9 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. In the case of the C / C++ operators the rules of calling these operators follow the syntactic rules of the C / C++ language.

The C language also has overloaded operators which perform different operations. For example, the operator '/' represents both the real division operation (in the case when at least one operand is real), and also the remainder of the division of two integer numbers (in the case when all the operands are integer).

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*).

Remark. The operator overloading is not related with the inheritance relation, so it does not generate polymorphic objects.

9.1 Generalities. Defining and using overloaded operators

The advantage of using overloaded operators in classes is because it simplifies the writing and the reading of an application, due to the syntactic form of calling these operators.

Example 9.1. 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. Using a function

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

II. Using a friend overloaded operator

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

It is observed the fact that the definition of an overloaded operator represents, practically, 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.

Because an overloaded operator is an associated function to a certain class, 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, because the first parameter is implicit in this case (it 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. Using an overloaded operator, member of the class

```
class polynomial {
  double *a;
  int n;
public:
  polynomial();
  // ...
  polynomial operator+(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, as it can be seen in the next table:

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

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; this fact is important for determining the number of arguments of the operator function.

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 can not 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 necessity of the last restriction will be presented in the next paragraph.

As seen before, the operators can be defined as member functions and also as friend functions of a certain class. An exception is made for the operators **=**, **()**, **[]**, **->**, and **->***, which must always

be 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.

9.2 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: *prefixed* and *postfixed*. 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”;     //Evaluation(n++)≡4
cout << n << endl;               //n=5
if (++n < 6) cout << ”KO\n”;     //Evaluation(++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.

For example, in the case of the incrementing operator **++** defined as a friend function of a class, for the prefixed form **++a**, the following call will be generated:

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

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

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

The auxiliary parameter is used only for distinguish between the two operators.

An 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 9.2. It will be implemented a list as a container. The access to the current object from the container is done by using the unary operator **->**, and the passing to the next element is done by using the operator **++**. 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; }
  // prefixed operator
  Node* operator++() {
    current = current->next;
    return current;
  }
  // postfixed operator
  Node* operator++(int) {
    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);
  // Erorr!! After the display of 5, it++ becomes NULL
  do
     it->Print();
```

```
  while(it++);
  it.BeginIterator();
  // Correct. It is displayed 3,7,5.
  do
     it->Print();
  while(++it);
  return 0;
}
```

9.3 The assignment operator

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

Moreover, 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 in consequence there are similarities between the assignment operator and the copy-constructor.

Example 9.3. Defining and using a class representing the complex numbers.

```
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;
}
```

The main difference between the assignment and initialization is the fact that in the case of initialization, in addition to the initialization of the member values from another object there is allocated a zone of memory for the respective object.

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 9.4.

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

class B {
  A a;
};

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

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

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.

For example, for the class:

```
class D: public B2, B1 {
  M1 m1;
  M2 m2;
  // ...
};
```

a sequence of the form:

```
class D d1, d2;
// ...
d2 = d1;
```

calls the assignment operators in the following classes' order: `B2,B1,M1,M2`.

In the case when an instance of a component class (from a composed class), or a base class (related to the inheritance relation) 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, because in this case will copied only the pointer value (object's address), and not the its referred object:

Example9.5. An assignment operator for the *polynomial* class previous defined:

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

polynomial& polynomial::operator=(polynomial& p) {
  a = new double [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. Usually it must be tested if such a situation appears and, eventually, to free the memory zone associated to the previous coefficients:

```
polynomial& polynomial::operator=(polynomial& p) {
  delete[] a;
  a = new double[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. For this reason, the case when it is tried the assignment of a variable to itself, must be treated separately.

Example:

```
polynomial& polynomial::operator=(polynomial& p) {
  if (&p == this)
    return *this;
  a=new double[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 9.6.

```
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) {
    k = d.k;               // the additional member is copied
    B::operator=(d);       // the base part is copied
    return *this;
  }
};
```

It is observed that in the previous example the assignment operator of the base class was used, to which it has been passed an object from the derived class, what is correct.

9.4 Binary operators

Most of the overloaded operators are binary, because the only ternary operator can not be overloaded, and there are few unary operators.

The possibility of defining binary operators as member functions or friend functions is different in function of their type, as said before. 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 the corresponding data type (e.g. the class where the operator was defined).

Example 9.7.

```
class Number {
  int n;
public:
  Number(int k=0): n(k) {}
  const Number operator+(const Number& k) const
    { return k + n.k; }
  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 9.8. 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[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;
}
```

9.5 Type conversion

The C language allows two ways of type conversion for data: an explicit conversion of type (the ***cast*** operator), and also an implicit conversion. The same conversion of types exists also in C++.

The explicit conversion does not suppose operator overloading, but a set of predefined operators of the C++ language.

A. Explicit conversion of types

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 advantage of using these specific operators is that each of them treats a certain category of type 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. The others conversion operators are specialized operators, which can be used in some particular cases.

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 normally
//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
update((B*)&cb);
A *pa = new A;
// Error! it must be B*, not A*
g(pa);
// Error! const_cast can not 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 the VFTABLE table (runtime type information).

Example.

```
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.

Moreover, for correct use, it is imposed a re-conversion to the initial type of the converted type.

Example 9.9.

```
#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;
}
```

B. 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);
```

if the function *f* is called with a integer parameter, *f(5)*, the compiler does a automatic conversion from the type *int* to *double*.

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 the ***conversion constructors***, or using the ***conversion operators***.

In the first case of using conversion constructors, an automatic conversion can be realized, *from the type of the parameter of the respective constructor to the type of the class where the constructor is defined*.

Example:

```
class A {
  int n;
public:
  // conversion constructor from the type 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;
}
```

At the second call of the function *f* is called the conversion constructor of class *A*.

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 9.11.

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

A second method which allows the automatic conversion of the types 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:

```
operator <name_class> ()
```

It is observed that the operator's name is the name of the destination class of the conversion, and the data type of the operator function is missing, being replaced by the keyword `operator`.

Example 9.12.

```
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;
  h(c);    // the conversion operator is called
  h(1);    // the conversion constructor is called
  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(a);          // Error!!
  return 0;
}
```

In the previous example, it can not be determined which version of the function *f* must be called.