

Functional Extension of Language C++.

I. IF and BREAK Statements. VOID and CLASS Types.

M.A.Malkov

Russian Research Center for Artificial Intelligence.

E-mail: ma@malkov.msk.su

Contents

1. Statement IF and Type VOID	84
2. Operations of post-Assingment	86
3. Statement BREAK	86
4. The Extension of Classes	87

The programming languages C and C++ have become the most popular ones due to their functionality.

The functional language has functions and no procedure. Besides in the functional language all statements work out value.

The functional languages have two stage of abstraction. The first stage is languages for the abstract computer with memory (the languages of von Neumann type). The second stage is languages for the abstract computer without memory (e.g. the languages of λ -calculus type).

The languages C and C++ belong to the first stage. They have no procedure, but the majority of their statements are not functional. The given work opens out a series of articles dedicated to an extension of the language C++, such that all the statements of this language become functional.

1. Statement IF and Type VOID.

The statement working out a value becomes an operation. Therefore, the statement "if" becomes a conditional operation, if it works out some value.

The conditional operation in the language C++ already exists, but its possibilities are extremely bounded because of the absence of the brief form of this operation.

The conditional statement has two basic forms:

if (A) B; else C;

if (A) B;

The conditional operation has only one form:

A? B:C;

The proposed extension is an attachment of one more form of this operation:

A? B;;

The value of the given operation is "B", if "A" is not equal to 0, or "void", if "A" is equal to 0.

One more extension of the language C++ is the keyword "void" not only to be a basic type, but also a value.

The value "void" belongs to any type. Besides this the value also belongs to the set of types, i.e. the keyword "void" remains a basic type.

The result of operations with one argument to be "void", should be determined in the specification of these operations. In particular, the left argument of assignment does not vary, if the right argument is "void". The result of this operation is "void" too.

EXAMPLE 1. E=D= A? B;;

If the value "A" does not equal 0, then E=D=B. Otherwise the values "D" and "E" do not vary. The same result we shall receive for the statement "if":

if (A) E=D=B; □

The statement "if" is possible to eliminate from C++, as all of its functions are executed by the conditional operation "?". Besides the conditional operation is functional in the sense that the statement "if" is not sound, if it stands after a sign of equality. The conditional operation is sound in any position.

The further extension of the language C++ is in introducing of the shortened construction of the conditional operation "A? B;;", used for realization of iterations (cycles): the iteration body "B" is

executed many times, if the value A is not equal to 0. The value of such operation is the variable equal to "B" after each iteration. After termination of iteration, i.e. at the last inspection of the condition "B", the value of the operation is equal to "void".

This extension of the conditional operation allows to eliminate such statements as "for", "while", and "do".

The statement "for (A; B; C) D;" is possible to be introduced in the form:

A; B? (D, C);

The statement "while (A) B;" can also be substituted by the shortened conditional operation:

A? B;

The statement "do B; while (A);" is similarly substituted:

n=1; n | A? (n=0, B);

The conditional operation in the shortened form is similar to the statement "while", but this operation is more functional in the mentioned above sense.

While using the statements "for", "while" and "do" it seems, that a program looks more understandable and obvious. Actually, we have just got used to these statements. Temporarily having eliminated these statements, we then shall refuse to apply them (especially with new semantic of post-assignments, see below).

EXAMPLE 2. The calculation of the sum of factorials is possible to be introduced by iterations:

factorial=S=1; int k=0; S+= factorial*= k<n? ++k;

On each step of iteration "factorial" is multiplied by "++k", the result of multiplication is added to "S". At the end of iteration process (k=n) result of iteration becomes "void". Therefore, at the end of iteration the values "factorial" and "S" do not vary. □

The proposed extension of a conditional operation is simple, but the obviousness of the program suffers.

The programs become more visual and clear if we complicate syntax and semantics of the extension:

- if the conditional operation uses iteration, the sign "?" is doubled;
- the operation "A? B;" can be substituted by the operation "A? B?";
- the succession of the statements after the sign "??" can be taken in parentheses or in braces;
- semantics of operations of post-assignment is changed: in the statements containing cycles, these assignments are executed after each iteration, in the statements not containing cycles - after execution of the statement, in both cases post-assignments fail, if the statement is not executed because of the value "void".

The statement is meant as an operation with the result not to be used. Inside the statement there can be other statements called the substatements.

As the value, which is worked out by a statement, will not be used, the calculation of this value is blocked as superfluous.

EXAMPLE 3. The statements

int i; char c[4]; A? i=5: c="abc";

are correct, because calculation of the value of the conditional operation is blocked. □

Post-assignments are the operations of post-decrement and post-increment. In the existing language C++ such increments and decrements are calculated in a condition part of conditional operation before finishing this operation, but it is illogical.

So the change of semantics of post-assignment is not an extension of the language C++, because it contradicts this language. But we can use it as an option of C++. The other option does not change C++.

The extension, permitting to conclude the sequence of operations in braces instead of parentheses, will be in detail explained in the next articles.

EXAMPLE 4. A++? B++:C++;

If "A" is not equal to 0, the result of the operation is "B", after execution of the operation values "A" and "B" will increase by 1, the value "C" will not change. If "A" is equal to 0, the result of the operation is "C", after execution of the operation values "A" and "C" will increase by 1, the value "B" will not change. □

2. Operations of post-Assignment.

In the existing language C++ there are only two operations of post-assignment. In extension of the language C++ all the operations of assignment can be not only prefix, but also postfix:

$=*$, $=/$, $=\%$, $=+$, $=-$, $=\ll$, $=\gg$, $=\&$, $=|$, $=\wedge$.

A blank should follow after symbols of these post-operations.

The post-decrements and post-increments are a brief record of operations with assignment:

"A++" is equivalent to "A+= 1", "A--" is equivalent to "A-= 1".

The operation of post-assignment has the following semantics:

- the value of the right argument is stored;
- the result of the operation of assignment is the value of the left argument (this means, that the operation of assignment is stored, its execution is put off);
- if the statement does not contain a cycle, all stored operations of assignment are executed after the execution of the statement;
- if the statement contains one cycle, all stored operations of assignment are executed after each iteration, but after termination (i.e. value "void") of iteration the stored values of operations are ignored and removed;
- if the statement contains some inserted cycles, a substatement of the inner cycle executed by the previous rule, then a substatement of the following (included) cycle is executed etc., and the substatement of a given cycle is a body of the following cycle or a part of this body.

EXAMPLE 5. $S=0; \text{int } i=0, j; i++\langle n?? j=0; j++\langle m?? S+=M[i][j];$

In this example there are two cycles and one statement "i++<n?? int j=0; j++<m?? S+=M[i][j];" with the substatement "j++<m?? S+=M[i][j];". If $i < n$, the operation post-assignment for i is stored and the internal cycle with the old value i is executed. After each iteration of the internal cycle, j is increased by 1, after termination of the iteration process (i.e. at $j=m$) j does not change (i.e. j does not change at the last inspection of the condition). The termination of an internal cycle is simultaneously the termination of one iteration on i . Then the stored operation of increment is executed and the value i is increased by 1.

EXAMPLE 6. $A--??B++;$

If "A" is not equal to 0, the value "B" is the result of one iteration. Further the value "A" is decreased by 1, and the value "B" is increased by 1.

If the new value "A" is not equal to 0, the new value "B" is the result of the second iteration. After this iteration, stored assignments are executed and values "A" and "B" again change.

If the new value "A" is equal to 0, the execution of the operation is ended and "A" does not change the value, since the stored operation of assignment defaults.

In the existing language the value "A" will be equal to -1 after a termination of a cycle.

Similarly for the operation "A++ < n?? B;" the value "A" after a termination of a cycle equals "n", if before the beginning of a cycle we have $A \leq n$. If before the beginning of the cycle we have $A > n$, the value A will not change. In the existing language the value A will equal to $n+1$ in the first case or will be increased by 1 in the second case.

3. Statement BREAK.

The statement "break" works out the value "void". This statement becomes overloaded. The body of this statement can be empty, can contain a label to be identifier, can be a call of function or can be an expression of some type.

If the label in a body of the statement "break" is missing, the usual execution will be realized as in the existing language C++, but all stored assignments are removed.

The statement *break* with a label in its body executes the next functions:

- to interrupt execution of an operation with this statement in the body of the operation, and all stored assignments removed;
- to point the statement for continuing the program after interrupting;
- to substitute the statement "go to";

- to substitute the statements “try”, “throw” and “catch” in the exceptions defined by a programmer.

The label, added by a colon, should precede the statement executed after iteration interrupting.

The label in operation “break” is special, it cannot be used the same way as in the statement “go to”:

- the statement marked by this label is executed only after interrupting by the according operation “break”, in all other cases it is skipped (not executed);
- after execution of the marked statement the next unmarked statement is executed (the other marked statements are skipped);
- the labeled statement can be out of the function containing “break”.

If the labeled statement is missing in the body of a function with *break*, the execution of this function ceases and the exception is handled in a function calling former, if the calling function contains the label. If the calling function does not contain the label, this function ceases too, and the exception is handled in the next calling function etc.

The label in the statement “break” can be absorbed by a standard label. The standard label is an analog of the statement “catch (...)”.

If it is necessary to give some information to the exception handler, then the label in the statement “break” is replaced by the call to a special function. This function is the exception handler. The definition of this function is located in the place where the exception handler should be. Between the head and body of the function the colon is inserted.

Thus, the exception handler has the next format:

functionName(formalParameterSpecification) : functionBody

This function always works out the value “void”. Therefore the type of the function can be omitted.

After function execution the first unmarked statement behind the function starts to work. The return to a call point does not take place.

For compatibility with the existing language the label in the statement “break” can be substituted by an expression of any type. In this case the statement “break” coincides with the statement “throw”. The exception handler becomes:

(exceptionSpecification):handlerStatements

The difference from the statement “catch” is in the absence of the keyword “catch” and in adding a colon.

The statement “try” is superfluous. Its block can be found by using localization of the exception handler.

4. The Extension of Classes.

The notion of class has appeared as an analog to the notion of mathematical structure.

The mathematical structure includes a set (data of some type) and operations (functions) defined on this set.

But such mathematical structure corresponds to one-sorted theory. For many-sorted theories the mathematical structure contains several sets. The data of different types correspond to these sets.

Being a mathematical structure with one set the class can have the same name as the data type. Being a mathematical structure with some sets, the class can have the name coinciding with the name of only one type. But then it is impossible to construct objects of other types from this class.

So we should construct a number of classes equal to a number of sets in mathematical structure. One of this classes is considered as basic, the other classes are considered as its “friends”.

But the concept of a class containing several data types and several functions is more natural. All these types are equal in rights and each of them can construct objects. The notions “class” and “mathematical structure” become really the same.

Instead construction “objectName.functionName(arguments)” we can use more natural construction “functionName(object.Name,arguments)”. We can use more contemporary way to allocate additional memory. We leave noting (except modifcator “friend” and pointer “this”) but get the most powerful tools for constructing the great program complexes with “up-down” technology of structural

programming. This is a real object oriented programming with the encapsulation, succession and polymorphism.

But the given offer changes the existing syntax. We need to ensure the succession of syntax for our changes.

For that the old syntax is completely saved, but it is added by one more format of the class declaration:

```
class className [:based classes] classMemberList
```

If a class contains only the data of one type, the existing syntax is used, but it is not recommended to use keywords “friend” and “this”.

If a class contains the data of several types, the class name stops to be a data type. All data types are defined inside a class with the help of the keyword “struct” with an implicit qualifier of access “public”. The members of structures have an implicit qualifier of access “private”.

EXAMPLE 8. Let's determine points, triangles and circles on a plane.

In the existing language three classes are constructed, the second and third class are friends to the first class, the third class is a friend to the second class:

```
class Point // the class of points
{double x, y; friend class Circle; friend class Triangle;
public: Point(){} Point(double x,double y){this->x=x; this->y=y;}//constructors
}; class Triangle //the class of triangles, given by three points
{Point p1, p2, p3; friend class Circle;
public: Triangle(){} Triangle(Point p1, Point p2, Point p3)//constructors
{//exception is arisen, if three points lay on the straight line:
if ((p1.x-p2.x)*(p2.y-p3.y)==(p2.x-p3.x)*(p1.y-p2.y))
throw “triangle is line”; this->p1=p1; this->p2=p2; this->p3=p3;
}
};class Circle //the class of circles, given by their center and radius
{Point p; double r; public:Circle(){} Circle(Point p, double r)//constructors
{//exception is arisen, if  $r \leq 0$ :
if (r<=0) throw “r<=0”; this->p=p; this->r=r;
}static Circle circumscribed(Triangle); //value of the function is the
//circle to be circumscribed about a given triangle.
static void Cout(Circle); //a circle display
static void Tout(Triangle); //a triangle display
};
```

In the expanded syntax these three classes are joined in one:

```
class Plane
{struct Point {double x,y}; //x and y are private, but protected into Plane
struct Triangle {Point p1,p2,p3}; //p1, p2 and p3 are private
struct Circle {Point p;double r}; //p and r are private
public: Point(){}; Point(double x, double y) {Plane::x=x; Plane::y=y;}
Triangle(){}; Triangle(Point p1, Point p2, Point p3)
{(p1.x-p2.x)*(p2.y-p3.y)==(p2.x-p3.x)*(p1.y-p2.y)? break “triangle is line”;
Plane::p1=p1; Plane::p2=p2; Plane::p3=p3;
}Circle(); Circle(Point p, double r)
{r<=0? break “r<=0”; Plane::p=p; Plane::r=r;
}Circle circumscribed(Triangle);
Circle incircle(Triangle);
void Cout(Circle);
void Tout(Triangle);
}
```

This construction is more natural, more clear, and what is important, more safe¹.

¹ Autor acknowledges M. Kamenshchikov for his excellent remarks.