.current]
l = CList([2, 4, 6, 8, 10, 12, 14])
it = iter(l)
```

```
print(next(it))
# 2
print(next(it))
# 4
```

- A more elegant way of automatically iterating is by using the **`for`** loop
  - The **`for`** loop can iterate over any iterable.

- In Python, the **`for`** loop is implemented as following

```
for element in iterable:
    # do something with element
```

is actually implemented as:

```
# create an iterator object from that iterable
iter_obj = iter(iterable)

# infinite loop
while True:
    try:
        # get the next item
        element = next(iter_obj)
        # do something with element
```

```
        except StopIteration:
            # if StopIteration is raised, break from loop
            break
```

- So internally, the for loop creates an iterator object, **`iter_obj`** by calling **`iter()`** on the iterable

Example. For the **`CList`** class:

```
l = CList([2, 4, 6])
for item in l:
    print(item, end=' ')
# 2 4 6
```

- It is not necessary that the item in an iterator object has to stop. There can be infinite iterators. We must be careful when handling such iterator

Example.

```
class CList:
    def __init__(self, elems = []):
        self.elem = elems
```

```
    def __str__(self):
        return "({0})".format(self.elem)
    def __lshift__(self, k):
        n = len(self.elem)
        return CList(self.elem[k:] + self.elem[:k])
    def __iter__(self):
        self.current = -1
        return self
    def __next__(self):
        if self.current == len(self.elem) - 1:
            self.current = 0
        else:
            self.current += 1
        return self.elem[self.current]

l = CList([2, 4, 6])
i = 0
it = iter(l)
while i<10:
    elem = next(it)
    print(elem, end = ' ')
    i += 1
# 2 4 6 2 4 6 2 4 6 2
```

G. Callable objects

- Python has the **`__call__`** special method for function call expressions applied to class instances
- All objects of a class having a **`__call__`** special method represent callable objects because their behaviour is similar to a function call
- The arguments of the **`__call__`** special method are the same as in the case of functions

Example.

```
Class Cuboid
    def __init__(self, a=1, b=1):
        self.a = a
        self.b = b
    def __call__(self, c=1)
        return a * b * c
```

```
cb = Cuboid(3,5)
vol1 = cb(2)
print(vol1)    # 30
vol2 = cb(10)
print(vol2)    # 150
```

H. Assignment operator

- In Python the ***assignment*** operator can not be overloaded

- Pyhon uses ***share semantics***, or ***reference semantics*** for assignment and parameter passing