PLUK LANGUAGE

Version 3.99

Reference Book

Moscow, 2001

1 CONTENTS

2 INTRODUCTION

2.1 Application

Pluk Language (hereinafter Pluk) is a general-purpose programming language. It is independent of any particular operating system and designed to be used on various software and hardware platforms (e.g. Windows, UNIX).

As of today, Pluk has been implemented on Windows 9x / NT / 2000. The implementation on UNIX has been temporarily discontinued.

2.2 Overview

In the hierarchy of contemporary languages, Pluk represents the third generation of languages. Pluk is an object-oriented interpreter (a 'lazy' compiler). Syntactically it is similar to C++. Like C++, it contains all means of procedure-oriented programming, which allows the 'old school' programmers to switch to object-oriented programming pretty easily. Frankly, the concept of object-oriented programming, in its 'pure' form, sometimes requires unnecessary and non-productive operations, such as development of classes, whose only purpose being to introduce procedures into the language.

Since the very beginning of its creation, Pluk has been intended to be a mixed (Pluk / C++) programming environment. Pluk could be used as an embedded interpreter: the main program is written in C++, but some ancillary functions (e.g. references to database) are executed by the Pluk machine called from the main program. To address this issue, there is a universal mechanism in Pluk for attaching functions and methods of C++. This mechanism enables the user to create classes whose methods are written partly in C++ and partly in Pluk. It is also possible to create an associated C++ class. Each object of this class will be associated with an object of a Pluk class. This allows hiding part of code and data in the C++ class, leaving just the interface at the Pluk level. It is possible to create Pluk objects in and invoke Pluk methods from code written in C++. In this case, if Pluk objects support redefined operators, the latter will be invoked 'transparently' through the C++ program.

Since Pluk is a high-level language interpreter, it imposes certain limitations on the use of some C++ methods. It is not possible, for example, to manipulate computer memory from the Pluk level, just in contrast to the 'high-level assembler' (a nickname of C++, given for its ability to access computer resourses directly). This limitation (it can be easily bypassed by means of associated C++ class), however, spares the programmer such tasks as freeing allocated memory, maintaining the integrity of array and string boundaries, etc.

An important feature that distinguishes Pluk from C++ is non-type variables. Naturally, at a certain point in time, a variable takes the type of its value (for example, a number, string, vector, object). However, the name associated with the variable does not have type. If, for instance, the variable **`b`** initially contained a numerical value **`5`**, its type was **`int`**. If later it were assigned a string **`"hello"`**, its type would be **`String`**. So, at the stage of coding, the interpreter deals with an abstract variable of no particular type. The type check takes place later, at runtime. For example, an attempt to add the variable **`b`** equaling **`"hello"`** to the variable **`a`** equaling **`5`** will throw an error at runtime.

Another feature that differs Pluk from C++ is that a function is a variable that contains a string of a special type: an interpretable string. Class methods, although not regular variables, also contain interpretable strings. It allows handling both functions and methods as regular variables of string type. The programmer can treat an interpretable string as a conventional string: find sub-string, replace / delete sub-string, merge strings, etc. Pluk does not support overloaded global functions (coinciding by names and differing by types of parameters) since a function is a variable and should be accessed by name. A variable name does not contain any auxiliary names that define types of parameters since the variable that presently contains a function may contain something else in the future (a number, for example). Class methods do carry additional information that determines the types of parameters; hence, Pluk supports overloaded methods. The type of a parameter can be determined only at runtime since just at that time the relationship between a method and its invocation point could be revealed.

Although Pluk is an interpreter, it is not continuously interpreting source code. Instead of doing this, it compiles source code into intermediary B-code to be executed by the virtual Pluk machine. A function is compiled at its first call only. As a result, compilations are 'spread' in time: new functions will be compiled, but already compiled code will be used in recalls.

There are two types of interpretable strings:

- an interpretable string of type **`sfunc`** (between the limiters **`<|`**…**`|>`**), compiled into B-code, preserving useful information for function debugging;
- an interpretable string of type **`rfunc`** (between the limiters **`{|`**…**`|}`**), compiled into the processor's machine-code with a loss of useful information critical for function debugging; the Pluk machine executes machine-code much faster than B-code, but the former allocates much more memory.

The Pluk machine of version 3.xx is a stack machine with an accumulator. It uses 103 four-byte statements. The first byte determines the type of a statement; all the other bytes depend on the statement type. When executing an interpretable string of type **`sfunc`**, the interpretation of statements in B-code is taking place. While compiling an interpretable string of type **`rfunc`,** machine-code is being generated for each statement in B-code. It goes without saying that an interpretable string of type **`rfunc`** dramatically falls behind C++ code in performance.

2.3 Requirements to User Skills

The user should have basic computer skills and learn this manual. She also needs to be familiar with the Standard Library Reference Book and Pluk IDE User Guide.

3 PROGRAM STRUCTURE

One of the main features that distinguish Pluk from C++ is that classes, methods and functions in the former are not defined by declarations, being just executable code. Due to this fact, classes, methods and functions can be defined at any stage of execution, similar to regular variables.

This allows the programmer to write programs in Pluk 'on the fly'. She can run part of source code, then change it or add something, and, without program termination, update the code. Updated functions in the stack will be duplicated: old bodies of the functions will remain in the stack until completion of their execution; new bodies of the functions will be used on recalls. Objects of updated classes are not deleted, but adopted to new class definitions.

If the programmer needs to create a release program, she can group source files into a project file. A project file contains the executable sequence of program files. Upon launching of the program, these files will be read in this sequence. A file's content will be assigned to a temporary function, which will be executed and deleted afterwards. All local variables defined in the function (but not in the functions defined in it) will be deleted upon its deletion. Once the last file has been executed, the program will not terminate. It will wait for events from the environment (for example, from the window system) sent to static objects, or their sub-objects, created at runtime. The static objects will be deleted and the program will terminate after either executing an **`end`** statement (see 5.7.5) or closing the program from the environment (for example, in the Windows environment, a program may be closed from the task bar).

Note that the sequence of file execution in a project is very important in Pluk, where definitions of classes, methods and functions are executable statements. A base class must be defined before a derived class, a class method must be defined after the class, etc. If a constant (see the **`#define`** statement, 5.7.6) is defined after its name has been used in a function, the code, referring to the global variable with such name, will be generated in the function. If the constant is still undefined at the moment of function execution (and there is no variable of such name), the error **`"Unknown variable"`** will be thrown.

4 WRITING PROGRAMS

There is no obligatory pattern for writing programs in Pluk. The programmer may resort to any layout just for reader's convenience. Space, tab and carriage return are regarded as lexeme delimiters. If a CR is located within a string, it is regarded as part of the string:

```
new a = "beginning and...
end of the string";
```

Comments in code could be of two types: `/*...*/` — block comments and `//...` — line end comments. Arbitrary nesting is allowed for the both types.

There is a special case of comments of the first type `/**...**/`. This is a system comment that is related to the body of a function, method, or class, which it precedes. A system comment is used for documentation of a function, method, or class. It is accessible at runtime in the Pluk IDE (see Pluk IDE User Guide) or by means of the methods of the class **`Pluk`** (see Standard Library Reference Book). For example:

```
/** Prints the string **/
global PrintStr =
<|
	// ...
|>;
/** Class of the device **/
class Device
{
	// ...
};
/** Opens the device **/
new Device::Open(void) =
<|
	// ...
|>;
```

If text of a system comment contains a **`#private`** statement, the function, method or class are private (not public) and invisible, under certain conditions, in the windows of the Pluk IDE. It allows the programmer to keep global functions and classes, needed just for technical purpose, away from the environment's browsing windows. For example:

```
/**
#private
This is a private class
**/
class A
{
	// ...
};
```

Text of a system comment may also include a **`#module`** statement`,` followed by the path to the module containing the class. The path consists of the names of modules and their sub-modules, separated with periods. If a **`#module`** statement is not present in the text, it means that the class is located in the root module. Breaking classes into modules allows displaying all program classes in Pluk IDE's windows in a tree-like structure. For example:

```
/**
#module root.system.filesystem
**/
class File
```

```
{
        // ...
};
```

5 LANGUAGE ELEMENTS

5.1 Variables

A variable name is a string comprised of letters, numbers, and underscores (up to 32 symbols). It must start with a letter or underscore, and is case-sensitive. A variable name may not coincide with a language keyword. The programmer may use Cyrillic letters. Below, there are some examples of variable names:

```
a;                      // correct
a12;                    // correct
_MyFunction;            // correct
12a;                    // wrong - cannot start with a number
```

Defining a variable, the programmer needs to indicate its scope of existence. The **new** statement defines a local variable, the **global** statement a global variable. It is always necessary to use these statements since it might help avoid accidental definitions in case of typing errors. For example:

```
new a;     // definition of a local variable
new a = 5; // definition and initiation of a local variable
global b;  // definition of a global variable
c = 10;    // implies the existence of variable c defined
earlier,
           // or else an error is thrown at runtime
```

A global variable exists until completion of a program or execution of the **delete** statement (see 5.7.3). A local variable exists as long as the function, in which it is defined, is being executed. If a local variable has been defined in a block (within the braces), it will be inaccessible from outside the block if called by name, although it will be deleted only upon exiting the function. Such a delayed deletion may be of some importance just in the case a local variable is an object with the destructor. The destructor will be invoked only upon exiting the function:

```
new A::~A(void) =
<|
      trace "deleted";
|>;
global F =
<|
param x;
      if (x > 0)
      {
            new a = instance A;
            trace "Object ";
      }
      trace "was ";
|>;
F(1);
```

The above example will print: 'Object was deleted'.

A global variable may not be defined within the braces:

```
if (n > 1)
{
        global X;       // error
}
```

However, it is possible to define a global variable in the body of a function. It occurs, for example, at the execution of a project's file that contains the global variable definition. The content of the file is considered to be the body of a temporary function. The variable **X** will appear after execution of the function **F** in the following example:

```
global F =
<|
        global X;
|>;
```

The above is also true for methods. For example, the method **A::G** will appear only after execution of the method **A::F**:

```
new A::F(void) =
<|
        new A::G(void) =
        <|
                // ...
        |>;
|>;
```

Non-initialized variables are of type **empty** and contain an **EMPTY** value.

5.2 Data Types

Following are the types of data supported by Pluk:

Token	Description
empty	Non-initialized value.
boolean	Logical value.
char	Unsigned 8-bit integer.
int	Signed 32-bit integer.
float	Floating-point 32-bit number.
double	Floating-point 64-bit number.
sfunc	Pluk function (interpretable string) compiled into B-code.
rfunc	Pluk function (interpretable string) compiled into the processor's machine-code.
cfunc	C++ function called from Pluk program.
pointer	Pointer at a variable (not allowed pointers at variables of the types **empty**, **boolean**, **char**, **int**, **float**, **double**).
object String	String for storing an arbitrary sequence of bytes. May contain arbitrary bytes, including zero bytes. A built-in class.
object Vector	Vector for storing an arbitrary sequence of variables. May contain elements of different types. For example, a vector having a number in the first element, a string in second, and another vector in third. A built-in class.
object Class	User-defined class.

The above tokens are used to determine the types of parameters taken by a method. In addition to type tokens, there are the following tokens may be used:

Token	Description
void	Empty list of parameters. The only parameter on the method parameter list.
number	Any numeral (**char**, **int**, **float**, **double**) or a child of the class **Number.**
func	Any function (**sfunc**, **rfunc**, **cfunc**) or a child of the class **Function**.

`any`	Any type.
`copy`	Parameter of the copy constructor. The only parameter on the method parameter list.
`...`	Variable number of parameters. The last item on the method parameter list.

When defining a parameter of type **`pointer`**, it is useful to specify the type it points at: **`pointer object Class`**.

Sometimes the user needs to define a class representing number or function. In order to define a numerical class, it is necessary to derive it from the class **`Number`**. One should note that **`object Number`** is synonymous to **`number`**. The numerical types **`char`**, **`int`**, **`float`**, **`double`** are the final children of the class **`Number`**, i.e. nothing could be derived from them. In order to define a functional class, it is necessary to derive it from the class **`Function`**. One should note that **`object Function`** is synonymous to **`func`**. The functional types **`sfunc`**, **`rfunc`**, **`cfunc`** are the final children of the class **`Function`**, i.e. nothing could be derived from them.

A pointer always points at an object, not at the name of the variable containing an object. If the object has been moved into another variable (with another name), the pointer will point at the object under the new name. For example:

```
        new a = instance A;
        new ptr = &a;               // ptr points at a
        new b;
        b <- a;                     // now ptr points at b
```

If a pointed-at object has been deleted, a pointer will point at an object of the class **`Dump`**, which is void of fields and methods:

```
        new A::F(void) =
        <|
            // ...
        }>;
        new a = instance A;
        new ptr = &a;           // ptr points at a
        ptr->F();               // a method of class A is invoked
        a = EMPTY;              // now ptr points at an object of class Dump
        ptr->F();               // error – a method of class A, not class
Dump
```

5.3 Operators

The following operators are recognized by Pluk:

Operator	Description
`=`	Performs destructive assignment of second operand to first. For all types of operands.
`:=`	Copies field namesakes of second operand into first, i.e. only the object fields whose names are present in the both operands will be copied. Second operand's fields equaling **`EMPTY`** will not be copied. For all types of operands.
`<-`	Performs destructive moving of second operand into first. Upon executing of the operator, second operand equals **`EMPTY`**. For all types of operands.
`+`	Adds second operand to first.
`+=`	Adds second operand to first, assigning the result to first.
`-`	Subtracts second operand from first.
`-=`	Subtracts second operand from first, assigning the result to first.
`+`	Unary plus, does not change the operand.
`-`	Unary minus, changes the sign of the operand.
`++`	Increment of the operand. In contrast to C++, no postfix increment.

`--`	Decrement of the operand. In contrast to C++, no postfix decrement.
`*`	Multiplies first and second operands.
`*=`	Multiplies first and second operands, assigning the result to first.
`/`	Divides first operand by second.
`/=`	Divides first operand by second, assigning the result to first.
`%`	Produces the remainder of division of first operand by second.
`%=`	Produces the remainder of division of first operand by second, assigning the result to first.
`**`	Takes first operand to the power equaling second.
`&`	Bitwise AND on first and second operands.
`&=`	Bitwise AND on first and second operands, assigning the result to first.
`|`	Bitwise OR on first and second operands.
`|=`	Bitwise OR on first and second operands, assigning the result to first.
`^`	Bitwise XOR (exclusive OR) on first and second operands.
`^=`	Bitwise XOR on first and second operands, assigning the result to first.
`~`	Bitwise NOT on the operand.
`&&`	Logical AND on first and second operands. Returns a Boolean value.
`||`	Logical OR on first and second operands. Returns a Boolean value.
`^^`	Logical XOR (exclusive OR) on first and second operands. Returns a Boolean value.
`!`	Logical NOT on the operand. Returns a Boolean value.
`==`	Compares first operand with second on equality. Returns a Boolean value. For all types of operands.
`!=`	Compares first operand with second on non-equality. Returns a Boolean value. For all types of operands.
`<`	Compares first operand with second on less-than relation. Returns a Boolean value.
`>`	Compares first operand with second on greater-than relation. Returns a Boolean value.
`<=`	Compares first operand with second on less-than-or-equal-to relation. Returns a Boolean value.
`>=`	Compares first operand with second on greater-than-or-equal-to relation. Returns a Boolean value.
`<>`	Compares first operand with second on less-than-equal-to-greater-than relation. Returns `-1` — less than, `0` — equal to, `1` — greater than.
`? :`	Conditional choice. Returns second operand if first equals **`TRUE`**, or third operand if first equals **`FALSE`**.
`<<>>`	Creates a vector with elements in the angle brackets, separated with commas.
`,=`	Appends second operand to the end of first, assigning the result to first. First operand must be a vector.
`@`	Merges first and second operands or appends second to the end of first. In the case of merging, first and second operands must be strings / vectors. In the case of appending, first operand must be a vector.
`@=`	Merges first and second operands or appends second to the end of first, assigning the result to first. In the case of merging, first and second operands must be strings / vectors. In the case of appending, first operand must be a vector.
`[]`	Access to an element of first operand by index (second operand). First operand must be a vector, second an integer.
`()`	Function call. The function parameters are in parentheses, separated with commas. If the operand is not a function, the call will return the operand value.
`@`	Unary operator. Allows passing a parameter to a function by reference and return of a

	value from a function by reference.
`[]`	Unary operator. Allows passing a variable number of parameters to a function.
`&`	Unary operator. Gets a pointer.
`*`	Unary operator. Dereferences a pointer.
`.`	Accesses a field of first operand by name (second operand). First operand must be an object.
`.*`	Accesses a field of first operand by the name contained in second. First operand must be an object, second a string.
`.`	Unary operator. Accesses an object field by name. May be used only in the body of a method of the object being accessed.
`.*`	Unary operator. Accesses an object field by the name contained in the operand. The operand must be a string. May be used only in the body of a method of the object being accessed.
`-> ()`	Invokes a method of first operand, with parameters in parentheses and separated with commas, by name (second operand). First operand must be an object.
`->* ()`	Invokes a method of first operand, with parameters in parentheses and separated with commas, by the name contained in second. First operand must be an object, second a string.
`::`	Resolves a class name (first operand) for a field / method name (second operand).
`::`	Unary operator. Resolves a global variable / function name.

One cannot redefine operators of the built-in types. User-defined operators use the number of operands as described above, but may be of different type. For example, the user may define a binary operator `[]` from the string:

```
new A::[](object String) =
<|
     // ...
|>;
```

Following are the operators that may not be defined by the user:

- `=`
- `:=`
- `<-`
- `? :`
- `<<>>`
- unary `@`
- unary `[]`
- unary `&`
- unary `*`
- `.`
- `.*`
- `->`
- `->*`
- `::`

For the type **`empty`**, as for any other type, the operators **`==`** and **`!=`** are defined. When sorting vectors containing some **`EMPTY`** elements, the comparison operators treat an **`EMPTY`** value as the least of all values. Thus, all of the following expressions equal **`TRUE`**:

```
EMPTY < 0
EMPTY < "aaa"
EMPTY < instance A
EMPTY < FALSE
```

5.4 Operator Precedence

In the table below, the operators are enumerated in the ascending order of their precedence:

Operator	Precedence	Description
@	1	Unary operator. Allows passing a parameter to a function by reference and return of a value from a function by reference.
=	2	Assignment.
:=	2	Copying of field namesakes.
<-	2	Moving.
+=	2	Addition with assignment.
-=	2	Subtraction with assignment.
*=	2	Multiplication with assignment.
/=	2	Division with assignment.
%=	2	Producing the remainder of division with assignment.
&=	2	Bitwise AND with assignment.
|=	2	Bitwise OR with assignment.
^=	2	Bitwise XOR (exclusive OR) with assignment.
,=	2	Appending with assignment.
@=	2	Merging with assignment.
? :	3	Conditional choice.
||	4	Logical OR.
^^	4	Logical XOR (exclusive OR).
&&	5	Logical AND.
==	6	Comparison on equality.
!=	6	Comparison on non-equality.
<	6	Comparison on less-than relation.
>	6	Comparison on greater-than relation.
<=	6	Comparison on less-than-or-equal-to relation.
>=	6	Comparison on greater-than-or-equal-to relation.
<>	6	Comparison on less-than-equal-to-greater-than relation.
|	7	Bitwise OR.
^	7	Bitwise XOR (exclusive OR).
&	8	Bitwise AND.
+	9	Addition.
-	9	Subtraction.
@	9	Merging.
*	10	Multiplication.
/	10	Division.
%	10	Producing the remainder of division.
**	11	Exponentiation.
++	12	Increment.
--	12	Decrement.
+	13	Unary plus.
-	13	Unary minus.
~	13	Bitwise NOT.
!	13	Logical NOT.

`&`	14	Unary operator. Gets a pointer.
`<<>>`	15	Creation of a vector of elements.
`[]`	15	Access to a vector's element.
`()`	15	Function call.
`.`	16	Access to a field.
`.*`	16	Access to a field (indirect).
`-> ()`	16	Method invocation.
`->* ()`	16	Method invocation (indirect).
`*`	17	Unary operator. Dereferences a pointer.
`.`	18	Unary operator. Access to a field.
`.*`	18	Unary operator. Access to a field (indirect).
`::`	19	Resolution of a class name.
`::`	19	Unary operator. Resolution of a global variable / function name.
`()`	20	Change of precedence.
`[]`	20	Unary operator. Allows passing a variable number of parameters to a function.

5.5 Constants

The user can define constants of any built-in type, excluding **`cfunc`**. For example:

```
EMPTY                // non-initialized constant
TRUE                 // Boolean constant
'a', '\0x7f'         // symbolic constant
5, 376799, 0x877f    // integer constants
5.0, 8.77e10         // 32-bit floating-point constants
3.14159265359        // 64-bit floating-point constant
"this is a test"           // string constant
<|param a; return a * a;|> // functional constant of type sfunc
{|param a; return a * a;|} // functional constant of type rfunc
<<1, 2, "test", 5.0>>      // vector constant
```

Symbolic and string constant may take hexadecimal values. The following symbolic constants are equivalent to each other (the same holds for string constants):

```
'\0xff', '\0xFF', '\xff', '\xFF'
```

A constant may be defined as an object of a class whose constructor can be executed at the compiling stage. The following statement defines the constant **`CLR_LIGHTRED`**, being an object of the class **`ColorRef`**:

```
#define CLR_LIGHTRED       instance ColorRef(255, 0, 0)
```

Following are the built-in constants in Pluk:

Constant	Description
`EMPTY`	Non-initialized constant.
`TRUE`	Boolean constant means true.
`FALSE`	Boolean constant means false.
`PI`	Constant π.
`CR`	Carriage return.
`NULL`	0.
`FD`	Symbol, a separator within a filename.
`SFD`	Single-symbol string, a separator within a filename.
`CmdLine`	Vector of command line parameters, excluding system parameters (for

	example, `/debug`).
`ExeName`	Full name of the program executable file.
`LogFile`	Full name of the program log-file. The constant will appear only if the program starts with the **`/l`** parameter on the command line.
`LogStackOnError`	If this Boolean constant is set, any error will result in tracing the stack content into the log-file. The constant will appear only if the program starts with the **`/stack`** parameter on the command line.

There are several dozens of constants that start with the prefix **`ERR_`**. They are the codes of system errors that may be thrown by the Pluk machine and standard libraries.

5.6 Expressions and Statements

All statements are terminated with a semicolon:

```
a = b + c * d;
for (new i = 0; i < n; ++i)
      x[i] = i;
```

Expressions and statements in Pluk look similar to those in C++. Some operators in Pluk, however, have different precedence. For example, the precedence of the dereference operator **`*`** is much higher in Pluk than in C++.

To specify the evaluation order (if it is necessary to change operator precedence), the parentheses **`()`** are used. To group a set of statements in one statement, the braces **`{}`** are used.

Below there are the flow control statements.

5.6.1 Conditional Statement

```
if (expression)
      true-expression;
[else
      false-expression;]
```

If the ***`expression`*** evaluates to true, the ***`true-expression`*** is executed; otherwise, the ***`false-expression`*** (may be omitted) is executed.

5.6.2 Loop Statement with Initialization

```
for (init-expression; expression; after-expression)
      body-expression;
```

The ***`init-expression`*** is executed, then the ***`expression`***. If the ***`expression`*** is evaluated to true, the ***`body-expression`*** and ***`after-expression`*** are executed, and the loop will reiterate. If the ***`expression`*** evaluates to false, the loop will be terminated.

Note that variables defined in the ***`init-expression`*** remain defined even after loop termination (in contrast to standard C++). So there is an example of checking the reason of loop termination:

```
for (new i = 0; i < n; ++i)
      if (a[i] == EMPTY)
            break;
if (i < n)
      trace "EMPTY found", CR;
```

5.6.3 Prefix Loop Statement

```
while (expression)
      body-expression;
```

The ***expression*** is executed in the loop. If the ***expression*** evaluates to true, the ***body-expression*** is executed, and the loop will reiterate. If the ***expression*** evaluates to false, the loop will be terminated.

5.6.4 Postfix Loop Statement

```
do
      body-expression
while (expression);
```

The ***body-expression*** and ***expression*** are executed. If the ***expression*** evaluates to true, the loop will reiterate. If it evaluates to false, the loop will be terminated.

5.6.5 Switch Statement

```
switch (switch-expression)
{
[case case-expression1:
      expression1;
      [break;]]
[case case-expression2:
      expression2;
      [break;]]
…
[case case-expressionn:
      expressionn;
      [break;]]
[default:
      expressionn+1;
      [break;]]
}
```

The ***switch-expression*** and ***case-expression***$_1$ are executed. If their values are equal, the ***expression***$_1$ is executed. If the ***expression***$_1$ is followed by a **break** statement, the switch statement is terminated; otherwise, the ***case-expression***$_2$ is executed and its value is compared to the value of the ***switch-expression*** (evaluated only once), etc. If all of the above equations are false or there is no **break** statement after the ***expression***$_n$, the ***expression***$_{n+1}$ is executed.

In a switch statement, the ***case-expression***$_i$ is an arbitrary expression, which is different from C++, where it is a constant. For example, the code

```
switch (TRUE)
{
case f():
      // code 1
      break;
case g():
      // code 2
      break;
default:
      // code 3
}
```

is equivalent to the code

```
if (f() == TRUE)
      // code 1
else
      if (g() == TRUE)
```

```
                // code 2
        else
                // code 3
```

5.6.6 Statement Terminating Loop or Switch

`break;`

Terminates a loop (see 5.6.2, 5.6.3, 5.6.4) or switch (see 5.6.5).

5.6.7 Statement Terminating Loop Iteration

`continue;`

Passes the flow control on to the beginning of the loop (see 5.6.2, 5.6.3, 5.6.4). In the case of a loop with initialization (see 5.6.2), the ***`after-expression`*** is executed.

5.6.8 Jump Statement

```
goto label;
// ...
label:
```

Passes the flow control on to the ***`label`***.

5.7 Additional Statements

Besides the flow control statements (see 5.6), there is a number of additional statements described below.

5.7.1 Statement for Tracing into Standard Output

trace ***expression*** $_1$, ***expression*** $_2$, …, ***expression*** $_n$;

Prints the value of the ***expression*** $_i$ into the standard output:

```
trace "a = ", a, " b(", j, ") = ", b(j), CR;
```

5.7.2 Statement for Expression Type

`typeof(`*`expression`*`);`

Returns a string that is the type name of the ***`expression`*** value:

```
if (typeof(a) == "int")
      // ...
else
      if (typeof(a) == "String")
            // ...
```

5.7.3 Statement for Removal of Global Variables

delete ***name*** $_1$, ***name*** $_2$, …, ***name*** $_n$;

Deletes global variables ***name*** $_i$.

5.7.4 Statement for Halting Program

`stop;`

Switches a program into a stand-by mode to wait for commands from the debugger (see Pluk IDE User Guide). The program execution may be continued from the debugger only.

5.7.5 Statement for Terminating Program

```
end;
```

Terminates the program execution.

5.7.6 Statement for Constant Definition

```
#define name expression
```

Defines the constant **`name`**, equaling the **`expression`** value. The statement is similar to the analogous statement of the C++ preprocessor but the **`expression`** may be just an expression evaluated at the stage of the constant definition.

5.8 Functions

In order to call a function, the user needs the following statements:

```
function-name([expression1, expression2, …, expressionn]);
```

For example:

```
f();       // call of a function with no parameters
g(5, 7);   // call of a function with two parameters
```

The following statements are used in function definitions:

```
new function-name = <| ... |>;
new function-name = {| ... |};
global function-name = <| ... |>;
global function-name = {| ... |};
```

The first function is local; the second is local and compiled into the processor's machine-code. The third function is global; the fourth is global and compiled into the processor's machine-code. A global function differs from a local one in the same way a global variable differs from a local variable (see 5.1). In fact, all of these statements are the definitions of variables, with assigned values of type **`sfunc`** / **`rfunc`**, which is quite similar to conventional variable definitions:

```
new a = 5;
global b = instance Vector(10);
a = <| // ... |>;
b = {| // ... |};
```

A variable is a function if it holds a value of type **`sfunc`** / **`rfunc`**. This allows local functions to be defined inside other functions:

```
global F =
<|
      // ...
      new f =
      <|
      param a, b;
            return a.Name <> b.Name;
      |>;
      v->QSort(f);
      // ...
|>;
```

Since functions are stored in the same namespace as conventional variables (unlike methods), there is no way to specify the types of their parameters (which is possible in the case of methods, see 5.9.2). If it is necessary, a function must check itself for the types of its parameters. The function performs the check by a **`typeof`** statement and throws an error each time the type is invalid:

```
global F =
<|
param s;
     if (typeof(s) != "String")
          Pluk->SetError(ERR_WRONG_PARAMETERS);
     // ...
|>;
```

5.8.1 Statement for Taking Parameters

The **param** statement is used to specify the parameters taken by a function:

param ***name₁, name₂, …, nameₙ;***

nameᵢ is the name of a taken parameter. The **param** statement must be first in the body of a function.

The number of parameters passed at the function call must be no less than the number of parameters to be taken (enumerated in the **param** statement); otherwise, the error **ERR_WRONG_PARAMETER_NUMBER** will be thrown.

5.8.2 Statement for Taking a Variable Number of Parameters

If a variable number of parameters are passed to a function, they could be taken by the **parest** statement:

parest ***name;***

name is the name of a variable containing the parameters taken by the function, but not enumerated in the **param** statement. The variable ***name*** contains a vector each element of which contains a pointer to a parameter if the latter is an object, or a copy of a passed parameter if the latter is not an object (because it is impossible to get a pointer at the parameter). In the function body, the **parest** statement must go on top or follow the **param** statement.

For example, a function may take, in addition to a string, one or two parameters (non-objects, in this case) and assign them to local variables:

```
global F =
<|
param s;
parest pars;
     new x, y;
     if (pars->Len() > 0)
          x = pars[0];
     if (pars->Len() > 1)
          y = pars[1];
     // ...
|>;
```

5.8.3 Value Return Statement

For a function to return a value, the **return** statement is used:

return [*expression*];

The function will return **EMPTY**, if the ***expression*** is absent.

5.8.4 Pass and Return by Reference

By default, a parameter passed to a function is copied into a parameter taken by the function, i.e. the parameter is passed by value. However, the user can arrange passing by reference, i.e. the taken parameter will be not a copy, but a synonym of the passed value. The arrangement may be performed in one of the two following forms:

- by describing a method parameter with the keyword **refer** (see 5.9.2);

- by calling a function / invoking a method using the unary operator `@` (see 5.3).

In the following example, the parameters **`x`** and **`z`** are passed by reference, and the parameter **`y`** by value, into the function **`g`**:

```
g(@x, y, @z);
```

By default, the expression value in the **`return`** statement (see 5.8.3) is copied into the value returned by a function. However, the user can arrange the return by reference, i.e. the returned value will be not a copy, but a synonym of the expression in the **`return`** statement (the expression must be l-value). It can be done by returning the value from a function / method using the unary operator `@` (see 5.3).

In the following example, the function **`f`** returns the value of the global variable **`a`** by value, whereas the function **`h`** returns it by reference:

```
global a;
global f =
<|
     return a;
|>;
global g =
<|
     return @a;
|>;
g() = 5;              // equivalent to a = 5
```

5.8.5 *Passing a Variable Number of Parameters*

To pass a vector as a parameter list (not just a vector) to a function, the unary operator `[]` is used (see 5.3). For example:

```
new p = <<1, "test", TRUE>>;
g(a, [p]);
```

is equivalent to

```
g(a, 1, "test", TRUE);
```

The same without the operator `[]`:

```
g(a, p);
```

is equivalent to

```
g(a, <<1, "test", TRUE>>);
```

The operator `[]` may be applied to the last function parameter only. Most frequently, the operator `[]` is used in a function to pass a variable number of the function's parameters to another function:

```
global F =
<|
param s;
parest pars;
     // ...
     G([pars]);
     // ...
|>;
```

Such application of the operator `[]` is most helpful when defining derived classes (see 5.9.5).

If it is necessary to pass a parameter list by reference, the statement must be as follows:

```
g(a, [@p]);
```

5.9 Classes

To define a class, the programmer should use a **class** statement:

```
class class-name [: parent-name1, parent-name2, …, parent-namen]
{
      [field-name1;
      field-name2;
      …
      field-namem];
[
global:
      [global-field-name1;
      global-field-name2;
      …
      global-field-namek];
]
};
```

The statement defines a class with the name ***class-name,*** which is derived from the classes ***parent-name$_i$*** and contains the fields ***field-name$_i$***, as well as the global fields ***global-field-name$_i$***.

Following is the definition of the class **Point**, containing the fields **x**, **y**:

```
class Point
{
      x;
      y;
};
```

A global field differs from a conventional one in that the former belongs to a class, not an object, i.e. a global field of a class is shared by all objects of the class. In the class **Point** the field **Precision**, specifying the number of decimal digits that will be used in printing a point's coordinates, is defined in the example below:

```
class Point
{
      x;
      y;
global:
      Precision;
};
```

Class names are stored in a different namespace than variable names, so it is possible (and sometimes quite handy) to define a global variable with the same name as the class of the object contained in this variable:

```
global Point = instance Point;
```

5.9.1 Access to Fields

To access an object's field, the binary operator **.** (period, see 5.3) is used. In the body of a method, to access a field of the object, for which the method has been invoked, the unary operator **.** (period, see 5.3) is used. For example:

```
new Point::Point(void) =
<|
      .x = 0;          // access to fields of the object itself
      .y = 0;
|>;
new p = instance Point;
p.x = 100;             // access to fields of the object p
```

```
p.y = 200;
```

Accessing a class global field is similar to accessing a regular object field. For example:

```
new Point::Point(void) =
<|
      // ...
      .Precision = 6;
|>;
```

or

```
new p = instance Point;
p.Precision = 6;
```

Frequently, the user accesses a class global field not via a class object, but by means of the binary operator `::` (see 5.3), where first operand is the class name, second the field name:

```
Point::Precision = 6;
```

The binary operator `.` (period) allows using a pointer as its first operand in accessing a field of the object it points to:

```
new p = instance Point;
new ptr = &p;
ptr.x = 100;                 // access to fields of the object p
ptr.y = 200;
```

5.9.2 Methods

The operator `->` (see 5.3) is used to invoke a method. Additionally, the operator `->` is used in the body of a method to invoke another method for the same object. But in this case, the operator has the `self` keyword (reference to the object the method has been invoked for) as its first operand:

```
new Point::Redraw(void) =
<|
      self->Clear();  // invocation of the method Clear of the
      self->Draw();   // object itself
|>;
new p = instance Point;
p->Redraw();    // invocation of the method Redraw of the object
p
```

The following statements are used to define a method:

```
new class-name::method-name(void) = <| ... |>;
new class-name::method-name(void) = {| ... |};
new class-name::method-name(type1, type2, …, typen) = <| ... |>;
new class-name::method-name(type1, type2, …, typen) = {| ... |};
```

The first method has no parameters; the second has no parameters and is compiled into the processor's machine-code. The third method takes parameters of type ***type$_i$*** (see 5.2); the fourth takes parameters of type ***type$_i$*** and is compiled into the processor's machine-code. For example:

```
new Point::Load(number, number) =
<|
param x, y;
      .x = x;
      .y = y;
|>;
```

As in the case of functions, all of the above statements are the definitions of methods with the empty body and the assigned value of type `sfunc` / `rfunc`.

A class may have several methods with the same names, but different types of parameters. In the class `Point`, for instance, two methods may be defined to move a point:

```
new Point::Move(number, number) =
<|
param x, y;
      .x += x;
      .y += y;
|>;
new Point::Move(object Point) =
<|
param p;
      .x += p.x;
      .y += p.y;
|>;
```

If the keyword **`refer`** appears before the type of a parameter, the parameter will be passed by reference (see 5.8.4). For example:

```
new Point::Move(refer object Point) =
<|
param p;
      .x += p.x;
      .y += p.y;
|>;
```

If a method takes a parameter by value, the parameter may be passed by reference at the point of method invocation by means of a unary operator `@` (see 5.3).

If a method already exists, it may be handled as a variable (for instance, another value could be assigned into it). To handle a method as a variable, the user must indicate the class name, method name and types of parameters; all of these will comprise the method name.

For example, the user can redefine the existing method **`Clear`** of the class **`Point`**:

```
Point::Clear(void) =
<|
      .x = .y = EMPTY;
|>;
```

or add the body of the addition operator of the class **`Point`** to the body of the function **`f`**:

```
f @= Point::+(object Point);
```

The operator **`->`** allows using a pointer as its first operand to invoke a method of the object it points to:

```
new p = instance Point;
new ptr = &p;
ptr->Redraw();  // invocation of the method Redraw of the object
p
```

5.9.3 Constructors and Destructors

Each class may have a special method (or several methods differing in the types of parameters) whose name coincides with the class name. This method is called a constructor. The constructor is invoked each time a class object is being created. Two constructors of the class **`Point`** are defined in the following example:

```
new Point::Point(void) =
<|
      .x = 0;
      .y = 0;
|>;
new Point::Point(number, number) =
<|
param x, y;
      .x = x;
```

```
        .y = y;
|>;
```

Each class may have a special method whose name coincides with the class name preceded by the prefix **~**. This method is called a destructor. The destructor is invoked each time a class object is being deleted. The example below presents the destructor of the class **Point**:

```
new Point::~Point(void) =
<|
        trace "Point is deleted", CR;
|>;
```

To create a class object, the programmer should use the keyword **instance** along with the class name and parameters passed to the constructor. For example:

```
new p1 = instance Point;         // Point::Point(void)
new p2 = instance Point();       // Point::Point(void)
new p3 = instance Point(5, 6);   // Point::Point(number, number)
```

There is a special type of constructor among all constructors that is invoked when an object is being copied. This is the copy constructor:

```
new a = instance Point(100, 100);
new b = a;             // the copy constructor is invoked
```

It is not possible to invoke the copy constructor directly since it is invoked automatically. If the user has not defined the copy constructor in a class, only the object's fields will be copied at the moment of object copying. To define the copy constructor, the user must indicate the keyword **copy** as a parameter list. The copy constructor takes just one parameter: a reference to the object from which copying takes place. For example, the copy constructor (just copying fields) is defined in the class **Point:**

```
new Point::Point(copy) =
<|
param src;
        .x = src.x;
        .y = src.y;
|>;
```

Any object supports delayed deletion. If a certain method is running and the object has been deleted by destructive assignment or the **delete** statement, the object will disappear from the sight of the 'outside beholder'. In the first case, the variable will hold a new value; in the second case, the global variable will disappear along with the object. For all the object methods, running at this moment, however, the object is still accessible under the name **self**. The object will be really deleted (i.e. the destructor will be invoked) only after return from the last object method. For example:

```
new A::F(void) =
<|
        self = instance C;
        trace typeof(self), CR;    // A
        trace typeof(a), CR;       // C
|>;
global a = instance A;
a->F();
```

One more important observation regarding object deletion: the order of deletion of variables (hence, the order of execution of their destructors) is not determined at the time of exit from a function or termination of a program. Sometimes it may result in certain problems, when several global objects try to use each other at program termination. For example, if the main window of a program (global variable **MainWnd**) tries to write something into the configuration file (global variable **Ini**) at the moment of deletion, it may learn that the file no longer exists:

```
new MainWnd::~MainWnd(void) =
<|
      Ini->WriteBool("Window", "Maximized", self->IsMaximized());
      // possibly, variable Ini no longer exists
|>;
global MainWnd = instance MainWnd;
global Ini = instance IniFile("App.ini");
```

To avoid such indeterminacy, the user should control the process of deletion of global objects. She can move the code of the program termination from the destructor into a window method, invoked at the time the window closes (all global objects still exist). The user can set the order of deletion of global objects in this method, knowing the relationships among them (what uses what at the time of deletion):

```
delete Pars;
delete Ini;
```

or

```
Pars = EMPTY;
Ini = EMPTY;
```

5.9.4 User-defined Operators

To define an operator, the programmer uses the same statements as for method definitions. A symbolic notation of the operator is used as a method name. In a unary operator, **self** is the operand. In a binary operator, **self** is the first operand, and the method parameter is the second operand. It is possible, for example, to define the addition operator for objects of the class **Point**:

```
new Point::+(object Point) =
<|
param p;
      return instance Point(.x + p.x, .y + p.y);
|>;
new p1 = instance Point(100, 200);
new p2 = instance Point(200, 300);
new p3 = p1 + p2;
```

5.9.5 Inheritance

Pluk supports class inheritance, including multiple one.

The class **Point3D** may be derived from the class **Point**:

```
class Point3D : Point
{
      z;
};
```

One can define a class that has two or more parents:

```
class ColorPoint3D : Point3D, ColorRef
{
};
```

In the case of multiple inheritance, the object fields inherited from a base class are present in just one copy. For example, an object of the class **D** contains just one object of the class **A**:

```
class A {};
class B : A {};
class C : A {};
class D : B, C {};
```

For the above example, the inheritance graph looks as shown in Fig. 5.9-1.

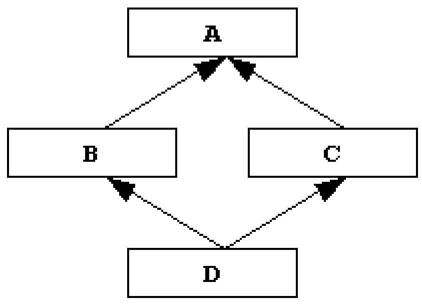

Fig. 5.9-1 Multiple Inheritance of a Class from Two Classes with the Common Parent

A derived class inherits all the methods of a base class. If an invoked method is defined in both a derived and base classes, the method of the derived class will mask the method of the base class (excluding the constructor). It means that all methods (saving the constructor) are virtual, using the terminology of C++. Accordingly, a method of the class, whose object is contained in a variable, will always be invoked if the method has been defined in this same class; otherwise, the method of the nearest base class will be invoked. The last case may throw the error **`ERR_AMBIGIOUS_METHOD_CALL`** if there are several base classes equidistant in the inheritance hierarchy from the given class in which the method is invoked. If it is necessary to invoke a method of just a base class, it could be done by means of the binary operator `::` (see 5.3), where first operand is the base class name, second the method name.

If it is necessary to invoke the base class constructor in the derived class constructor, this could be done by invoking a method with the base class name. For example:

```
class A {};
new A::A(int) =
<|
param n;
      // ...
|>;
new A::Setup(void) =
<|
      // ...
|>;
class B : A {};
new B::B(int) =
<|
param n;
      self->A(n);           // parent constructor is invoked
      // ...
|>;
new B::Setup(void) =
<|
      self->A::Setup(n);    // parent method is invoked
      // ...
|>;
```

All of the above regarding masking of methods of a base class by methods of a derived class is applicable to accessing fields in the case of using the binary operator `.` (period, see 5.3). In the case of using the unary operator `.` (period, see 5.3), however, it is always possible to access a field of the class whose method uses the operator (or the nearest base class having the same field), but not of the class whose object is contained in the variable. If the binary operator `.` (period), with first operand equaling **`self`**, is used instead of the unary operator `.` (period), it will enable access to a field of the class whose object is contained in **self** (or the nearest base class having the same field). For example:

```
class A
{
      x;
};
new A::F(void) =
<|
      .x = 1;          // access to field of class A
      self.x = 2;      // access to field of class B
|>;
class B : A
{
      x;
};
new b = instance B;
b->F();
b.x = 3;               // access to field of class B
```

The constructor of a base class must be invoked directly (if it should be invoked at all) at any point in the constructor of a derived class (even in nested functions). The copy constructor is an exception. The copy constructor is always automatically invoked, first for base classes then for a derived class. It is not necessary, for instance, to invoke the copy constructor of the class **Point** from the copy constructor of the class **Point3D**:

```
new Point3D::Point3D(copy) =
<|
param src;
      .z = 2 * src.z;
|>;
```

The destructor of a base class is also invoked automatically. Its invocation is performed in the reverse order: first for a derived class then for base classes.

At the development of a derived class, a problem arises of redefining a great number of the base class constructors for they will not be invoked automatically. This problem may be solved by means of the unary operator **[]** (see 5.8.5). It is possible to define the constructor of the derived class **B** able to take any parameters and pass them to the constructor of the base class **A** (if the appropriate constructor is absent, an error will be thrown):

```
class B : A {};
new B::B(...) =
<|
parest p;
      self->A([p]);
|>;
```

Meanwhile, additional special constructors may be defined in the class **B**.

5.10 Handling Errors

5.10.1 Local Handling

For local handling of errors thrown by a program, an **onerror** statement is used. It sets up an error catch-point (handler). At the moment of throwing an error, the flow control is passed on to a handler activated at the execution of the last **onerror** statement in a function in the stack. This results in a rollback from the stack functions to the nearest **onerror** statement, accompanied by deletion of local variables. For example:

```
global F =
<|
param x;
     new y;
     y = 1 / x;
     onerror
     {
          trace "Error 1", CR;
          return;
     }
     y = 1 / (x - 1);
     onerror
     {
          trace "Error 2", CR;
          return;
     }
     y = 1 / (x - 2);
     return y;
|>;
global G =
<|
param x;
     new y;
     y = F(x);
     onerror
     {
          trace "Error 3", CR;
          return;
     }
     y = F(x - 10);
     return y;
|>;
G(0);
G(1);          // Error 1
G(2);          // Error 2
G(10);         // Error 3
G(11);         // Error 1
G(12);         // Error 2
```

If there is no **onerror** statement encountered during the execution of the functions in the stack, the rollback from all the functions takes place, and the program slips into a stand-by mode, waiting for events from the operating environment.

It is often necessary to distinguish the errors that result in passing the flow control into the body of an **onerror** statement. Every error has a description string. Additionally, an error may be registered by means of the method **Pluk::CRegError** and assigned a unique integer code. By convention, the name of a constant whose value is equal to the error code should start with the prefix **ERR_**.

The user can throw an error by means of the method **Pluk::CSetError**, using just a description string. At the catch-point, the user must apply the method **Pluk::CGetError** that returns the description string. The user can throw a registered error by means of the method **Pluk::SetError**, using the error code. At the catch-point, the user may apply either the method **`Pluk::CGetError`**, returning the description string, or the **`Pluk::GetError`**, returning the integer error code.

Upon executing the code inside the **`onerror`** statement, the user may continue the rollback from the stack with the same error, using a **`rollback`** statement (instead of a **`return`** statement or code completion). For example, in the case of error-throwing in the method **`File::Read`** invoked from the method **`IniFile::ReadSection`**, two messages are printed: 'Reading file error' and 'Reading section error':

```
class File
{
     // ...
};
new File::Read(int) =
<|
param n;
     onerror
     {
          trace "Reading file error", CR;
          rollback;
     }
     // ...
|>;
class IniFile : File
{
};
new IniFile::ReadSection(object String) =
<|
param name;
     onerror
     {
          trace "Reading section error", CR;
          rollback;
     }
     // ...
     new s = self->Read(n);
     // ...
|>;
```

In addition to this, the user can throw another error inside an **`onerror`** statement. For example, in case of throwing any error in the method **`IniFile::ReadSection,`** the error **`ERR_INI_FILE_READ_ERROR`** is thrown:

```
new IniFile::ReadSection(object String) =
<|
param name;
     onerror
     {
          Pluk->SetError(ERR_INI_FILE_READ_ERROR);
     }
     // ...
|>;
```

Code inside an **`onerror`** statement usually ends with a **`return`** or **`rollback`** statement, or by throwing another error. Otherwise, the code will be completed and the flow control will be passed on to the statement immediately following the **`onerror`** statement. Hence, the code in which the error has been thrown will be executed again. It might be useful if it is necessary to repeat the attempt until it succeeds:

```
global ReadFromFile =
<|
     new fileName;
     onerror
     {
          if (ConfirmationBox("File " @ fileName @
               " not found. Try another file?") != IDYES)
               rollback;
     }
     // Get the filename from the user
     fileName = GetFileNameFromUser();
     // ...
|>;
```

Upon throwing an error, the flow control is passed on to the catch-point activated at the execution of the last `onerror` statement. This may cause certain problems. For example, the user often catches errors within the loop to prevent erroneous iterations from interfering with successful ones. Upon exiting the loop, however, the handler defined in the loop will go on intercepting errors thrown elsewhere in the function. So the user needs to add a handler with a `rollback` statement at the end of the loop:

```
for (new i = 0; i < n; ++i)
{
     onerror
     {
          continue;
     }
     // ...
}
F();       // bad - error in F is caught in the loop body!
onerror
{
     rollback;
}
F();       // ok - error in F is not caught in the loop body
```

If an error is thrown inside an `onerror` statement, the flow control is passed on to the handler activated at the execution of the last `onerror` statement in the function that calls a given function. For example:

```
global F =
<|
param x;
     new y;
     onerror
     {
          trace "Error 1", CR;
          y = 1 / (x - 1);
          return;
     }
     y = 1 / (x - 1);
     onerror
     {
          trace "Error 2", CR;
          y = 1 / (x - 2);
          return;
     }
     y = 1 / (x - 2);
```

```
          return y;
     |>;
     global G =
     <|
     param x;
          onerror
          {
               trace "Error 3", CR;
               return;
          }
          return F(x);
     |>;
     G(1);          // Error 1, 3
     G(2);          // Error 2, 3
```

If a function contains a syntax error, the error will not be caught by any **`onerror`** statement defined in the function, because the latter will not be compiled altogether. In this case, a syntax error will be thrown in the function that calls the function with the error.

5.10.2 Global Handling

The **`handler`** statement is used for global handling of errors thrown by a program. A **`handler`** statement links the registered error code to a global function (or a global class field of type **function**) that will be the global handler:

```
handler error-code, function-name;
```

`error-code` is the error code, ***`function-name`*** is the handler name.

The **`handler`** statement returns the previous handler (or nothing, if a handler has not been defined yet). It is possible to define a new handler and call an old one inside it:

```
     global MyErrorHandler =
     <|
          // ...
     |>;
     handler ERR_MY_ERROR, MyErrorHandler;
     // ...
     global MyErrorHandler2 =
     <|
     parest p;
          if (OldErrorHandler != EMPTY)
               OldErrorHandler([@p]);
          // ...
     |>;
     global OldErrorHandler = handler ERR_MY_ERROR, MyErrorHandler2;
```

The following parameters are passed to the global handler:

Parameters	Description
`ErrorCode`	Error code.
`AddStr`	Error description string.
`FirstLine`	The number of the first line of the function the error has been thrown in (starting from the beginning of the file); equals –1 if unknown.
`CurrLine`	The number of the line in the function the error has been thrown on (starting from the first line of the function); equals –1 if unknown.
`Object`	Passed if the error code equals **`ERR_NON_MEMBER`** (access to a nonexistent field of the ***`Object`***) or **`ERR_METHOD_NOT_FOUND`** (invocation of a nonexistent method of the ***`Object`***).

If the global handler invokes a **`rollback`** statement, the flow control will be passed on to the local handler (see 5.10.1); otherwise, the code that has thrown the error will go on running.

If the global handler for **`ERR_OUT_OF_MEMORY`**, **`ERR_UNDEFINED_CLASS`**, **`ERR_NON_MEMBER`**, **`ERR_METHOD_NOT_FOUND`** does not invoke a **`rollback`** statement, the statement that has thrown the error will be executed again. It allows the error handler to free memory, define a missing class / field / method, substitute the object with an object supporting the missing field / method, etc.

5.11 Handling Events

Pluk enables the user to handle events sent from the operating system, i.e. pass the flow control on to a certain method upon receiving a certain event by an object. In contrast to the global handler, the event handler is a class method. Naturally, the class must be derived from a class, representing the system resource able to generate events. For example, the class **`GWnd`** represents a window able generate window events, the class **`PCom`** represents a communication client / server able to generate communication events (see Standard Library Reference Book).

To link the event code to the class method being the event handler, an **`event`** statement is used:

```
event expression, class-name::method-name;
```

If an object of the class ***`class-name`*** (or a derived class) receives an event whose code equals the integer value of the ***`expression`***, the object's method ***`method-name`*** will be invoked. The ***`expression`*** often equals the value of a constant, a member of a group of prefixed constants. For example, the names of window events constants start with the prefix **`WND_`**. The type of parameters of the method ***`method-name`*** should satisfy the parameters passed at the point of event generation.

Almost in all cases, events are generated by an operating environment (for instance, the window environment). The user, however, can generate an event herself, using a **`simulate`** statement:

```
simulate (expression, object)();
simulate (expression, object)(param1, param2, …, paramn);
```

`object` receives the event whose code equals the integer value of the ***`expression`***, without parameters (first statement) or with parameters ***$param_i$*** (second statement).