Short language reference

The Structure of a B++ Program

Each B++ program consists - as in C - of a *set of functions*. The entry point is usually the `main()` function. In addition to pure functions, the highest structural level can also contain *processing instructions*, *variable definitions* and *class declarations/definitions*.

Unlike in C, global variables can be created within the implementation of functions using the `#defvar` processing instruction.

As in C and C++, it is not possible to define functions locally, i.e. within other functions.

In principle, *every identifier must be declared before the first reference* to it.

The following inevitable "Hello World" example, which is intentionally a little more complicated than necessary, shows the typical program structure:

```
/* B++ Hello World example */

// write a string to console
void printString(var str)
{
   var n = strlen(str);
   for (var i=0; i<n; ++i)
      putc(str[i], stdout);
}

var main()
{
    printString( Hello World - this is B++\n );
    return 0;
}
```

As you can see, there is a great syntactic similarity to C/C++.

First of all, it should be said that B++ is format-free. This means that indentations, blank lines, line breaks, etc. have no syntactic meaning. They only serve to make the source text easier to read. There is no limit to the length of a single line of source text, but to make the text easier to read, it is advisable not to use unnecessarily long lines.

Lines 1 and 3 in the example above contain comments. A function `printString` is then defined, which takes a character string in the `str` parameter and outputs its content character by character to the console.
It uses the predefined functions **strlen** (determines the length of the character string) and **putc** (file output of a character) as well as the also predefined variable `stdout`, which represents a reference to the standard output - usually the console.
In lines 6 and 7, the locally (i.e. within the function) valid variables `i` and `n` are defined using the keyword **var**.

The keyword **var** is used in two other contexts in the example too: In line 11 it indicates that the `main` function returns a value. In line 4 it appears before the `str` parameter in the function header as a replacement for a type identifier, as it would have to be written in C.
In front of the function header in line 4 there is also the keyword `void`, which indicates that the `printString` function does not return a value.

In these contexts, the keywords **var** and `void` are not absolutely necessary and could therefore be omitted. Its purpose is, among other things, also in using automatic documentation generators for B++ sources, such as Doxygen, which are actually designed for the syntax of C or C++.
They can also improve the understandability of the source code. If you mark a function with `void`, the compiler also prevents the return of a value, but if you specify **var**, a value must actually be returned with `return` statement.

The entry function `main` now calls `printString` with the character string to be output as an argument.

The special meaning of the return value of the `main` function should be pointed out here. Since there is no caller of this function within the program, the value is returned to the operating system itself (under DOS and in the Windows console as an 'errorlevel'). This results in the restriction that the return value can only be an integer. For this reason, B++ first checks the type of the return value after the main function ends. If it is an integer value, it is returned, otherwise 0.

The "Hello World" example looks even more C-like when you use the optional static type definitions:

```
/* B++ Hello World example */

// write a string to console
void printString(string str)
{
   int n = strlen(str);
   for (int i=0; i<n; ++i)
      putc(str[i], stdout);
}

int main()
{
    printString( Hello World - this is B++\n );
    return 0;
}
```

Here, instead of the keyword **var**, a type identifier is used (`string` and **int** in the example). In addition to improved readability, a large part of any necessary validity checks can be delegated from the source code - and thus the responsibility of the programmer - to the compiler or the runtime environment, which in the best case leads to shorter source code and better (more stable) programs - in order to price of a slightly slower runtime behavior.

Syntax Elements

This chapter explains the basic syntactic elements that are important for writing B++ programs. This is not a complete formal syntax description. Rather, the most important language elements are presented in comparison to C/C++.

Comments

There are three types of comments in B++:

- **C-style comments:**
  Such comments have the form `/* comment text */`.
  They can span multiple lines but cannot be nested.
- **C++ style comments:**
  These comments are written in the form `// comment text`.
  They reach to the end of the current source code line.

- **Comments starting With @ or #!:**
  This form of comment is used to embed B++ sources in shell scripts (batch files under DOS). For B++ itself it is identical to the C++ comments. However, since the @ character in batch scripts (or the combination #! in Unix shell scripts) initiates a valid command line whose output is simply suppressed, this type of comment can be used to include B++ code in such scripts.

Identifiers

For identifiers in B++ essentially the same rules apply as in C or C++.

Identifiers consist of a sequence of letters (without umlauts and other language specific characters) and numbers or the underscore (_) and the dollar symbol ($). They must not start with a number; the notation of identifiers is case sensitive.

There is no limit on the length of identifiers or the number of significant characters in B++.

Examples of identifiers:

```
MyClass
myValue
_anInternalVariable
p1
$A
```

In general, identifiers should be chosen so that they make it easier to read the source code later. This means in particular that identifiers should, if possible, allow a direct conclusion about their intended use.

Literals

Literals in the narrower sense are constant values noted directly in the source code. This initially includes the notations of values of the various elementary data types. In addition, in B++ there are extended forms for initializing vectors and dictionaries as well as function literals and literally expressions.

Character Literals

A character literal is a single character with a character code between 0 and 255 (up to FFFFH for UNICODE source files), enclosed in quotes, e.g. 'A','x',' ','7','\n'.
The last character in the list above has a special feature. It does not represent a printable character, but a *control character*. The representation of special characters is always started with a backslash (\), often also called an *escape character*.
ollowing table lists the escape codes interpreted by B++ with their meaning.

Escape-Code	Meaning	Character Code (HEX)
\0	NUL-Byte	00
\a	Beep	07
\b	Backspace	08
\f	Page Break	0C
\n	Line Break	10
\r	Carriage Return	0D
\t	Hor. Tabulator	09

\v	Vert. Tabulator	0B
\\	Backslash	5C
\'	Single Quote	27
\"	Double Quote	22

Like C/C++ (and following the same rules), B++ allows the use of escape sequences to notate the code of non-printable characters as character literals. The code can be specified in hexadecimal or octal. So a construct like `'\xFF'` or `'\377'` corresponds to the character with code 255.
It is also possible to specify the value in binary form in an escape sequence. In this case it starts with `\0b`. The character with code 255 could also be specified as `'\0b11111111'`.

In addition, the escape sequences `'\uXXXX'` (UTF-16 characters) and `'\UXXXXXXXX'` (UTF-32 characters) are also interpreted analogously to C++11, where XXXX stands for a sequence of hexadecimal digits.

Character literals (or character types) are integral types and can be treated like integers (type `int`), where the character code corresponds to the numerical value. Accordingly, they can be used in the source code wherever an integer can appear.

String Literals

String literals are sequences of any printable characters or escape codes (see Table above) or escape sequences, enclosed in quotation marks. If the quotation mark itself occurs in a character string, it must be specified as an escape code. String literals can be of any length.

Examples of string literals:

```
"Hello World!"
"He said, 'I'm tired.'"
"Address:\n\tName:\n\tStreet:\n\tCity:\n"
"c:\\prog\\bp2\\bpp  c hello  o hello.bpm"
"This string says 'Beep'\7 on the console!"
```

As in C++, string literals can be concatenated directly. This allows long string literals to be easily spread across multiple lines.

Example:

```
var s = "This String parts "
"belong together."
```

Unlike C/C++, line breaks are allowed within string literals without a backslash (\) at the end of the line. If there is a backslash at the end of the line, the line break in the resulting string is suppressed.

Example:

```
"It's hard to believe:
   This string works
      across multiple lines \
and also remembers the indentations
and line breaks."
```

B++ also allows the use of "raw strings", analogous to C++11. A raw string is a sequence of arbitrary characters whose content is not interpreted.
The notation is as in C++11, i.e. according to the syntax

```
rawstring ::= 'R"' [tag] '(' { any_character } ')' [tag] '"' .
```

The optional *tag* is an arbitrary identifier that is used as an extended end identifier if the character string ')"' occurs in the text itself.

Examples:

```
var s1 = R"(Here \n or \0xFF and " are not interpreted.
Line breaks are also simply adopted.)";

var s2 = R"end( If the combination )" occurs in the text,
you need a tag to mark the end.)end";
```

Integer Literals

Integer literals are representations of whole numbers in decimal, binary, octal, or hexadecimal notation, which can optionally be preceded by a sign (+ or -). Octal numbers - like in C - are characterized by the fact that the first digit is a zero and all other digits are in the range between 0 and 7. Binary numbers start with `0b`, hexadecimal numbers start with `0x`. All other numbers are decimal numbers. By default, integer literals are mapped internally as 32-bit integers (type `int`) and therefore have a value range of -2^{31} to $+2^{31}-1$. If the value of a literal exceeds this range, the internal representation is as a 64-bit integer (`long` type), value range -2^{63} to $+2^{63}-1$.

As in C++, the suffixes U and L or LL (can be combined, case-insensitive) can be appended to the literal to specify the internal representation as an unsigned constant or to have wide of 64 bit.

As in C++11, it is allowed to insert apostrophes into the notation to separate groups of numbers for better readability.

Some possible variants of notations are shown below:

```
17               // 32-bit integer in decimal representation
-234567L         // 64-bit integer in decimal representation
0x7FFFFFFUL      // unsigned 64-bit integer, hexadecimal
021U             // unsigned 32-bit integer, octal
0b1000'1001'0001 // 32-bit integer in binary representation with groups of digits
0x2dff0          // 32-bit integer in hexadecimal representation
```

Extended suffixes:

Starting with version 1.3 of B++, the interpretation of suffixes of integer literals was modified and expanded. The goal is to be able to specify the data type of the resulting value exactly.

The following rules apply in detail:

- The suffix S (or s) forces a signed value. It cannot be combined with the U suffix.
- The suffix L can be followed by a number 0, 1, 2, 4 or 8. The number 0 specifies that the 'smallest' data type that can represent the value should be used; the remaining digits specify the required data type size in bytes.
- If no suffix L is specified, the result is 32 or 64 bits wide, depending on the value.
- If no suffix U or S is specified, the signed type is preferred, which means that the signed type is used if the magnitude of the value can be represented with it, otherwise the unsigned type is used.
- If the data type size is explicitly specified, it is guaranteed that the value can be represented with the resulting type.

- If the suffix S is specified and the value is written in decimal notation, it is guaranteed that the magnitude of the value can be represented in the positive value range of the resulting type.
- Violations of the stated guarantees will result in an error in the translation.

The following example demonstrates some variants of extended suffix notation:

```
17L1      // 8-bit integer in signed decimal representation (implicit)
128L1     // 8-bit integer in unsigned decimal representation (implicit)
17SL1     // 8-bit integer in signed decimal representation (explicit)
17UL1     // 8-bit integer in unsigned decimal representation (explicit)
128SL1    // --> Error: 128 cannot be displayed in the range 0..127
0x80SL1   // 8-bit signed hexadecimal integer (explicit)
0x8000SL0 // 32-bit signed integer
0x8000L0  // 16-bit unsigned integer
```

Note

The extended suffix rules are largely compatible with the previous ones. If the extensions are not used, the result is the same as before - with one exception: If the value of a literal without specifying the suffix U exceeds the positive value range of the **long** type, but not that of ulong, no error is reported, but a silent result of type ulong created.

Floating point literals

Floating point literals are representations of real1 numbers according to the syntax

```
fltlit ::= [+|-] digit[.digit {digit}][e|E [+|-] digit{digit}] .
```

where digit is a decimal digit. By default, the internal representation is double precision (8 bytes, type **double**), which limits the precision to 15 digits of the mantissa and the value range is between $-1.7*10^{308}$ and $1.7*10^{308}$. The suffix F (or f) explicitly specifies the use of single precision (4 bytes, type **float**, value range between $-3.4*10^{38}$ and $3.4*10^{38}$ with 7 digits of the mantissa).

Examples of floating point literals:

```
3.141593      // double
-123.4567E23  // double, expotential representation
123e-4F       // float, expotential representation, no decimal places
-1.234567e-2f // float, expotential representation
2.5E+6        // double, with optional + before exponents
3E14          // double, exponential representation, no decimal places
```

Extended Literals

Variables of the container data types **buffer**, **vector** and **dictionary** can be initialized using literals. Additionally, literals can be used wherever a constant instance of such a container is needed. The notation of these literals is described in connection with the corresponding data types.

Function literals allow the creation of so-called *anonymous functions* that can be assigned as a value to a variable, used as temporary inline functions or similar to the function macros known from C/C++. They are discussed in the chapter about **functions** below.

Literal expressions

B++ allows the creation of *literal expressions*. This means expressions in which all operands themselves represent literals. All operators that are applicable to constant operands can be used in such expressions.

Literal expressions are not evaluated at runtime, but directly by the compiler. Your result is a literal again. They can be used in a program anywhere where a constant value is syntactically permitted.
In addition, they are important in context of conditional translation for evaluating conditions.

Since version 1.7, the concept of literal expressions was expanded so that they are evaluated either directly by the compiler (as before and by default) or by temporarily translating them into bytecode and interpreting the result. The latter method enables all global and static symbols present in the translation context to be used as operands in literal expressions. These include global variables, constants and functions, predefined classes and static members of classes, as well as static local variables and constants in functions.
However, this extension increases compilation time. For each literal expression, bytecode is first generated in the form of a temporary function, and then interpreted.
As this extension is usually needed in rare cases only, it is disabled by default. It can be enabled and disabled using the `#pragma compexpr` processing instruction.

> **Note**
>
> B++'s compiler does not automatically optimize code. This means that expressions are normally evaluated at runtime unless they are used to statically initialize variables - even if their operands are literals (and therefore constant).
>
> With the help of the keyword `litexp` an expression `<expr>` in the form
>
> ```
> litexp(<expr>)
> ```
>
> can explicitly marked as a literal expression and thus its calculation can be forced by the compiler (i.e. during program compilation), which leads to more compact and, above all, faster code.
>
> Since version 1.7, marking can be done in the following form too:
>
> ```
> compexp(<expr>)
> ```
>
> where the keyword `compexp` enforces the extended form of interpretation.
>
> The keywords `litexp` (for the simple form) and `compexp` (for the extended form) override the default behaviour for interpreting literal expressions (`#pragma compexp`) for the respective expression. These keywords can also be used together within an expression to control the interpretation of parts of a more complex literal expression.

Named Literals

Literals can be assigned a symbolic name (an identifier). In this way, symbolic constants or with the help of function literals inline functions can be created. Named literals are defined using the processing instructions `#deflit` or `#literal`.

> **Note**
>
> Because the compiler treats each individual source file as a separate translation unit, the identifiers of named literals are only valid within the source file in which they were defined. If you want to use symbolic constants defined in this way in several translation units, they should be defined in their own (header) file and this file should be integrated into the related source files using `#include` instruction.

Predefined literals

The predefined literals `true` and `false` exist for the `bool` data type to represent its possible values.

The predefined literals `UNICODE`, `WIN32`, `WINCE`, `MSDOS`, `UNIX`, `LINUX` and `BPPI64` provide information about which platform the currently used B++ compiler itself was compiled and whether it is the Unicode version. These literals are of type `int` and have the value 1 (or *`true`*) if the given condition applies at compile time, and 0 (or *`false`*) if not.

The predefined literal `__bplusplus` is of type `uint` and contains 100 times the version number of the compiler when compiling, e.g. 130 for version 1.30.

Also predefined are `PTRSIZE` (type `uint`), which specifies the size in bytes of an address of the system used during translation, and `NPOS` (since version 1.4 and defined as `uint(-1)`) to identify an invalid index value.

Starting with version 1.5 there are also the predefined literals `CHARSIZE` and `WCHARSIZE` (both `uint`), which specify the size of the internal character representation and the type size of `wchar_t`, respectively.

The literals `EDOM` (Domain Error) and `ERANGE` (Range Error) are predefined as the (unfortunately only) standardized error identifiers in the C library. They can occur in connection with numeric functions when invalid argument values used or number ranges are exceeded.

To simplify work with type information, named literals are predefined as symbolic names of the data types supported by B++. Following table contains a complete overview.

Literal	Type ID	Data Type
BVT_NULL	0	**null**
BVT_VOID	1	no value (return type of functions)
BVT_AUTO	2	uninitialized auto-variable
BVT_NIL	3	`null` (return type of functions)
BVT_SBYTE	64	**sbyte**
BVT_BOOL	65	**bool**
BVT_CHAR	66	**char**
BVT_BYTE	68	**byte**
BVT_UCHAR	70	**uchar**
BVT_U8CHAR	71	**u8char**
BVT_SHORT	72	**short**
BVT_USHORT	76	**ushort**
BVT_CHAR16	78	**char16_t**
BVT_WCHAR	79	**wchar_t**
BVT_INT	80	**int**
BVT_INTPTR	81	**intptr**
BVT_UINT	84	**uint**
BVT_UINTPTR	85	**uintptr**
BVT_CHAR32	86	**char32_t**

BVT_LONG	88	**long**
BVT_ULONG	92	**ulong**
BVT_FLOAT	112	**float**
BVT_DOUBLE	120	**double**
BVT_FILE	128	**FILE**
BVT_CODE	129	**function** (native)
BVT_VAR	130	variable/element of dictionary
BVT_OBJECT	160	**object**
BVT_STRONGOBJECT	161	object variable of fixed class type
BVT_CLASS	168	**Class**
BVT_STRING	176	**string**
BVT_BUFFER	177	**buffer**
BVT_CHARBUFFER	178	**charbuffer**
BVT_VECTOR	180	**vector**
BVT_BYTECODE	181	**function** (B++)
BVT_LOCALVARINFO	182	variable info (intern)
BVT_DICTIONARY	184	**dictionary**
BVT_OBJECTREF	224	reference to `object`
BVT_CLASSREF	232	reference to `Class`
BVT_STRINGREF	240	reference to `string`
BVT_BUFFERREF	241	reference to `buffer`
BVT_CHARBUFFERREF	242	reference to `charbuffer`
BVT_VECTORREF	244	reference to `vector`
BVT_BYTECODEREF	246	reference to `function` (B++)
BVT_DICTIONARYREF	248	reference to `dictionary`

The "Literal" column contains the symbolic names, "Type ID" is the associated numerical value, e.g. used by the predefined functions **gettype** and **gettypename**. The words in **bold** in the "Data Type" column can be used as type identifiers for the explicit typing of variables.

Also predefined are the literals `LC_ALL` (0), `LC_COLLATE` (1), `LC_CTYPE` (2), `LC_MONETARY` (3), `LC_NUMERIC` (4): and `LC_TIME` (5) for the categories of the **setlocale** function, `SEEK_SET` (0), `SEEK_CUR` (1) and `SEEK_END` (2) for the origin of the **fseek** function as well as `P_WAIT` (0), `P_NOWAIT` (1), `P_OVERLAY` (2), `P_NOWAITO` (3) and `P_DETACH` (4) for modes of **spawn** function.

In addition, based on the C99 standard, there are predefined literals `__func__` (`string`), `__FILE__` (`string`) and `__LINE__` (`int`) to specify the current method or function name, the file name or the current line number during translation. The values of these literals are formed by the scanner depending on the current position in the source text.

Definitions, Statements and Blocks

As already mentioned, a B++ program essentially consists of class declarations, variable and function definitions at the highest source code level. Additionally there are processing instructions too.

Function definitions consist of a *header*, which contains the name of the code object, as well as a *block* that contains the actual content.
A block is a collection of consecutive *statements* and (optionally) *variable definitions*.
In B++ a block - such as in C/C++ is delimited by curly brackets ({ and }).

Every statement or definition contained in a block must (as in C/C++) be terminated with a semicolon (;). A block can appear wherever a statement is syntactically allowed. This is particularly important in connection with control structures.

Processing Instructions

Processing instructions, unlike all other instructions, are executed by the compiler during source code translation and not by the bytecode interpreter.

Although B++ does not have a preprocessor, they are very similar to preprocessor instructions in C/C++ and are written the same way. In particular, this means that they are not terminated with a semicolon. Like preprocessor instructions in C/C++, processing instructions can also be notated anywhere in a source text.

The processing instructions available in B++ are described in detail below.

The #include Statement

The `#include` processing instruction includes a source file in the currently translated source text as if its contents had been noted at the location of the `#include` instruction.

As in C or C++, there are two forms of notation:

`#include` "*filename.ext*" or
`#include` <*filename.ext*>.

The difference between the two forms of notation lies in the way in which the search for the file to be included is done.
In the first variant, the search is carried out first in the current directory (the directory of the file that contains the `#include` directive), then in all directories of the files that contain higher-level `#include` directives, and finally along the path via the environment variable `BPPINC` specified search path. With the second variant, search in "superior" directories is not done.
In both cases, a relative or absolute directory path can be specified before the file name.

The #use Statement

The `#use` statement includes a precompiled bytecode library in the current program. The symbols contained in the library are then adopted into the program. This means that when compiling a program that includes a library with `#use`, bytecode is created that contains the entire contents of the library.

Usage: `#use` "*filename.ext*"

The file name can be preceded by a relative or absolute path. B++ first looks for the specified file in the current directory (the directory of the file containing the `#use` statement). If the search there is unsuccessful, the directories specified in the `BPPLIB` environment variable are also searched.

Application example:

Let a library be named `tstfunc.bpp` and export a function named \t testfunc.

```
// using #use example   module tstfunc.bpp
```

```
void testfunc(var n)
{
     print("testfunc(",n,") from tstfunc.bpp\n");
}
```

The library is now translated into bytecode:

```
bp2 -c tstfunc -o tstfunc.bpm
```

... and used by the following program:

```
// using #use example   main module
#use  tstfunc.bpm

void main()
{
   testfunc( Hello World );
}
```

Note

If the filename used as an argument contains an asterisk (*), this character is replaced with a build-specific identifier, which can contain the following components:

- u : denotes a Unicode version
- d : denotes a debug version
- _64 : indicates a 64-bit version

This allows the integration of different bytecode modules depending on the build variant of the runtime environment used for execution.

The #import statement

The #import statement allows dynamic B++ extension libraries (DLLs) to be included at compile time.

Usage: `#import` "*filename.ext*"

Here, the file name can be preceded by an absolute or relative path too.
The search for the library to be loaded occurs first in the current directory (the directory of the file containing the #import statement) and then in the search path specified by the `PATH` environment variable.

Note

If the filename used as an argument contains an asterisk (*), this character is replaced with a build-specific identifier, which can contain the following components:

- u : denotes a Unicode version
- d : denotes a debug version
- _64 : indicates a 64-bit version

This allows bytecode modules to be created that may incorporate other extension libraries when executed on a different target platform.

The #deflit, #literal and #undeflit statements

These processing instructions are used to define named literals or to remove the definition of a named literal.

With
`#deflit` **<identifier> <litexpr>** or
`#literal` **<identifier> <litexpr>**
a named literal is defined with the specified identifier. Any literal expression can be used as the assigned value.

The difference between the two forms is that `#literal` only takes effect if there is not already a literal definition under the specified name, while `#deflit` overwrites an existing definition if necessary.

With
`#undeflit` **<identifier>**
an existing definition of a named literal can be removed.

Examples:

```
#deflit PI 3.141593
#literal TRUE 1
#literal FALSE 0
#literal HIWORD function(x) { return (x >> 16) & 0xFFFF;}
#literal LOWORD function(x) { return x & 0xFFFF;}
```

The last two lines define two functions as literals that extract the high order or low order 16-bit word from a 32-bit word, respectively. This form is comparable to function macros. The main difference is that these are "real" (anonymous) functions, not macros that are only expanded when they are used.

The #define and #undef statements

Until version 1.4, `#define` was an alias for `#literal` and `#undef` was an alias for `#undeflit`.

Since version 1.5, these instructions largely correspond to the preprocessor commands of the same name in C/C++, i.e., they are used to define macros.

In particular, variadic macros, i.e. those with a variable number of arguments, are supported in accordance with the C99 standard.
The gcc-specific extension is also supported, which allows the notation of ## before `__VA_ARGS__` to suppress a comma in front of a variable argument list if the variable argument list is empty.
In addition, the concatenation operator ## can also be written after `__VA_ARGS__`. In this case, it suppresses any comma that may appear after an empty variable argument list.

Since version 1.7, support for `__VA_OPT__(x)` was addad (\x is expanded, if `__VA_ARGS__` is not empty. As a converse, there is also there is also `_VA_DEF(x)__`, where `x` is expanded if `__VA_ARGS__` is empty.

Additional notes about macros:

Because B++ does not have a real preprocessor, both the macro names and formal parameters must always be identifiers in the B++ sense. Replacing keywords is not possible.
The macros themselves are not stored internally as text, but as a token list. Accordingly, the replacement is also based on the existing token list. This means that operations performed in lexical analysis are not possible in macro expansion. This applies, for example, to the recognition of numeric literals.

When macros are expanded, any existing parameters are first replaced across the entire macro content, including the handling of variable argument lists, as described above.

The next step handles the remaining concatenation operators (##). The following combinations of tokens (left ## right) are possible for concatenation:

- identifier or keyword ## identifier, keyword, or decimal integer
  Result is an identifier or a keyword.
- special character ## special character, if the combination results in a token
  Result is the corresponding (operator) token.
- string with any other token (left or right of ##)
  Result is a string.

Next, stringify operators (#) are processed. Any token can be used for argument (follow-up token), except for processing instruction identifiers.

In the final step, consecutive character strings are combined to carry out the concatenation that would otherwise be performed by the scanner.

The #defvar statement

Using `#defvar`, initialized global variables can be defined inside and outside of functions.

Use:
`#defvar <varname> <value>`
Where `<varname>` is a valid identifier and `<value>` is the associated value.
The variable value is initialized at compilation time and not at runtime. Therefore, only literals (or literal expressions) can be specified for value.

Examples:

```
#defvar version  2.1.0
#defvar bufSize 2048
```

Note

- In contrast to the definition of named literals, which are only evaluated when they are used, "real" variables are actually created here and initialized with the value of the literal specified as a parameter. The visibility of these variables is not limited to the current translation unit.
- Since version 1.4, the behavior of the `#defvar` statement when a variable already exists has been changed: it is now ignored, whereas previously the existing definition was overwritten.

The statements #ifdef, #ifndef, #if, #elif, #else and #endif

Since version 1.2, B++ supports the conditional translation of sections of source code in a notation that largely corresponds to that of the C preprocessor.

The following table provides an overview of the syntax and meaning of the conditional translation instructions.

Instruction	Meaning
`#ifdef` *`defname`*	Translation if a named literal or a macro *`name`* is defined.
`#ifndef` *`defname`*	Translation if no named literal or macro *`name`* is defined.
`#if` *`literal_expression`*	Translation if *`literal_expression`* equals `true`.
`#elif` *`literal_expression`*	Conditional alternative branch. Translation if *`literal_expression`* equals `true`.
`#else`	Unconditional alternative branch.

`#endif`	End of conditional translation section.

In table above, *`literal_expression`* means a literal expression, which can also contain expanded macros as components.
Within a literal expression the pseudo operator

`defined(`*`name`*`)`

can be used to check the existence of a named literal or a macro. The bracketing of *name* is optional.

The #error and #warning statements

These instructions are used to output a message text during translation.

`#error <message>` or
`#warning <message>`,

where `<message>` is a literal expression that results in a string.

While `#warning` only outputs a message with line number and position, `#error` aborts the translation.

The #pragma statement

The `#pragma` statement is used to control the compiler during translation.
It has the general syntax

`#pragma <`*`function`*`> <`*`parameter`*`>`,

where the following functions (identifier for <*`function`*>) are supported:

Function	Parameter	Description
`cdefine`	`on`, `off`, `push` or `pop`	Determines whether `#define` and `#undef` are used according to the C standard (`on`, default) or are simply aliases for `#deflit` and `#undeflit`, respectively (`off`). The latter is primarily used for compatibility with earlier versions. The current state can be saved to a stack with `push` and restored with `pop`. If `pop` is called without prior `push` (when the stack is empty), this will result in a compiler error.
`compexpr`	`on`, `off`, `push` or `pop`	Determines whether literal expressions are evaluated directly by the compiler (off, default setting) or temporarily translated into bytecode and interpreted (on). The current state can be saved to a stack with `push` and restored with `pop`. If `pop` is called without prior `push` (when the stack is empty), this will result in a compiler error.
`warning`	Output text (literal expression)	Outputs warning message during translation. This corresponds to the behavior of `#warning`.
`message`	Output text (literal expression)	Outputs text message during translation.
`codepage`	number or string (literal expression)	Sets the encoding of the current source file. If an integer is specified as the parameter, this is interpreted as the code page number (according to the Windows standard). Otherwise, an attempt is made to interpret the parameter as

		an encoding identifier (e.g. "iso-8859-1", "Latin-1" or "UTF-8"). By default, the standard operating system character set is assumed.

Marking literal expressions with #()

This processing instruction was required until version 1.2 of B++ to explicitly mark literal expressions. Since version 1.3, the use of the `litexp` keyword is preferred instead, but it is still supported for backwards compatibility and can be used in the form

```
#(<expr>)
```

to mark <expr> as a literal expression. There must be no space between the hash symbol (#) and the opening bracket.

#endsrc statement

The `#endsrc` statement marks the (early) end of a B++ source code, or in other words, any text after `#endsrc` is ignored by the compiler.
The purpose of its application is primarily to integrate B++ sources into shell scriopts (batch scripts).

The folloeing example program is a batch script for MS-DOS, called `graph.bat` that contains B++ source code:

```
@bp2 graph.bat
@goto end
// global settings
#define sizeX 240
#define sizeY 64
#define graphMode 4
#define textMode 7

main() {
        var px=sizeX;
        var py=sizeY;
        setscrmode(graphMode); // switch to graph mode
        var ms = timer();
        var n=0;
        for (var x=0;x<px;x+=2) {
                ++n;
                var x2= px -x;
                line(x,0,x2,py-1,1);
        }
        for (var y=0;y<py;y+=2) {
                ++n;
                var y2= py-1 -y;
                line(0,y,px-1,y2,2);
        }
        ms = timer() - ms;
        var r = n*1000 / ms;
        gets();
        setscrmode(textMode); // switch back to text mode
        print(r, " lines per second\n");
}
#endsrc
:end
```

The program fills a screen area (set with `sizeX` and `sizeY`) with lines and measures the time required for this. At the end the drawing speed (lines per second) is output.

The first two lines contain instructions for the DOS command processor. The preceding @ character suppresses the screen echo under DOS and marks the lines as comments under B++.
The B++ source code ends with the `#endsrc` statement.

The last line belongs to the command processor again - it is the jump label that marks the end of the program.

Variables, Constants and Datatypes

B++ is primarily a language that is usually referred to as *weakly* or *dynamically* typed. This means a little more precisely:

In B++, not the variables, but only the values have a data type.

Optional static typing is achieved by restricting variables to only accept values of the same type as the value with which they were initialized.

Variables

Variables are basically simply storage locations for any value that carry a lfreely selectable identifier.

In B++, variables *must always be defined before they are used for the first time*.

If a variable is defined at the "outermost" program level - i.e. outside of classes or functions - its scope is *global*, i.e. the variable can be accessed from anywhere in the program. This can lead to unwanted side effects in larger programs if you accidentally use the same variable in different places for different purposes.

In order to keep this risk low, there is *no implicit definition of global variables* in B++ by assigning a value to an arbitrary identifier, as is possible, for example, in JavaScript.
A global variable must therefore be defined explicitly - either at the global program level (outside all functions and classes) or with the help of the processing instruction `#defvar`.

In real programs, global variables are rather an exception and are used almost exclusively to exchange data between individual, more or less independent program parts.
In order to create reusable and easily maintainable programs, global variables should be avoided as much as possible and - if they are absolutely necessary - well documented

In B++, the *scope* of local variables (and their lifetime) extends to the *function* within which they are defined. The definition takes place either in the form of a list of local variable identifiers at the end of the function header, separated from the parameter list by a semicolon, or by using the keyword `var` or a type identifier. Below is an example:

```
// using function-level variables
// calculate n!
var fac(n; i, f)
{
        if (n==0)
                return 1;
        for(f=n, i=n-1; i>1; --i)
                f *= i;
        return f;
}

void main()
{
        for (var i=1;i<=10;++i)
                print("fac(",i,")=",fac(i),"\n");
}
```

When developing new programs, variable declaration/definition with `var` (or an explicit type specification) is preferred. It allows local variables to be defined anywhere in the implementation body of a function while initializing them at the same time.

On the one hand, this improves the readability of the code, but on the other hand, it can also improve the efficiency of the program by only executing initializations when necessary, i.e. the relevant program branch is actually reached.

Constants

Constants were introduced with version 1.5 of B++.

Technically, constants are variables with the restriction that after they have been initialized they can only be accessed by reading.
Their definition is analogous to that of variables, with the type or the (then optional) keyword `var` being preceded by the `const` keyword.

Like variables, constants can also be declared globally, as members of classes or temporarily (within the implementation of functions). The same rules apply to scopes and lifetimes as to variables.

Unlike variables, constants must always be initialized immediately.<(p>

There is no equivalent to the `#defvar` processing instruction for constants, and there is also no way to define constants in the function header.

Data Types

In B++, all values occurring in a program are stored in so-called *value objects*. Such an object essentially has a type information as well as the assigned value itself or a reference to it. Accordingly, one can distinguish between *value types* and *reference types*. This distinction is important for use in two ways:

- When assigning to variables, value types are copied completely, whereas in case of reference types only the reference to the data area is copied. Therefore, data changes here affect all variables that refer to the respective value.
  An example:

  ```
  main()
  {
          var s1 = "Hello"; // initialize s1 as string
          var s2 = s1; // assign variable s1 to s2
          s2[1] = 'a'; // modify second character in s2
          print(s1,"\n"); // print value of s1
  }
  ```

  The output here is "Hallo" and not as you might expect "Hello".
  The simple reason is that the variables `s1` and `s2` only contain references to the same data area. A "real" copy of the character string in `s1` could be created, for example, using the **string** function.
- Constants are always copied (deep copy) when assigned to variables, regardless of whether they are a value or reference type.
  Example:

  ```
  main()
  {
          const var s1 = "Hello"; // initialize s1 as string
          var s2 = s1; // assign copy of s1 to s2
          s2[1] = 'a'; // modify second character in s2
          print(s1,"\n"); // print value of s1
  }
  ```

In contrast to first example, \s1 is defined here as a constant whose value is completely copied when assigned to the variable `s2`. Consequently, the output here is "Hello" because the change in the second character only affects the variable `s2`.
The `var` keyword in the declaration of `s1` could also have been omitted.

- The value objects mentioned and with them the data of the value types are created on the stack and cleaned up with it. The situation is different with reference types. Their data area is normally managed using reference counting, i.e. it is released when it is no longer used by any variables. There is also the option of explicit memory management by the user program.

Value Types

Value types in B++ are the numeric types and the type `null`.

Integral data types

These types represent integer values of varying precision. Following table provides an overview of this.

Type	Width (bits)	Value range	Remark
`bool`	8	`false`, `true`	Boolesn value, internally unsigned byte
`sbyte`	8	-128 ... 127	Signed byte
`char`	8	-128 ... 127	Signed single-byte character (like `char` in C)
`byte`	8	0 ... 255	Unsigned byte
`uchar` (**`char8_t`**)	8	0 ... 255	Unsigned single-byte character (like `unsigned char` in C); **`char8_t`** is an alias of `uchar`
`short`	16	-2^{15} ... $2^{15}-1$	`WORD` with sign
`ushort`	16	0 ... 2^{16}	`WORD` unsigned (like `unsigned short` in C)
`char16_t`	16	0 ... $2^{16}-1$	16-bit Unicode characters
`int`	32	-2^{31} ... $2^{31}-1$	`DWORD` with sign
`uint`	32	0 ... $2^{32}-1$	`DWORD` unsigned
`char32_t`	32	0 ... $2^{32}-1$	32-bit Unicode characters
`u8char`	32	sequence of characters	String representation of an UTF-8 character, LSB is first character
`wchar_t`	8, 16 or 32	Like `uchar`, **`char16_t`** or **`char32_t`** (platform dependent)	Wide character (like `wchar_t` in C)
`intptr`	32 or 64	like `int` or \t long (platform dependent)	Representation of an address (signed)
`uintptr`	32 or 64	like `uint` or \t ulong (platform dependent)	Representation of an address (unsigned)
`long`	64	-2^{63} ... $2^{63}-1$	like `long long` in C
`ulong`	64		

		0...2^{64}-1	like unsigned `long long` in C

Unlike in C/C++, signed and unsigned types are not distinguished by a modifier (`signed` or `unsigned`), but rather the distinction belongs directly to the type identifier. The `signed` and `unsigned` modifiers can be used in conjunction with the types `char`, `byte`, `short`, `int` and `long` (and according to the same rules as in C/C++) for C/C++ syntax compatibility, but are not part of a type name.
This corresponds to e.g. a variable declaration

```
unsigned short x
```

exactly the declaration

```
ushort x.
```

Additionally, the integral types in B++, with the exception of `wchar_t`, `intptr`, and `uintptr`, have a fixed width regardless of the compiler and the execution platform. This makes it easier to transfer programs - even in precompiled form - between different platforms.

The `intptr` and `uintptr` data types are used to record platform-specific addresses, usually temporarily. They are therefore 32 bits wide on 16- and 32-bit platforms and 64 bits wide on 64-bit platforms. Internally, `intptr` is treated like a signed integer, `uintptr` like an unsigned integer, so that address calculations can also be carried out.

The basic type of integral data types is the `int` data type. This means that integer literals are interpreted as `int` if their value can be mapped to the `int` type and type is not explicitly marked.

Until version 1.3, B++ did not recognize a "real" Boolean data type; instead, `int` values were used, with the value zero (0) being considered *false* and all other values being considered *true*. However, the reserved identifier `bool` could be used to declare variables to better clarify the intention.
Since version 1.4 `bool` is an independent data type. In numeric operations, `bool` values are treated as unsigned bytes.
The character types were introduced with version 1.5. Previously, individual characters (or character literals) were also represented as `int`. Values of character types are integral types and can therefore be used in numeric expressions, where the character code represents the value.
The type `u8char` was introduced with version 1.9. Previously, the type identifier **`char8_t`** was used for this purpose, but this did not correspond to the type of the same name in C++. Now **`char8_t`** is simply an alias to `uchar`.

The following operations can be applied to values ??of integral data types:

- Assignment to a variable
- Use in numerical expressions
- Use in string expressions
- Bit operations
- numerical comparisons
- Use as a function parameter

Values of integral types can be combined with values of `float` and `double` types in numeric expressions and comparisons without explicit type conversion. In this case, the entire expression is implicitly converted to `float` or `double`. The same applies to the assignment to a statically typed variable of type `float` or `double`.
When using integral values in string expressions (concatenation), there is an implicit conversion to an `int` value, which is interpreted as a character code.

Data type `float`

The `float` data type represents a single-precision (32-bit) floating-point real number. The value range is between -3.4E38 and +3.4E38 with a numerical precision of 6 digits. Smallest amount that can be represented is 1.2E-37.

Allowed operations for the `float` type are:

- Assignment to a variable
- Use in numerical expressions
- numerical comparisons
- Use as a function parameter

Values of type `float` can be combined with values of integral types and `double` type in numeric expressions and comparisons without explicit type conversion. This involves an implicit conversion of the entire expression into the "larger" data type, i.e. when combined with integral types the value is of type `float`, when combined with `double` it is of type `double`. When assigned to a statically typed variable of integral types or `double` type, an implicit conversion to the corresponding type occurs.

Data type `double`

The `double` data type represents a double-precision (64-bit) floating-point real number. The value range is between -1.7E308 and +1.7E308 with a numerical precision of 15 digits. Smallest amount that can be represented is 2.2E-308.

Allowed operations for the `double` type are:

- Assignment to a variable
- Use in numerical expressions
- numerical comparisons
- Use as a function parameter

Values of type `double` can be combined with values of integral types and `float` type in numeric expressions and comparisons without explicit type conversion. This involves an implicit conversion of the entire expression into the "larger" data type, i.e. the type `double`. When assigned to a statically typed variable of integral types or `float` type, an implicit conversion to the corresponding type occurs.

Data type `null`

The `null` data type denotes an uninitialized value. In the source code of a B++ program, data of this type are described by the keyword `null`, which can also be interpreted as a constant and the only possible value of the type `null`. Additionally, a number of predefined functions can return `null`.

Allowed operations for type null are:

- Assignment to a variable
- Use in comparisons
- Use as a function parameter

Values of type `null` can be combined with all other types in comparisons without explicit type conversion. The value `null` is smaller than any other value.

Reference types

All data types in B++ that are not value types are reference types.

Data type `string`

The `string` data type represents a character string. Values of this type can be created explicitly using string literals, by the predefined function **newstring** or the conversion function **string**. In addition, they arise when using the concatenation operator (+) and as the return value of some other predefined functions.

Allowed operations for the string type are:

- Assignment to a variable
- Use in string expressions
- Use in comparisons
- Use as a function parameter
- Access individual characters using the [] operator

> **Note**
>
> Direct access to elements of a (named) string literal using the [ ] operator is not permitted.
> When assigning a string literal to a variable, a copy of the string is created. They can be accessed without restriction.

Data type `FILE`

The `FILE` data type is a reference to an open file. Instances of this type are created with the predefined function **fopen** and released again with **fclose**.

Permitted operations for the FILE data type are:

- Assignment to a variable
- Use in comparisons
- Use as a function parameter (especially for predefined **I/O functions**)

Data type `vector`

The `vector` data type represents data field of fixed (i.e. defined at creation time) or dynamic size. Its elements can be any values, even of different types. Instances of `vector` type are created using the predefined functions **newvector** or **Vec** or **T**.
Alternatively, a value of type `vector` can also be created (initialized) as a literal. The following syntax is used for this:

```
vectorliteral ::= '[' [ literal { ',' literal }] ']' .
```

So a comma-separated list of values, enclosed in square brackets, is noted, where the values themselves are arbitrary literals (including vector literals).

Following example illustrates this:

```
var intvec = [1,2,3,4,5];
var mixedVec = [1,  Hello , 3.14, null];
var matrix2d = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
  ];
```

Valid operations for the vector data type are:

- Assignment to a variable
- Use as a function parameter
- Access individual elements using the operator [ ]

> **Note**
> Direct access to elements of a (named) vector literal using the [ ] operator is not permitted.
> When assigning a vector literal to a variable, a copy of the entire data field is created, which can be accessed without restriction.

Data type `dictionary`

The `dictionary` data type represents an associative array, i.e. a set of key-value pairs, where the keys are character strings and the values are of any type. By using values of the `dictionary` type, arbitrarily complex hierarchical data structures can be built. The keys are always unique within a hierarchy level.

Instances of type `dictionary` are created using the predefined function **newdict**(sorted=0). The *sorted* argument can be used to specify whether the elements should be physically sorted according to the alphanumeric sequence of the keys (value <> 0) or left in the order in which they were added (value = 0).

Alternatively, a value of type `dictionary` can also be created (initialized) as a literal. The following syntax is used for this:

```
dictliteral ::='{' ['!'] [key ':' literal {',' key ':' literal}]'}' .
```

So a comma-separated list of key-value pairs enclosed in curly brackets is noted, with keys and values, each separated by a colon. Character strings (in quotation marks) or valid identifiers can be specified for the keys. The values are arbitrary literals, including vector and dictionary literals.

Since version 1.4 of B++, an exclamation mark (!) can be written after the opening curly bracket, which forces a physical sorting according to the keys - like the sorted argument of the newdict function.

Example:

```
var person = {
        first name:  Hugo ,
        lastname:  Meyer ,
         years : 25
};
```

Permitted operations for the dictionary data type are:

- Assignment to a variable
- Use as a function parameter
- Access to individual elements / creation of new elements using operators **.** and [ ]

> **Note**
> Direct access to elements of a (named) `dictionary` literal using the operator [ ] or the dot operator is not permitted.
> When assigning a dictionary literal to a variable, a copy of the entire data field is created. They can be accessed without restriction.

Data type `buffer`

The `buffer` data type provides a buffer for unstructured binary data. Variables of this type serve as read or write buffers for binary input/output functions (see **I/O functions**) and for transferring data areas to system functions.

Instances of type `buffer` are created explicitly using the **newbuffer**(size, fillByte=0) function. Alternatively, a value of type `string` or `charbuffer` can be passed to the **newbuffer** function. In this case, the buffer is created with a copy of the contents of the argument passed.
In addition, buffer variables can be created from `int`, `float` and `double` values using the conversion function #buffer(arg), for example for passing as reference parameters to system functions.

Since version 1.4 of B++ there is also a literal representation for creating a buffer in the form

```
bufferliteral ::=  <<  litexpr .
```

`litexpr` is a literal expression that returns the absolute or relative path to a file from which the buffer contents are initialized. This allows constant data (resources) to be included directly in the generated program, similar to including source files.
The search for the file determined by `litexpr` follows the same rules as for include files enclosed in quotation marks.

Example using a buffer to include an external file as a resource:

```
#literal textres << "externtext.txt"
void main() {
   print(string(charbuffer(textres)));
}
```

Valid operations for the `buffer` data type are:

- Assignment to a variable
- Use as a function parameter
- Individual elements (bytes) are accessed using the [ ] operator, where the elements are treated as unsigned integer values.

> **Note**
>
> Direct access to elements of a (named) `buffer` literal with the operator [ ] is not permitted.
> When assigning a buffer literal to a variable, a copy of the entire buffer is created. They can be accessed without restriction.

Data type `charbuffer`

The `charbuffer` data type provides a character buffer. The individual elements are 8 or 16 bits wide, depending on whether you are working with ASCII/ANSI or Unicode. Variables of this type serve as buffers for character string manipulations and for passing data areas to system functions.

Instances of type `charbuffer` are created explicitly using the function **newcbuffer**(size, fillChar=0). Alternatively, a value of type `string`, `buffer` or `charbuffer` can be passed to the **newcbuffer** function. In this case, the buffer is created with a copy of the contents of the argument passed.
In addition, instances of this type can be created as a concatenation of individual values of the types `string`, `buffer` and `int` using the constructor function **CBuf**.

Allowed operations for the charbuffer data type are:

- Assignment to a variable
- Use as a function parameter

- Individual elements (characters) are accessed using the operator [ ], where the elements are treated as integer values.

Data type `function`

The `function` data type represents a function that can be called within the program. Internally, a distinction is made between predefined and user-defined functions (i.e. even written in B++)1. From the user's perspective, this distinction is not necessary. Explicit creation of instances of functions is not possible. There is always exactly one instance of a function, which is created for predefined functions when initializing B++ and for user-defined functions during the translation of the function declaration.

Valid operations for the function data type are:

- Assignment to a variable
- Call
- Use as a function parameter

> **Note**
>
> If a function literal that contains at least one static variable in its definition or represents a coroutine (i.e. contains a `yield` statement in its implementation) is assigned to a variable, a new instance of the function is created. In this way, several functions with identical implementation can be created, but strictly speaking they are different functions because their static variables are independent of each other.

Data type `Class`

The `Class` data type represents a class available within the program. Classes are templates for concrete objects, which is why explicit instantiation of classes is not possible. There is always exactly one instance of a class, which is created during the translation of the class definition.

Classes are the only user-defined data types in B++.

Valid operations for the Class data type are:

- Assignment to a variable
- Access to static variables defined within the class
- Calling static functions defined in the class
- Use as a function parameter in the RTTI functions
- Use as a template for creating object instances with the `new` operator

Data type `object`

The data type `object` represents concrete instances of a class in the sense of objects whose structure is formed according to a template given by a class. Objects are created from templates (`Class` type) using the `new` operator.

Permitted operations for the object data type are:

- Calling member functions
- Assignment to a variable
- Use as a function parameter
- Use in expressions if appropriate operators have been defined

Statically Typed Variables

Statically typed variables are declared by explicitly specifying a data type instead of the `var` keyword. This can be one of the predefined type identifiers or the name of a previously defined class.

To Variables defined in this way can only be assigned values that are of the declared type. If possible, an implicit type conversion takes place. The following general conversion rules apply:

- All numeric types (including the `bool` type in this context) can be converted to each other. If the value range of the target type is smaller than that of the source type, dada loss may occur.
  When converting between integer and real values, there may also be a loss of precision in the number representation.
- The value `null` is assignment compatible with all reference types.
- A variable of type `FILE` can be assigned with the integer value 0 (to indicate an uninitialized file variable). Values of type `intptr` are generally assignment compatible to `FILE` too.
- The types `string` and `charbuffer` can be implicitly converted to each other. The conversion creates a copy of the value.
- Any objects (instances of classes) can be assigned to variables of type `object`.
- An instance (object) of a class type can be assigned to a variable of specified class type, if class of object is of same class type as defined for the variable or of e derived class.
- Classes can additionally define conversion operators that allow the implicit conversion of their instances into different data types.

In addition, there are standard functions for explicit type conversion for values into the predefined data types (see **RTTI and type conversion**).

Variables that are declared with the `auto` keyword are a special case. Such variables also have a fixed (static) data type, but - as in C++11 - this is set to the type of the assigned value when they are first initialized. Therefore they must be initialized immediately during definition.

> **Note**
>
> - The compatibility check for assignments takes place at runtime, unless the type of the assigned value is already determined at compile time, which is the case with literals. As a result, type compatibility violations lead to runtime errors that are signaled as exceptions of type `string` and can be handled by the program.
> - The behavior of \auto variables is not entirely identical to that of C++-11. In B++, the type of local (temporary) *auto* variables in functions is determined at runtime.
>   Starting with version 1.8, if a variable is initialized with an object instance, it receives the full type of the initializing object (`BVT_STRONGOBJECT`), similar to C++ behavior. Previously, the type `object` (`BVT_OBJECT`) was used.

Initialization of variables and constants

The initialization of global variables, member variables in a class definition as well as static variables in functions and default values for function parameters takes place at compilation time. Thus, the initial value must be a literal (or the result of a literal expression).

Local (temporary) variables within a function are initialized at runtime. Therefore, the result of any expression can be used for initialization here.

When initializing statically typed variables, the type compatibility of the assigned value is checked during initialization.

The initialization of variables at compile time is done using the assignment operator (=) in the form

```
type varname = initexpr or      var varname = initexpr
```

For statically typed variables that are initialized at runtime - analogous to C++ - also (alternatively) the 'constructor notation', i.e. the form

```
type varname(initexpr)
```

can be used.
If variables of the built-in standard types are initialized in this way, the conversion function into target type is implicitly called. When objects (variables of a specific class type) are initialized, an instance of the corresponding class is created and its constructor is called.

The initialization of constants is exactly the same as that of variables; the distinction is achieved by prefixing the declaration with the keyword `const`.

Note

- Because the scope of constants (as well as variables) defined at the function level is the function, constants should not be defined inside loops. This would lead to a runtime error on the second loop pass.
- A variable of type `FILE` can be initialized in 'constructor notation' too. This is identical to an initialization by assignment with a call to the **fopen** function. A definition
  ```
  FILE fp(filename,mode)
  ```
  corresponds exactly to the form
  ```
  FILE fp = fopen(filename,mode)
  ```
  In both variants, the file opened in this way must (or should) be explicitly closed again with **fclose**.

Implicit initialization

If a variable is not explicitly initialized when it is defined, it implicitly receives a default value. The following general rules apply:

- A dynamically typed variable (type `var`) has an initial value of `null`.
- Variables of type `bool` are initialized with the value `false`.
- Variables of numerical types are initialized with the numerical value zero (according to specific type).
- Variables of type `string` are initialized as an empty string.
- Variables of container types (`buffer`, `charbuffer`, `vector`, `dictionary`) are initialized with an empty container.
- Variables of other reference types (`FILE`, object types) generally have the initial value `null` unless they are strictly typed local (temporary) object variables in functions.
- For temporary strictly typed object variables, an object instance is created and the respective constructor is called without arguments. This assumes that the respective class provides a constructor that can be called without parameters.

Note

> Implicit initialization of constants is not permitted; they must always be explicitly initialized when declared.

Implicit and Explicit Type Conversions

Whenever data of different types is used in an expression, appropriate data conversions (or type conversions) are required. The rules applicable to such conversions in B++ are summarized below.

Within purely numerical expressions, the operands involved are implicitly converted to the type with the largest positive value range, but at least to the `int` type, unless only `bool` type operands occur. The rules that apply here essentially correspond to those of C/C++:

- Values of types `bool` (in mixed expressions), `sbyte`, `char`, `uchar`, `byte`, **`char16_t`**, `short` and `ushort`, **`char8_t`** and `wchar_t` are generally converted to `int`.
- Values of type `wchar32_t` are converted to `uint`.
- For the size of the numeric types applies
  `int` < `uint` < **`long`** < `ulong` < **`float`** < **`double`**.
- The sizes of the `intptr` and `uintptr` types depend on the platform. On 64-bit platforms, they correspond to the types `long` and `ulong`, otherwise they correspond to the types `int` and `uint`.

Strictly speaking, implicit type conversion in numerical expressions only takes place for operators with two operands (infix forms). The conversion affects the operands involved in an individual operation.
For example this would be in an expression

```
2 * 3u - 1.7
```

first the int value 2 is converted to the `uint` type for the multiplication and then the result is converted to the `double` type.

If objects appear as operands in a numeric expression, any existing operator methods of the objects for the relevant arithmetic or bit operator are preferred over a type conversion. If no such operators exist, a conversion operator is searched for in the target type.

For all other non-numeric values, no implicit typecasting is performed in expressions. Here it is the programmer's job to carry out an explicit type conversion if necessary.

Explicit type casts occur in the form

`typename``(`*`expr`*`)`,

where *`typename`* is the target type and *`expr`* is an expression whose result should be converted to the target type.

Predefined functions are available for explicit conversion to the built-in data types - see **RTTI and type conversion**.

For user-defined object types (classes), type conversion operators can be defined as member functions. It is also possible to define explicit conversions to object types as global functions.

Implicit type casts are also performed when assigning values to variables or to arguments of function calls, as long as the assignment target (the variable or the argument) is statically typed. In this case, the functions for explicit type conversion or the corresponding operators of objects are called automatically.

Predefined variables and constants

In B++, as in C, there are three predefined global variables of the `FILE` type that point to the system's standard input and output files:

- ***stdin***: standard input
- ***stdout***: standard output
- ***stderr***: standard error output

These standard files cannot be opened with **fopen** nor closed with **fclose**.

The predefined variable __TMDELAY__ determines how frequently the program checks, while executing, whether any functions connected to installed timers need to be run. The default value is 10.

In addition, there are the following predefined constants (from version 1.5, previously they were variables too):

- `__OSTYPE__`: name of the operating system (`string`) on which the B++ runtime environment is running. Possible values are currently "WIN32" for the Windows desktop version, "WINCE" for Windows CE, "DOS" for MS-DOS systems, "LINUX" for Linux systems or "UNIX" for other UNIX systems.
- `__UNICODE__`: build variant of the runtime environment (`int`).
  It has the value 1 in the Unicode version, otherwise 0.
- `__PTRSIZE__`: Size of an address (pointer) of the runtime environment in bytes (`uint`).
  It thas the value 8 on 64-bit systems, otherwise the value 4.

With the help of these constants, a B++ program can adapt its behavior at runtime to the peculiarities of the environment in which it is being executed - in contrast to the predefined literals, which are evaluated at compilation time. This is important because the bytecode generated by the compiler is independent of the platform on which it is later executed - only the runtime environment is platform-specific. However, it may be that different system-specific functions are available on different target platforms or that certain system properties must be taken into account.

```
if (__OSTYPE__ == "WINCE")
        Window(0,240,320,
           WS_OVERLAPPEDWINDOW|WS_CLIPCHILDREN,WS_EX_OVERLAPPEDWINDOW|
           WS_EX_CONTROLPARENT|WS_MAXIMIZE,title,_mainMenu);
else
        Window(0,640,480,
           WS_OVERLAPPEDWINDOW|WS_CLIPCHILDREN,WS_EX_OVERLAPPEDWINDOW|
           WS_EX_CONTROLPARENT,title,_mainMenu);
```

Example above shows an excerpt from the initialization of the main window of a Windows application implemented with B++. Here, depending on whether the application is running on Windows CE or not, the main window is initialized with different sizes and styles.

Operators

The concept of operators in B++ largely corresponds to that of C or C++. This chapter is therefore limited to a brief reference to the operators available in B++ and existing differences to C++. For classes (or objects) some of the operators can be defined or redefined.

Arithmetic Operators

The arithmetic operators implement the four basic arithmetic operations in conjunction with numerical operands. The addition operator can also be used on character strings (`string` data type). In this case it serves as a *concatenation operator*.

B++ knows the following arithmetic operators:

Operator	Meaning	Example
+ (unary)	positive sign	+a
- (unary)	negative sign	-a
*	multiplication	a * b
/	division	a / b
%	remainder of division (modulo)	a % b
+	addition	a + b
-	subtraction	a - b

Note

- The behavior of the operators depends on the data types of the operands. If both operands have integral data types, an integer operation is carried out, if at least one operand has the type `float` or `double`, a floating point operation.
- If the operands have different data types, a conversion to the type of the larger value range takes place before the operation is executed.
- The result of floating point operations is of type `float` if none of the operands is of type `double`, otherwise of type `double`. The result of integer operations is at least of type `int`.
- If an operand of the addition operator (+) is of type `string`, the operator acts as a concatenation operator. The other operand must then either be of type `string` or of an integral type. In the latter case, the least significant byte (or word in UNICODE builds) of the numeric operand is interpreted as the code of a single character, unless the second operand is of a character type. If the operand is of a character type, it is converted into the internal character set. In non-Unicode builds, there is special handling for characters of type **`u8char`**. In this case, no conversion to the internal character set takes place. This allows UTF-8 strings to be constructed independently of the internal character set.
- The sign operators behave like those in C/C++. In particular, this means that they convert an integral operand with a smaller range of values to a value of type `int`. The unary minus also reverses the sign of numeric values.
- All arithmetic operators can be defined as member functions for object types

.

Comparison Operators

The comparison operators - also known as *relational operators* - are used to compare their (always two) operands.

B++ knows the following comparison operators:

Operator	Meaning	Example
==	equality	a == b
!=	unequality	a != b
>	greater than	a > b
>=	greater than or equal to	a >= b
<	less than	a < b
<=	less than or equal to	a <= b

Note

- If numeric values of different types appear as operands, an implicit conversion is initially carried out into the type with the largest range of values. This can lead to problems due to numerical inaccuracies that occur during type conversion, especially in case of operators == and !=.
- If one of the operands is numeric, the other operand must be numeric too.
  A comparison of a numeric value with a value of object type is also possible,if the object provides a conversion to the corresponding numeric type or a suitable implementation of the `equals` or `compare` methods.
- Character strings (`string`) can be compared with character strings or with `null`.
- Values of all other reference types can be compared with values of the same type (including weak references of the corresponding type) or with `null`. The following applies in detail:
  - An element-by-element comparison is made for the types `buffer`, `charbuffer` and `vector`.
  - For the remaining reference types, the addresses of the data areas are compared, unless they are object types (classes) that implement the `compare` or `equals` methods and thereby define a changed behavior of the comparison operators.
- Values of type `null` are less than all other values.

Logical Operators

The logical operators link truth values with each other. The following rules apply when interpreting values that are not of type bool as truth values:

- Values of type `null` have the truth value `false`.
- Numeric values have the truth value `false` if their absolute value is 0, otherwise `true`.
- All other values (values of reference types) have the truth value `false` if their value is `null`, otherwise `true`.

The following logical operators are defined in B++:

Operator	Meaning	Example
!	NOT	!a
&&	AND	a && b
||	OR	a || b

Note

- The operator ! always returns a result of type `bool`, i.e. `true` or `false`.
- Expressions with the operators && or || are evaluated from left to right. The evaluation stops when the result of the expression is determined. The return value is then the operand that caused the evaluation to be aborted. For this reason, the results of the operators must not be implicitly considered as `bool` values. Using the **bool** conversion function, they can be explicitly converted into the Boolean values `true` or `false`.
- The NOT operator can be defined as an element function for object types (classes).

Bit Operators

The bit operators are only directly applicable to operands of integral types. They implement operations on the individual bits of the operands. In addition, they can be defined for object types (classes).

The following bit operators are defined in B++:

Operator	Meaning	Example
~	bitwise NOT	~a
&	bitwise AND	a & b
|	bitwise OR	a | b
^	bitwise XOR	a ^ b
<<	shift left bitwise	a << 3
>>	shift right bitwise	a >> 3

The value of the right operand of shift operators cannot be negative and should not exceed the number of bits of the left operand.

The behavior of the >> operator depends, as in C/C++, on the type of the left operand. If the left operand is signed, the sign is preserved (for negative values, the most significant bit is set to 1). If the left operand is unsigned, the MSB is always set to 0.

If the operands of the operators &, | or ^ are of different types, they are converted to the larger type before the operation is executed - the rules of C/C++ apply here as well, analogous to the **arithmetic operators**).

Assignment Operators

The assignment operators are used to assign a value to a variable. As in C/C++, assignment is implemented with the = operator, which can be combined with the two-valued arithmetic or bit operators. The table below shows the options:

Operator	Meaning	Example
=	simple assignment	a = b
+=	assignment with addition	a += b
-=	assignment with subtraction	a -= b
*=	assignment with multiplication	a *= b
/=	assignment with division	a /= b
%=	assignment with remainder	a %= b
|=	assignment with bitwise OR	a |= b
&=	assignment with bitwise AND	a &= b
^=	asignment with bitwise XOR	a ^= b
<<=	assignment with left shift	a <<= 2
>=	assignment with right shift	a >>= 2

Note

- The simple assignment operator can be used with all data types. Note that when assigning values of reference types, no copy of the value is assigned, but only a reference to it.
  When using static typing, an assignment is only possible if the value to be assigned has the

same data type as the target of the assignment or can be implicitly converted to this type - see also **Statically Typed Variables**.
Starting with version 1.8 of B++ the simple assignment operator can be redefined for object types (classes) in order to implement a different behavior - see **Defining the Assignment Operator**.

- The compound assignment operators are combinations of the simple assignment operator and an **arithmetic** or **bit operator**, where the assignment target serves as the left operand of the respective operation.
  For example, the expression `a += b` is identical to `a = a + b`.
- The operands of compound assignment operators follow the same rules as the corresponding simple operators.

Increments and Decrement Operators

The increment and decrement operators are unary operators. They increase (operator ++) or decrease (operator –) their operand by the value 1. The operand must be a variable initialized with an integral type or an object type.

Both operators can be used in prefix or postfix notation. A difference between these notations is when they are used within expressions. In prefix notation, the value of the subexpression is equal to the value of the operand after the increment/decrement, while in postfix notation, the value of the operand before the increment/decrement is returned.

The following table illustrates the variants:

Operator	Meaning	Example	Corresponding Expr.
++x	increment (prefix)	b = ++a	a = a+1, b = a
x++	increment (postfix)	b = a++	b = a, a = a+1
--x	decrement (prefix)	b = --a * 7	a = a-1, b = a * 7
x--	decrement (postfix)	b = a-- * 7	b = a * 7, a = a-1

Note

- The increment and decrement operators produce shorter and faster bytecode than their "written out" equivalents (see Table above). They are therefore particularly useful for modifying counting variables in loops.
- The byte code generated by the prefix notation is shorter and therefore faster than that of the postfix notation. Therefore, the prefix notation should be preferred if the special behavior of the postfix variant is not required.
- In compound expressions, these operators should be used with caution, and only when the slightly higher efficiency is really necessary, as undesirable side effects may occur in addition to poorer readability. For example, in an expression
  `c = a || (--b == 1)`
  the variable `b` is never decremented as long as *a* evaluates to `true`.
- The operators can also be defined for object types (classes).
- Due to the implementation, the operators cannot be applied directly to the return value of a function, even if it is an object that provides suitable operator functions. A true L-value is required for the operand.

Miscellaneous operators

Index Operator

The index operator is expressed by square brackets [ ]. It is used for index-based access to single elements of a container (`dictionary`, `vector`, `buffer`, `charbuffer`) or a `string`.

Usung example:

```
// Operator [] Example
main()
{
        var vec = T(1,2,3,4); // creates a vector
        vec[1] = 7;           // assigns 7 to the second element of vec
        print(vec[1],"\n");   // prints 7 to the console
}
```

For use with a dictionary, the operand in the parentheses must be of type `string` (key), in all other cases of an integral type (index).

The operator can also be defined for object types, where the type of the index operand depends on the implementation.

Dot Operator (Property Operator)

The dot is used in many programming languages to refer to elements of structured data types (structures, objects). In B++ it is a predefined operator for the type `dictionary` and its semantics are the same as the index operator. For objects, it is used to access non-static data elements. In addition, the dot operator can be redefined for objects or their classes, e.g. to define pseudo-properties.

The operator has the following general form

```
variable . identifier,
```

where `variable` must be of either `dictionary` or `object` type.

For `dictionary` variables, `identifier` is the identifier (key) of an entry.

For example:

```
// operator . example
main()
{
        auto dict = { a: 1, b: 2 }; // defines a dictionary
        dict.c = dict.a + dict.b;   // adds new value c to dictionary
        print(dict.c);              // prints the value
}
```

Parentheses

Parentheses are operators too. They are used in two contexts:

- as a delimiter for the argument list of functions when declaring and calling
  `myFunc(a, b, c)`
- to control the order in which expressions are evaluated - "We'll evaluate parentheses first!"
  `x = (a + b) * c`

Classes can define the operator as a member function. Instances of these classes can then be called like functions.

Condition Operator

The condition operator has the general form:

```
condexpr ? expr1 : expr2 .
```

The expression `condexpr` (a condition) is evaluated first. Its result is interpreted as a Boolean value. If the result is `true`, `expr1` is evaluated, otherwise `expr2` is evaluated. The result is the result of the whole expression.

Example:

```
// Get minimum
var min(var a, var b)
{
  return a < b ? a : b;
}
```

> **Note**
>
> The return type of the function in the example above is not necessarily known. For example, if `a` is of type `int` and `b` is of type `float`, the condition (a<b) is temporarily converted to `float`. Depending on the result, the function's return value will be of type `int` or `float`.

Comma Operator

The comma operator allows you to concatenate multiple expressions into an expression list. In other words: This operator can be used to record multiple expressions where syntactically only one is expected. The result of the whole expression (i.e. of the list) is the result of the last partial expression.

> **Note**
>
> The comma operator should not be confused with the *delimiter comma* used in parameter- or variable lists.

New, Delete, ::, and -> Operators

These operators are important in connection with classes and objects or for explicit memory management (`delete` operator). They are explained in more detail in context of operator definitions, and are listed in the table below for completeness.

Operator	Meaning	Example
new	Creates an object instance of a class type.	`a = new A()`
delete	Deletes an object instance.	`delete a`

::	Definition of methods,
	Call of class methods
		`A::A() { ... }`
		
		`i = A::getMaxId()`

->	Calls instance methods.	`name = a->getName()`

The :: operator is also used to explicitly reference global symbols within the implementation of functions and members of namespaces.

Unary reference operators & and *.

These two operators are applicable to reference types and allow the conversion of a value into a weak reference (operator &) or the conversion of a weak reference into a value (operator *).

A weak reference is a reference to a data object of a reference type that is explicitly excluded from reference counting and therefore from automatic memory management. Its main purpose is to prevent memory leaks caused by cyclic references in data structures. They can also be used to implement explicit memory management.

Note

- Weak references cannot be assigned to statically typed variables.
- The unary dereference operator * can also be explicitly defined for objects (or their classes). The corresponding operator function is called when the operator is used on a "real" object, i.e. not on a weak reference.

Precedence of operators

When multiple operators are used in an expression, there is a question about the order of evaluation. To determine such an order, a ranking (hierarchy) of operators is defined. B++ largely follows here the rules of C/C++.
The following table shows the hierarchy of the operators in B++. Operators with higher priority are evaluated before those with lower priority. Operators of equal priority are evaluated from left to right according to notation.

Priority	Category	Operators
0	comma	,
1	assignment	= += -= *= /= %= &= ~= ^= <<= >>=
2	decision	?:
3	OR (logical)	||
4	AND (logical)	&&
5	OR (bitwise)	|
6	XOR (bitwise)	^
7	AND (bitwise)	&
8	equality	== !=
9	relation	< <= >= >
10	shift	<< >>
11	addition	+ -
12	multiplication	* / %
13	prefix	+ - ! ~ & * ++ -- new delete
14	postfix	() [] . ++ -- ->
15	range	::

Keywords and reserved identifiers

Keywords are terminal symbols in the syntactic sense. This means that they are identified during lexical analysis and assigned to special internal symbols. Within a program, keywords have a fixed (central) meaning. Therefore, it is not possible to create custom identifiers that match the keywords.

Most of the keywords in B++ are borrowed from C or C++ and have an analogous meaning. The following table gives an overview.

Keyword	Category	Meaning
`null`	Type constant	Identifier for data type `null` and its only value.
`void`	Type constant	Identification of functions without a return value (instead of a return type)
`do`	Program control	Construction of loop statements
`while`	Program control	Construction of loop statements
`for`	Program control	Construction of loop statements
`break`	Program control	Cancels a loop or selection statement
`continue`	Program control	Jumps to the start of a loop statement
`if`	Program control	Construction of conditional statements
`else`	Program control	Construction of conditional statements
`switch`	Program control	Initial identifier of a selection instruction (case distinction)
`case`	Program control	Identification of a selection value in a selection statement
`default`	Program control	Identifier of the standard branch of a selection statement
`return`	Program control	(early) return from a function
`yield`	Program control	return from a coroutine
`try`	Exceptions	Initiation of a protected block
`catch`	Exceptions	Start of exception handling
`throw`	Exceptions	Throws an exception
`var`	Declaration	Declaration of a variable of dynamic (arbitrary) type
`const`	Declaration	Declaration of a constant, can be used as a modifier together with var or a data type, or modifier of a method declaration; can also be used as a modifier of the return value of a function.
`mutable`	Declaration	Declaration of a member variable that can also be changed for constant objects; can be used as a modifier with var or a data type.
`auto`	Declaration	Declaration of a statically typed variable with type determination by initialization
`function`	Declaration	Declaration of a function literal
`class`	Declaration	Start of a class declaration
`namespace`	Declaration	Start of the declaration of a namespace
`static`	Declaration	Declaration of a class member or a static local variable
`public`	Declaration	Introducing a public section in a class declaration
`protected`	Declaration	Introducing a protected section in a class declaration
`private`	Declaration	Introducing a private section in a class declaration
`new`		Creating an object instance

	Memory management	
`delete`	Memory management	Destroying an object instance or value from a different reference type (weak reference)
`operator`	Declaration	Declaration of an operator function
`litexp`	Declaration	Marking a literal expression
`compexp`	Declaration	Marking a compiled literal expression
`TRON`	Debugging	Turn on trace mode
`TROFF`	Debugging	Turn off trace mode
`TRSTEP`	Debugging	Turn on single step mode

The `TRON`, `TROFF`, and `TRSTEP` keywords, which are written as separate statements in the source code, support elementary debugging at the level of the bytecode generated by the compiler.
In trace mode, which is turned on with `TRON` and turned off with `TROFF`, the program flow is output in textual form at the bytecode level, e.g. to a console when using the command-line version of B++, or to a special output window of a GUI.
`TRSTEP` also turns on trace mode, but also causes the program to stop after each bytecode instruction is executed. The single step operation is terminated when reaching a `TROFF` instruction or by a special keyboard input (ESC in the command line version).
The trace output shows the program addresses, the executed bytecodes, the position of the stack pointer, and the types and values of the affected operands.

In addition, B++ recognizes the following **reserved identifiers** (sometimes called "magic names")

- `this` : Reference to the current object in member functions
- `self` : Reference to the current function (method)
- `dtor` : Alias identifier for the destructor of a class
- `__cloned` : Initialization method that is automatically called after copying an object
- `defined` : Pseudo-operator to test if a named literal is defined
- OP_CALL, OP_VREF, OP_VSET, OP_PREF, OP_PSET, OP_NEG, OP_PLUS, OP_INC, OP_PIC, OP_DEC, OP_PDEC, OP_ADD, OP_ADD_R, OP_SUB, OP_SUB_R, OP_MUL, OP_MUL_R, OP_DIV, OP_DIV_R, OP_REM, OP_REM_R, OP_BOR, OP_BOR_R, OP_BAND, OP_BAND_R, OP_XOR, OP_XOR_R, OP_SHL, OP_SHL_R, OP_SHR, OP_SHR_R, OP_NOT, OP_BNOT, OP_UNREF
  Special function names for defining operators

The reserved identifiers are not keywords in the sense described above. However, they implicitly have a special meaning in the program and should not be used as general identifiers. The same applies to the names of the predefined **data types** and the identifiers of the **predefined named literals** as well as the **predefined variables and constants**.

Program Control

Any program that is intended to be more than just a series of instructions requires ways to control the flow of the program. A basic distinction is made between *branching based on conditions* and repeating program sections.

Many programming languages also allow unconditional branch statements. Such instructions are not provided in B++ and are therefore not discussed further.

Conditions (if-else)

The **if-else statement**, which has the same syntax and semantics as the C/C++ if-else construct, is used to control program execution depending on a condition:

```
if (condition) statement1 [ else statement2 ] .
```

If the expression *condition* is `true`, *statement1* is executed, otherwise *statement2* is executed. The statements to be executed can be single statements or blocks of statements. In particular, the *if-else statement itself* can be used to formulate nested conditions or case distinctions.

Example:

```
main()
{
        FILE fp = fopen("testfile.txt","rt");
        if (fp) // same as fp != null
        {
                // do something
                fclose(fp);
        }
        else
                print("Could not open testfile.txt");
}
```

Case distinctions (switch-case)

B++ supports case distinction using the ***switch-case statement***, which is based on the corresponding construct in C/C++ in syntax and semantics, and is exactly the same as in JavaScript.

Unlike C++, `case` can be followed by any expression, not just a constant. As long as a comparison with the values after `case` is defined, the expression after switch can be of any type (not only an integral type).

As in C/C++, there is a "fall through", i.e. if a case branch is not exited with `break`, the statements of the following branch are also executed.

The `default` branch is optional. If used, it must appear *after* the case branches.

Example:

```
switch (myVariable)
{
        case 1:
                // do something
                break;
        case 2:
                // do something else
        // fall through
        default:
                // do something else
}
```

Repetitions (Loops)

B++ uses the `while`, `do-while`, and `for` statements to formulate repetitions of sections of code. The syntax and semantics of these statements are the same as the corresponding constructs in C/C++.

The while statement

The **while statement** has the general form

```
while ( condition ) statement
```

The statement is executed (repeatedly) as long as the *condition* returns `true`.

Example:

```
// Prints contents of a file to console
void main()
{
        FILE fp = fopen("testfile.txt","rt")
        if (fp) // same as fp != null
        {
                while(!feof(fp))
                {
                        string s = fgets(fp);
                        print(s);
                }
                fclose(fp);
        }
        else
                print("Could not open testfile.txt");
}
```

The do-while statement

The ***do-while*** statement has the general form

```
do statement while (condition)
```

It is very similar to the `while` statement. Again, the *statement* is executed as long as the *condition* is `true`. The main difference is that the condition is only checked at the end, so the *statement* is always executed at least once.

Example:

```
// Prompt user for input until a blank line is entered
void main()
{
        string s;
        do
        {
                print("Input line: ");
                s = gets();
                // do something
        }
        while (strlen(s) > 0);
}
```

The for statement

The ***for*** statement is the most powerful form of iteration. As in C/C++ (and unlike languages such as BASIC or PASCAL), it is not necessarily used to process a fixed number of loops, but is a generalization or extension of the while statement. The for statement has the following general form

```
for ( init ; condition ; reinit) statement
```

Before the processing of a `for` loop begins, the initialization expression *init* is executed once. The *condition* expression is evaluated before each iteration of the loop. If it returns `true`, the *statement* is executed, otherwise the loop is exited. The *reinit* expression is executed at the end of each loop iteration (i.e., after the *statement* was executed).

Example:

```
// Lists all arguments of command line to console
void main()
{
        auto argvec = getargs(); // gets argument vector
        int n = vecsize(argvec); // gets number of elements
        for(int i=0; i<n; ++i)   // prints each element to the console
                print(argvec[i],"\n");
}
```

Note

The expressions in the head of a `for` statement can also be empty (omitted). In this case, no action is taken for *`init`* or reinit. If *`condition`* is omitted, an infinite loop is created that can only be aborted explicitly with `break` or an exception. The generated bytecode then contains no condition check, which means that an endless loop of the form `for(;;) ...` is processed faster than one constructed with `while(true)...`.

Nested Loops

Loops can be nested. A typical case for using nested loops is to fill a multi-dimensional array, as shown in the following example:

```
// creates and fills a 2-dim array
void main()
{
        int rows = 8;
        int cols = 8;
        // creates a table
        vector array = newvector(rows);
        for (int i=0; i<rows; ++i)
                array[i] = newvector(cols);
        // initialize table
        for (i=0; i<rows; ++i)
                for (int j=0; j<columns; ++j)
                        array[i][j] = rows * i + j;
        // do something
        // ...
}
```

There is no limit to the maximum nesting depth in B++.

The break and continue Statements

The `break` and `continue` statements can be used within loops for additional flow control. With `break`, you can terminate a loop early, i.e., regardless of the actual termination condition. With `continue`, the rest of the current statement block of a loop is skipped and - if the termination condition is not fulfilled - the next loop pass is executed.

Example:

```
// Writes all printable characters from testfile.txt to the console
int main()
{
        FILE fp = fopen("testfile.txt","rt");
        if (!fp)
        {
                print("Error opening file\n");
                return 1;
```

```
        }
        while(true) // loop forever
        {
                if (feof(fp))
                        break;      // abort loop when reaching end of file
                var c = get(fp);
                if ((c < ' ') && (c != '\n'))
                        continue;   // ignore control characters except EOL
                putc(c,stdout); // write character to console
        }
        fclose(fp);
        return 0;
}
```

Error Handling (try - catch - throw)

The error handling concept is based on that of C++. Accordingly, a "protected block" is introduced with the keyword `try`, followed by an exception handling block introduced with `catch`.

A `try-catch` block has the following general form

```
try statementblock catch ( exceptionid ) statementblock
```

Using

```
throw exception
```

an exception can be thrown within an executable program section (a function).
Within a `catch` block, `throw` can be used without specifying an exception to rethrow the already caught exception.
B++ - like C++ - allows the use of any data type to throw exceptions.

Example:

```
try
{
        var myVar = anyFunc();
        if (myVar) < 0)
                throw "myVar must not be negative";
        // do something else
}
catch (e)
{
        print(e, "caught\n");
        throw; // rethrow exception
}
```

Note

- The identifier used for the exception to be handled in the `catch` block is arbitrary and, unlike other local variables, it is visible only within this block (not throughout the enclosing function). It also overrides a local variable of the same name in case of a name conflict.
- Since B++ does not have static typing of exceptions, the definition of several catch blocks in a row does not make sense and is therefore not allowed.

Functions

As already mentioned **above**, functions are the central structuring elements of any B++ program.
Basically, there are two types of functions: *predefined* and *user-defined*. **Predefined functions** are an integral part

of the B++ runtime system. They are available in every program and can be called by the user, i.e. the programmer, but normally theycannot be changed.

The actual behavior of a program is determined by user-defined functions. The entry point `main` - see chapter **The Structure of a B++ Program** - is such a function itself.

Function Definition

A custom (user-defined)function has the general form

```
[returntype\] funcname ( parameterlist [; localvarlist] ) body
```

where *funcname* is the function name, which must be a valid **identifier**). *parameterlist* is a comma-separated list of named function parameters, and *localvarlist* defines the names of other local variables valid within the function (also a comma-separated list). Finally, *body* is a statement block containing the body of the function. Both *parameterlist* and *localvarlist* are optional. If *localvarlist* is omitted, the semicolon used to separate the lists can also be omitted.

The names of the formal parameters listed in *parameterlist* can optionally be preceded by a data type and the `const` modifier (to define a constant parameter). When a parameter is defined as constant, the function can only have read access to its value.

A return type can optionally be specified before the function name, using one of the keywords `var`, `null`, `void`, or a specific type name. In this case, `void` prohibits the return of a value, while the specification of `var`, `null`, or a type name requires that a `return` statement with a return value is actually used.
The keyword `var` or the specification of a type identifier is optionally allowed before the named parameters. If a type identifier is specified here, the corresponding argument is statically typed.

Local variables can be defined not only in the list in the function header, but also anywhere in the function body using the `var` keyword or by specifying a type identifier or the `auto` keyword (to define a statically typed variable). This also allows (or, in the case of `auto`, forces) immediate initialization of the variables when they are defined.
Variables defined in the function body are also visible within the entire function, not just the statement block surrounding it.
Variable definition must be done in the source code before using a local variable.

Named parameters are treated as local variables within the function body. Global identifiers are visible within a function unless they are covered by local identifiers of the same name. In the case of coverage, global variables can be referenced using the prefixed `::` operator.

Unlike most other scripting languages, B++ does not *allow implicit definition of global variables by simply assigning a value to an arbitrary identifier*. This definition avoids the risk of unintentionally creating global variables through typing errors. If it is actually intended to create a global variable within a function body, this must be done explicitly using the **#defvar** processing instruction.

Unlike C/C++, B++ functions strictly speaking *always* return a function value. If a function is exited with `return` followed by an argument, the value of that argument is the function value. Otherwise, the value is undefined - it is the value currently at the top of the stack.
When the return type of a function is explicitly specified, the type of the function value is checked after the function is called. If necessary and possible, an implicit conversion to the required return type is performed.

Examples of function definitions:

```
// parameterless function without return value
void sayHello()
{
   print( Text from function body\n );
}

// named parameter, local variables and return value
var factorial(n; i, result)
{
   for(result=1, i=n; i>1; --i)
     result *= i;
   return result;
}

// same as above, but using static typed variables declared in body
int factorial(int n)
{
   int result = 1;
   for(var i=n; i>1; --i)
     result *= i;
   return result;
}

/* no named parameters but local variable and return value */
var queryName(;name)
{
   name =  ;
   while (strlen(name) == 0)
   {
      print( Tell me your name:  );
      name = fgets(stdin);
   }
   return name;
}
/* same as above but using static typed variable definition in body */
string queryName2()
{
   string name =  ;
   while (strlen(name) == 0)
   {
      print( Tell me your name:  );
      name = fgets(stdin);
   }
   return name;
}
```

Calling Functions

A function is called by specifying the function name followed by a comma-separated list of parameters. Syntactically, a function call can stand alone as a statement or in place of a value within an expression.

Examples:

```
main()
{
        sayHello();              // single statement - return value is ignored
        var f = factorial(17);   // assigns return value to variable
        var g = sqrt(2.0) * 0.5; // uses function in expression
        print(queryName());      // uses function for parameter of another function
}
```

Note

- When calling a function, you must specify *at least* as many parameters as specified in the function declaration, unless default values for parameters have been defined. If more parameters are specified, the first (i.e. leftmost) parameter values are assigned to the named parameters.
- The number of parameters passed is checked at runtime. If fewer parameters are passed when calling a function than required by the declaration, this does not lead to a compiler error, but to a runtime error.

Like C/C++, B++ supports recursive function calls.

Example:

```
var factorial(var n) // recursive version of the factorial function
{
        if (n > 1)
        return n * factorial(n-1);
        return 1;
}
```

The possible depth of recursion is limited only by the size of the stack.
Since B++ uses a dynamic stack, this is of no practical importance, provided there is enough memory. For very large recursion depths, a limitation comes more from the size of the stack available to the execution environment, since the B++ VM itself processes function calls recursively (in MS-DOS environments the stack has a maximum size of 64 KB, in Win32 it is 1 MB by default).

Emulation of reference parameters

In B++, parameters are always passed to functions by value.

If the passed value itself is of a **reference type**, this does not matter - the referenced value can be modified by the function if necessary, provided the argument was not declared as `const`. Parameters of **value types**, on the other hand, cannot be modified to "outside", because a copy of the value is passed to the function.

If it is necessary or intended to modify a value type variable passed as a parameter within a function, it must be encapsulated in a reference type.

As of version 1.5, B++ provides ox classes for all predefined value types, as well as a box class for each value type in the **boxtypes library**.
The general box class is called **B_var**, the specialized ones have the name of the respective base type with the prefix `B_`. All these classes have a public property "val" that represents the encapsulated value, and the specialized forms also have a conversion operator to the encapsulated type.

Example:

```
#use "boxtypes.bpm"

void incInt(B_int v) {
        ++ v.val;
}

void main() {
        vector v = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
        auto n = v.size();
        for (uint i=0;i<n;++i) {
                B_int vi(v[i]);
                incInt(vi);
                v[i] = vi.val;
```

```
        }
}
```

The example demonstrates the use of the **B_int** type to emulate passing numbers as reference parameters.

The `boxtypes` library can be included in precompiled form using **#use** (as in the example) or included in the compilation process as the source file **boxtypes.bpp** before the source file used.

Variable Parameter Lists and Default Parameters

As mentioned **above**, it is possible to pass more parameters to a function than are specified in the declaration. Taking advantage of this feature, B++ allows the construction of functions with any number of parameters. To access the passed parameters, the predefined function **arg** is used instead of the parameter names. The number of parameters can be get with **argcnt**.
Since version 1.2 of B++, the 'ellipse' (...) can optionally be used as the last element of the formal parameter list of a function to explicitly mark the parameter list as of variable size and achieve a notation analogous to C/C++.

Usage example:

```
var sum(...)
{
        var result = 0;
        var n = argcnt();
        for (var i=0; i<n; ++i)
        result += arg(i);
        return result;
}

void main()
{
        print(sum(0,1,2,3,4,5,6,7,8,9),"\n");
}
```

You can also specify default values for named parameters in a function declaration. Such values are used implicitly when the function is called with fewer than the declared number of parameters. Any literal value (including literal expressions) is allowed as a default value.

See also an example:

```
// creates a string serialization of a vector
string serializeVector(vector vec, string delimiter = ",")
{
        string result = "";
        int n = size(vec);
        for (int i=0; i<n; ++i)
        {
                if (i>0) result += delimiter;
                result += vec[i];
        }
        return result;
}

void main()
{
        // Print comma-separated list of command line arguments
        print(serializeVector(getargs()),"\n");
        // Print command line arguments, each argument on its own line
        print(serializeVector(getargs(),"\n"),"\n");
}
```

This prints the command line parameters specified during the call first as a comma-separated list (using the default value for the delimiter argument), and then line by line.

Static Variables in Functions

Within a function, the keyword `static` can be used to define static variables whose visibility is limited to the function body, but which - unlike 'normal' local variables - have a fixed address (and are not temporarily created on the stack when function is called).

Static (local) variables are defined in the form

```
static [type] name [= value];
```

The variable identifier (*name*) may be preceded by an explicit type specification (*type*). Optionally, the variable can be initialized with an initial value. Initialization occurs at compile time (not at runtime), so value must be a **literal**.

Static variables retain their value between function calls, allowing state information to be used within a function. Typical applications for this are counting and generator functions - for a simple example see below.

```
// Simple ID generator function
uint createID()
{
        static uint _id = 0;
        return +++id;
}
```

Function Pointers

In Section **Data Types** was explained the data type `function`, which is associated with a value of type `function`. In fact, a function can be treated like a "normal" value within certain limits, i.e., it can be assigned to a variable, passed as a parameter to other functions, or used as the return value of a function itself. Since function type values store nothing but a reference to the executable code of the respective function, they are comparable to function pointers in C/C++ and can be used in an analogous way.

Example for defining and using a Function Pointer:

```
void myFunc()
{
        print("myFunc called\n");
}

main(;f)
{
        f = myFunc; // assign myFunc to variable f
        f(); // call function
}
```

Function Overloading

B++ identifies a function only by its name and not (like C++) by its full signature. This means overloading, i.e. defining more than one function with the same name is not really possible. Redefining an existing function will not cause an error, but will completely replace the previously defined function of the same name.

However, "pseudo-overloading" is possible in B++ using the variable parameter lists and function pointers described in the previous sections. Following example shows the principle:

```
var myFunc1 = null;

myFunc(i) {
        print("myFunc()")
        print("myFunc(",i,") called\n");
}

myFunc2() {
        if (argcnt() != 2)
                myFunc1(arg(0));
        else
                print("myFunc(", arg(0), ",", arg(1), ") called\n");
}

main() {
        myFunc1 = myFunc;
        myFunc = myFunc2;
        myFunc(1,2);
        myFunc(1);
}
```

The goal here is to define two functions that can both be called with the name myFunc, where one processes one parameter and the other processes two parameters.

First, the single-parameter variant is defined under the name `myFunc`. The two-parameter variant is initially given a different name (`myFunc2`) and is implemented so that it can process a variable number of parameters. If the number of parameters passed is not two, it calls a function `myFunc1`, otherwise it executes its own code.
The main function then assigns the code from `myFunc` to the (global) variable `myFunc1` and the code from `myFunc2` to `myFunc`.

Anonymous Functions and Function Literals

An anonymous function is a function without an associated name. An anonymous function is always defined using a *function literal* of the form

`function [ [classid] ] [returntype]( parameterlist ; localvarlist ) body`

It differs from a "normal" function definition in that it is introduced by the keyword `function`.
Behind `function`, you can optionally specify a reference to a class (*`classid`*) enclosed in square brackets, where *`classid`* stands for the identifier of a class. This allows you to define *anonymous methods*, i.e. functions that are associated with a class. The type of return value is then specified, again optionally, followed by the specification of the parameters and, if applicable, the local variables and the implementation block.

Anonymous functions do not differ in their behavior from regular functions, but they open up some additional possibilities:

- They can be assigned to local variables within a function (or to member variables of an object), making them visible only locally.
- They can be used like literals, so they can be used anywhere a literal value can appear - especially as part of literal expressions and as the value of default parameters , or within the literal constructors of the vector and dictionary types.
- They can be used in place of a value, e.g. defined inline within a function call or expression and executed immediately.

Example:

```
// Defines replacement for WIN HIWORD/LOWORD macros using named literals
#literal HIWORD function(x) { return (x >> 16) & 0xFFFF;}
```

```
#literal LOWORD function(x) { return x & 0xFFFF;}

// use anonymous compare function for argument of qsort function
var myVec = [4,2,1,7,0,8,9];
qsort(myVec, function (a,b) { return a-b; });

// call anonymous function in an expression
var str = function() {
   var result = "";
   for (var c = getc(); c >= ' '; c = getc()) result += c;
   return result;
}();
print(str);
```

Note

A (compiled) function literal that is called directly as an anonymous function in an expression or as an argument, as in the example above, has a temporary instance created on each call if the function defines at least one local static variable or represents a coroutine (i.e. contains `yield` statement), which takes a certain amount of time.
For very frequent calls, e.g. in a loop, it makes sense to first assign the literal to a local variable and use it in the expression. The above sorting could also be written as follows

```
// use anonymous compare function for argument of qsort function
var myVec = [4,2,1,7,0,8,9];
var cmpfunc = function (a,b) { return a-b; };
qsort(myVec,cmpfunc);
```

In this case, however, there are no runtime differences because the comparison function used here does not define any static variables.

Recursive Calls to Anonymous Functions

Since an anonymous function has no name, the question arises how to implement recursive calls. For this, B++ uses the reserved identifier `self`, which can be used within a statement (or expression) to reference the enclosing function itself. It is used like a normal function name.

Example for using `self` for recursion in anonymous functions:

```
// recursively calculate the Fibonacci number
var fib = function int(int arg) {
        return arg < 2 ? arg : self(arg-1)+self(arg-2);
};
print(fib(12)); // prints 144
```

Using `self` for recursive calls is not necessarily limited to anonymous functions, but is also possible in named functions and member functions (methods) of objects.

Coroutines

Starting with version 1.6 of B++, a simple concept of asymmetric coroutines was introduced.

In B++, a coroutine is a special function that exits the function body with `yield` *`[retvalue]`* instead of `return` *`[retvalue]`*.

A coroutine is defined and called like a "normal" function. When you exit it with `yield`, its current state (values of the current parameters, local variables, reference to the following instruction) are stored. The next time it is called, it goes directly to the stored instruction position, where the local variables and, if necessary, the argument values

are restored. The latter are only restored if they are used again, i.e. if no values were set at the time of the call - so the saved values become default values for optional arguments and replace the original default values if necessary.

Following example demonstrates the behavior of a coroutine:

```
int cofunc(int i=0) {
        int j = i;
        while (j>0) {
                print (__func__," ",i,"\n");
        yield j--;
        }
        return j;
}

int main() {
        for (int j=0;j=cofunc(j+10);)
                print(j,"\n");
        return 0;
}

/* prints following:
cofunc 10
10
cofunc 20
9
cofunc 19
8
cofunc 18
7
cofunc 17
6
cofunc 16
5
cofunc 15
4
cofunc 14
3
cofunc 13
2
cofunc 12
1
*/
```

A simple backward counter is implemented in this example.
The coroutine `cofunc` receives a start value as a parameter, which is initially assigned to the local (temporary) variable *`j`*.
The `while`-loop outputs the function name and call parameters. Then it returns value of *`j`* using `yield` keyword and decrements *`j`*. If j==0, `return` is used. instead of `yield` to exit.
The `main` function calls `cofunc` ten times, each time with an incremented argument.
As you can see from the output, the newly passed argument is used in the repeated calls, but the cached value of variable *`j`* is used internally.
When called repeatedly, `cofunc` is continued behind the `yield` statement, here at the end of the `while` loop.

Note

- Like all functions, coroutines can be used as member functions of objects and/or defined as and/or defined as a function literal.
- Because a coroutine has only one place to cache its state (this memory behaves like a hidden static variable), coroutines cannot be called recursively and cannot call each other.

You can work around this limitation by creating a new instance of the coroutine before each recursive call. This can be done either by using the **clone** function, or by defining the coroutine as a function literal and assigning that literal to a variable before calling it.

- If a coroutine is defined with named parameters but no default values, the corresponding parameters must also be specified when it is called repeatedly. Otherwise a compiler error will occur. This is because calling a coroutine is no different to calling a normal function for the compiler, and a coroutine is not an independent data type in B++.
- When a coroutine is exited with return (even implicitly at the end of the body), it temporarily loses its coroutine property, i.e. no call state is cached and calling it again behaves like calling a normal function.
- Calling a coroutine generally takes significantly more time than calling a normal function, because the state must be saved and restored at each call. Therefore, coroutines should only be used where they are really useful, i.e. if they simplify the code.

Object Oriented Programming

B++'s approach to object orientation is simpler than C++'s, even though they are syntactically similar. Nevertheless, the basic paradigms of object-oriented programming - data encapsulation, inheritance and polymorphism - are taken into account.
There are also additional options, such as the construction of partial classes, the subsequent redefinition of member functions, and the definition of operators on objects.
The following sections describe the principles and options.

Structure of a class

The following example shows the structure of a class in B++.
It shows the beginning of a fictitious class library with a root class `MyBaseClass` and a derived class `MyDerivedClass`. All object instances of this library should have a unique identifier (`_ID`), the assignment of which is the sole task of the base class.

```
// Declaration of MyBaseClass
Class MyBaseClass
{
   MyBaseClass();
   uint getID() const;
   uint _ID;
   static uint createID();
   static uint _maxID = 0; // ID counter, initialized to zero
}

// Implementation of MyBaseClass
MyBaseClass::MyBaseClass() { _ID = createID(); }
uint MyBaseClass::getID() const { return _ID; }
uint MyBaseClass::createID() { return ++_maxID; }

// Declaration of MyDerivedClass
class MyDerivedClass : MyBaseClass
{
   MyDerivedClass(string name = "", var value = null)
   string getName() const;
   void setName(string name);
   var getValue() const;
   void setValue(var value);
   // member variables
   string _name;
   var _value;
}
```

```
// Implementation of MyDerivedClass
MyDerivedClass::MyDerivedClass(var name = "", var value = null)
{
   MyBaseClass();
   _name = name;
   _value = value;
}
var MyDerivedClass::getName() const { return _name; }
void MyDerivedClass::setName(var name) { _name = name; }
var MyDerivedClass::getValue() const { return _Value; }
void MyDerivedClass::setValue(var value) { _value = value; }
```

At first glance, the class declarations and implementations are similar to what you are used to in C++. However, there are a number of important differences:

Visibility of elements

In B++, member and class functions as well as class variables (`static`) are implicitly `public` without explicit access specifiers.
Member variables are implicitly *protected*, i.e. they are only visible in the declaring class and its derivatives.

The keywords `public`, `protected`, and `private` are available to explicitly specify access, and the same rules apply as in C++.

Unlike C++, there are no `friend` declarations, so there is no way to override the access rules for individual "foreign" classes or functions.

No multiple inheritance

Unlike C++, a B++ class has at most one base class. There is also no interface concept like in Java or C#. Inheritance itself is always public, i.e. all public members of the base class also become public members of the derived classes.

Declaring variables and methods

In B++ classes, declaration of member functions in the class declaration is optional. For example, `MyDerivedClass` in the example above could have been declared as follows:

```
class MyDerivedClass : MyBaseClass
{
        string _name;
        var _value;
}
```

This allows you to add methods to classes later, if necessary.

Member variables and all static members must be declared.

If the `static` modifier appears before a comma-separated list of variables, it applies to all of its elements.

> **Note**
>
> As a result of the optional declaration of methods and the ability to redefine methods later, argument lists (and also default arguments) specified in the later implementation completely replace those listed in the declaration, if any. Therefore, unlike e.g. in C++, standard argument values must also be specified in the method head during implementation - see example at start of this section.

Element constants and constant methods

Element constants are implicitly static, i.e. defined at the class level. The modifier `static` in the declaration is therefore optional and only serves for syntax compatibility with C++. Like static variables of a class, element constants must be listed in the class declaration. They are always initialized at compilation time, which means that an element constant must always be initialized with a literal or a literal expression.

Like in C++, *constant methods* are declared with a `const` following the argument list. Just like in C++, constant methods must always be real member functions (i.e. not `static`).
Within such a method, the `this` reference is considered constant, so normally no member variables can be changed.
An exception to this is, as in C++, member variables that are declared with the `mutable` modifier.

> **Note**
>
> Unlike in C++, calling functions that are not declared as `const` is also possible from constant objects. Errors are generated only when element variables that are not marked as `mutable` are modified. This behavior is necessary in order to be able to continue using existing methods from older versions (without `const` and `mutable`) as well as native methods.

Restricted initialization of class variables

In B++, member and class variables (static members) can be initialized when they are declared, with the initialization taking place at compile time and not at runtime. Therefore, only literals (and not function calls) can be used for initialization. The initialization must always be noted within the class declaration (see `_maxID` in `MyBaseClass` above).

Objects are always dynamically created

All instances of classes (objects) have the internal type `object` and are therefore **reference types**. The only way to create them is using the new operator:

```
var myObj = new MyDerivedClass("number",1000);
```

Technically, this also applies to statically typed object variables that are initialized at definition. The notation

```
MyDerivedClass myObj("anumber",1000);
```

is nothing more than a shortened notation for

```
MyDerivedClass myObj = new MyDerivedClass("number",1000);
```

Therefore, it is not possible to create variables of this type directly after a class declaration, as in C++, which also explains why there is no need for a semicolon after the closing parenthesis of the class declaration (notation of a semicolon is allowed to make documentation tools happy).

It should also be noted that, unlike in C++, assigning to a statically typed object variable for initialization does not construct an object.

```
MyDerivedClass myObj = "anumber"
```

would only lead to an error, because it tries to assign a string to an object variable of type `MyDerivedClass`.

Inline Definition of Classes

B++ also allows classes to be fully or partially defined "inline", i.e. the implementation of a class's methods can be noted directly in the class definition. The following example shows a modification of the **above** example, where access specifiers are also used.

```
// definition of MyBaseClass
class MyBaseClass
{
private:
   uint _ID;
   static uint _maxID = 0;   // ID counter, initialized to zero
public:
   static uint createID() { return ++_maxID; }
   MyBaseClass() { _ID = createID(); }
   uint getID() { return _ID; }
}

// definition of MyDerivedClass
class MyDerivedClass : MyBaseClass
{
private:
   string _name;
   var _value;
public:
   MyDerivedClass(string name = "", var value = null)
   {
      MyBaseClass();
      _name = name;
      _value = value;
   }
   string getName() { return _name; }
   void setName(string name) { _name = name; }
   var getValue() { return _value; }
   void setValue(var value) { _value = value; }
}
```

Note that all elements of a class that are used within an inline implementation must be *declared before they are used*, that is, they must be listed in the class definition before the method.

Inline implementation usually results in a shorter and sometimes clearer notation. The separation of declaration and definition (implementation), on the other hand, is preferable when a class interface is to be provided as a public part of a library.

Constructors and Destructors

A ***constructor*** is a special member function of an object whose main task is to initialize the member variables with appropriate initial values when the object is created.

In B++, as in C++, a constructor has the same name as its class. Since function overloading is not possible, a class can have only one constructor. This constructor must return a reference to the created object. In the original version of **BOB** this meant that a constructor always hatd to be left with the statement

```
return this;
```

The B++ compiler automatically appends the corresponding bytecode to the end of each constructor function, so that the statement only needs to be noted here if the constructor is left prematurely (due to an exit condition).

Unlike e.g. in C++, a constructor in B++ does not automatically call the constructor of an existing base class. Such a call must be coded explicitly. An example to illustrate:

```
class Base
{
   Base();
}

Base::Base()
{
   print(ctor of Base called\n);
}

class Derived : Base
{
   Derived();
}

Derived::Derived
{
   Base();
   print(ctor of Derived called\n);
}

main()
{
   var obj = new Derived();
   // do anything
   obj = null;  // releases object
}
```

Note

- In B++, the constructor can also be called like a normal member function if necessary.
- You cannot create instances of a class that does not have its own constructor. This property can be used to define abstract classes. As an alternative, a constructor can also be declared `protected` to prevent the creation of object instances.

A special type of constructor is the ***__cloned*** method, introduced in version 1.9.
When an object is completely copied, either implicitly when assigning a constant object to a variable or explicitly using the **clone** function, all member variables are usually copied. A "real" constructor call does not occur.
In some cases, however, it may be necessary to initialize the copied object. For example, copies of objects derived from `MyBaseClass` in first example of section **Structure of a class** would have the same ID as the original object. This means that such objects would no longer have unique identifiers.
This problem can be solved by providing a `__cloned` method, which, if present, is called on the copy immediately after it is created.
The `MyBaseClass` class therefore is now modified as follows:

```
// definition of MyBaseClass
class MyBaseClass
{
private:
   uint _ID;
   static uint _maxID = 0;   // ID counter, initialized to zero
public:
   static uint createID() { return ++_maxID; }
   MyBaseClass() { _ID = createID(); }
   uint getID() const { return _ID; }
   void __cloned() { _ID = createID(); }
}
```

The `__cloned` method is expected to be a parameterless member function without any return value. In the example above, it is used to assign a unique identity to the copy of object.

> **Note**
>
> Derived classes can override the `__cloned` method. An automatic call to an inherited method from a derived class - analogous to the constructor - does not occur, but it can be explicitly made in the implementation.

A ***destructor*** is, in a sense, the opposite of the constructor. It is used to free any external resources that an object may had occupied (additional memory, open files, etc.) when it is destroyed.

B++ essentially adopts the concept of destructors from C++. Specifically, this means

- A destructor is declared as a parameterless member function with the class name preceded by a tilde (~).
- The destructor is automatically called just before the object is destroyed, i.e. inside the destructor all elements (including the `this` reference) are still valid.
- At the end of the destructor function, the destructors of all base classes are also called recursively, i.e. the destructor code of the derived class is always executed *before* that of the base class. Classes without destructors are skipped in the class hierarchy.

For clarity, the above example is extended:

```
class Base
{
   Base();
   ~Base();
   getClassName();
}

Base::Base() {
   print(ctor of Base called\n);
}

Base::~Base() {
   print(dtor of Base called\n);
}

Base::getClassName() { return Base; }

class Derived : Base
{
   Derived();
   ~Derived();
   getClassName();
}

Derived::Derived {
   Base();
   print(ctor of Derived called\n);
}

Derived::~Derived {
   print(dtor of Derived called\n);
}

Derived::getClassName() { return Derived; }

main() {
   var obj = &new Derived();
   // do anything
   obj = delete obj;
}
```

Note

- In the example, the unary operator & is used when assigning the derived instance to the variable `obj` to force explicit memory management. This is done only to demonstrate the use of the `delete` operator. Normally, one would use a simple assignment that would automatically free obj when the function exits.
- The `delete` operator always returns `null` in B++. It can therefore be used - as here in the `main` function - as the right expression of an assignment to mark the freed variable as invalid.
- Due to implementation, B++ maps the destructor to a member function with the reserved identifier *dtor*. As an alternative to the C++-compliant notation, this identifier can also be used for the declaration/definition of the destructor. In addition, the destructor can be called with this name like a normal member function.
  Up to version 1.8 of B++, the destructors of the base classes were also called here! This is no longer the case from version 1.9 onwards.

Accessing Member Functions and Variables within a Class

If the implementation of a member function contains function calls or variable accesses, a suitable element is first searched for in the current class and, if necessary, in its base classes. If the search is unsuccessful, the function or variable identifier is assumed to be a global identifier. If the identifier is preceded by the keyword `this` followed by the method call operator (->), the identifier is searched only in the respective object.

The behavior is similar for static elements, but to specify a concrete class, the class name and the scope resolution operator (`::`) are specified before the actual identifier.

In this context, the problem arises that implementing a member function may require access to a global symbol (variable or function) with the same name as an element of the current class.
In B++, as in C++, access to global symbols can be forced using the range resolution operator, as shown in example below.

```
// globally defined function
message() { print("global function message called\n") };

class MyClass
{
   MyClass();
   message();    // member function message
   doAction();
}

MyClass() {}
MyClass::message() { print(function MyClass::message\n);}
MyClass::doAction(;obj)
{
   message();          // call internal message method
   this->message();    // call internal message method,
                       // search in current class only
   ::message()         // call global function message
}
```

Virtual Methods

In B++, there is no "early binding" in the strict sense (Section **below** describes how such behavior can still be achieved.). This means that all methods of a class, even static ones, are virtual. Let's take a look at example of destructor calls, and change its `main` function to the following

```
main()
{
   var obj = new Base();
   print(obj->getClassName(),"\n");
   obj = new Derived();
   print(obj->getClassName(),"\n");
}
```

Here instances of the `Base` or `Derived` classes are assigned one after the other to the same variable (`obj`) and their `getClassName()` methods are called. The first call activates the original version from `Base` and the second call activates the overridden version from `Derived`.

It often happens that a derived class overrides a method of its base class to extend its functionality, but not to replace it completely. Then it is desirable to simply call the contents of the inherited method instead of reimplementing it completely. B++ uses a special syntax for this:

To call the inherited ancestor method within the implementation of an overriding method, the function name is prefixed with `BC_` (for BaseClass).

```
class MyBaseClass
{
   MyBaseClass();
   writeInfo();
}

MyBaseClass::MyBaseClass() {}    // ctor does nothing
MyBaseClass::writeInfo() { print("MyBaseClass::writeInfo() called\n"); }

MyDerivedClass : MyBaseClass
{
    MyDerivedClass();
    writeInfo();
}

MyDerivedClass::MyDerivedClass() {}    // ctor does nothing
MyDerivedClass::writeInfo()
{
   print(MyDerivedClass::writeInfo() called\n);
   this->BC_writeInfo();    // calls inherited writeInfo method
}
var main()
{
   var obj = new MyDerivedClass();
   obj->writeInfo();
   return 0;
}
```

Note

- The inherited ancestor method must be called via the `this` pointer (see example above). Otherwise the compiler would generate an "ordinary" function call.
- The call with the *BC_* prefix refers to the base class of the class that contains the calling method. Analogous to the prefix *BC_*, the prefix ***DC_*** (for defining class) can be used. Here the call refers to the class in which the calling method was defined. It can be used to prevent a self-recursively

> calling method that is overwritten in a derived class and called from there with *BC_* from calling the method of the derived class again in the recursion.

The following example shows the method call within the implementation of methods with and without the use of `this`, as well as the use of `BC_` and `DC_`.

```
class ClA
{
}
ClA::ClA() {}
ClA::test() { print("ClA::test\n"); }
ClA::callTest() {
   print("\n",__func__," called from class ",getclassname(this),"\n");
   print("this->DC_test : "); this->DC_test();
   print("this->test(); : "); this->test();
}

class ClB : ClA
{
}
ClB::ClB() {}
ClB::test() { print("ClB::test\n"); }
ClB::callTest() {
   print("\n",__func__," called from class ",getclassname(this),"\n");
   print("test() : "); test();
   try {
      print("this->BC_test : "); this->BC_test();
   }
   catch (e) { print(e,"\n"); }
   print("this->DC_test : "); this->DC_test();
   print("this->test(); : "); this->test();
   print("this->BC_callTest(); : "); this->BC_callTest();
}

class ClC : ClB
{
}

ClC::ClC() {}
ClC::test() { print("ClC::test\n"); }

void main()
{
   var obj = new ClC();
   obj->callTest();
   obj = new ClB();
   obj->callTest();
   obj = new ClA();
   obj->callTest();
}
```

The simple call to the `test` method (implemented in the `callTest` methods) produces the same result regardless of whether it is prefixed or not. However, an explicit object reference (`this`) is absolutely necessary for the prefix calls.

Extended Syntax of the -> Operator

In B++, the `->` operator is used exclusively to call instance methods of an object and has the general form

objref->methodname(argumentlist)

In addition to this standard form (see examples in **section before**), there is an extended variant of the call:

```
objref->(expr)(argumentlist)
```

Here, instead of the method name, an *expression* is specified that returns either the name of the method to be called as a string or a reference (in the sense of a pointer) to the code of the method to be called.

An expression that returns a string can be used to determine the names of methods to be called at runtime. The behavior of the method call itself is exactly the same as in the standard form; in particular, the prefixes `BC_` and `DC_` (see **above**) can be used too.

In addition, the extended form allows "early binding" of methods, i.e., their non-virtual invocation. This can be achieved by using

```
objref->(classname::methodname)()
```

where `classname` is the name of `objref`'s own class or one of its base classes, and `methodname` is the method to be called.

This means that the method to be called is not determined at runtime (as with virtual calls), but the reference to the method to be executed is integrated directly into the bytecode. Accordingly, the class and the method to be called must be (at least) declared at the time of compiling the calling code.

Method Pointers and Method Literals

Methods of an object (element functions) differ from ordinary functions only in their relation to a class and in the possibility of accessing member variables of objects (instances of the class). Accordingly, references to methods can be assigned to variables or used as arguments of function calls. Variables that have been assigned a reference to a method can be viewed as *method pointers*, analogous to the **function pointers**.

The following example shows the assignment of an element function to a variable and variants of the call.

```
class ClassA
{
public:
  ClassA() {}
  Test(i) { print (__func__,"(",i,")\n"); }
};

main()
{
   var obj = new ClassA();
   var a = ClassA::Test;
   obj->(a)(1);
   a(obj,1);
}
```

The assignment to a variable is done in the same way as for read access to a static variable in a class. The call is made either via an object reference with the **extended form** of the -> operator or like a simple function call, where an object must be passed as an additional (first) parameter.

Methods can also be defined as literals, using the definition syntax

```
function '['classid']' [returntype] '('paramlist [;localvarlist]')' body
```

(see **Anonymous Functions and Function Literals**)

`classid` refers to a class that must already exist (be declared) at the time of the literal definition. Furthermore, all variables belonging to the class that are used within the literal method must be defined - see example below.

```
class A
{
private:
   string _txt;
public:
   A(string txt="hello\n") { _txt = txt; }
}
// defines global variable g initialized with member function literal
var g = function [A] string() { return _txt; }

main()
{
   var a = new A();
   print(a->(g)());
}
```

Note

The example shown here demonstrates a way to temporarily extend existing classes with additional methods. Note that this bypasses the access protection defined for the class members. Therefore, this option should be used carefully and its use should be well documented.

Local Classes

As of version 1.5, B++ allows the 'local' definition of classes, i.e. the definition of classes as elements of another, higher-level class. Typical use cases for this are defining iterators and encapsulating implementation details of custom container classes and other helper classes for implementation.

The following example demonstrates the use of local classes by implementing a simple list class with a forward iterator and an internal element type.

```
class SimpleList
{
   class Element
   {
    public:
       var _value = null;
      Element _next = null;
      Element() {}
   }
   class iterator
   {
   private:
      Element _el = null;
      SimpleList _list = null;
   public:
      iterator(SimpleList list, Element el) {
         _el = el;
         _list = list;
      }
      bool equals(iterator other) {
         if (!other) return false;
         return (addr(_list) == addr(other._list) &&
                 addr(other._el) == addr(_el));
      }
      Element operator*() { return _el; }
      iterator operator++() { if (_el) _el = _el._next; return this; }
      iterator operator++(int) {
         auto res = new iterator(_list,_el);
         return ++res;
      }
      iterator next() { iterator res = clone(this}; return ++res; }
   }
private:
```

```
   Element _start = null;
public:
   SimpleList() {};
   iterator begin() { return new iterator(this,_start);}
   iterator end() { return new iterator(this,null); }
   iterator insert(var value, iterator pos = null) {
      auto e = new Element();
      e._value = value;
      auto iter = begin();
      if (!pos || pos == iter) {
         e._next = _start;
         _start = e;
         return begin();
      }
      auto end = end();
      for (auto it=iter; it != end(); ++it;) {
          auto nxt = it->next();
          if (nxt == pos || nxt == end) {
              auto before = *it;
              e._next = before._next;
              before._next = e;
              return nxt;
          }
      }
      return end;
   }
   iterator erase(iterator pos) {
      auto iter = begin();
      auto end = end();
      if (!_start) return end;
      if (!pos || pos == iter) {
        _start = _start.next;
        return begin();
      }
      for (auto it = iter; it != end; ++it) {
         auto nxt = it->next();
         if (nxt == pos || nxt == end) {
            auto before = *it;
            if (*nxt) {
               before._next = (*nxt)._next;
               (*nxt)._next = null;
            }
            else before._next = null;
            return ++it;
         }
      }
      return end;
   }
}
```

Note

Unlike methods and member variables, local classes are always public.

Local classes are accessed by client code using the scope resolution operator (`::`).

Partial classes

B++ supports partial classes in a similar way to C#, but without using a special keyword.

A *partial class* is a class whose definition is spread over several blocks, which may be in different source files. This opens up a number of possibilities:

- Distribute large classes over several files
- Deal with different aspects of a class in separate sections

- Class content can be made variable, i.e. depending on the context, a concrete class can be composed of different parts.
- Use for forward declaration of classes unknown at compile time of a module
- Existing classes can be extended "later".

For the following example we assume that a program uses a class `Class1`, creates an instance of it and activates a `doAction` method on it.
The class `Class1` itself is finally defined and implemented in another module, which is combined with the main program when called.

First the main program:

```
// doAction example  main module action.bpp

class Class1      // class prototype
{
   doAction();
}

main() {
   var obj = new Class1();
   obj->doAction();
}
```

`Class1` is only defined here as an empty prototype - this is what the compiler requires. A first version of the complete definition and implementation takes place in a module `actcl1_1`:

```
// first version of Class1 module  actcl1_1.bpp

class Class1
{
    Class1();                       // ctor
    doAction();
    var _callCnt;                   // counter for calls of doAction
}
// implementation

Class1::Class1() { _callCnt = 0; }
Class1::doAction()
{
   print(++_callCnt,". call of doAction in module actcl1_1\n");
}
```

Now you can compile the main program and the module into bytecode, and then run the main program using the library module:

```
bp2 -c action -o action.bpm
bp2 -c actcl1_1 -o actcl1_1.bpm
bp2 -r action actcl1_1
```

Without modifying the main program, you can reimplement the class Class1 in another module and use it as an alternative to the above:

```
// second version of Class1 module  actcl1_2.bpp

class Class1
{
    Class1();                       // ctor
    doAction();
}
```

```
// implementation

Class1::Class1() {}                  // ctor does nothing
Class1::doAction()
{
   print("doAction in module actcl1_2 activated\n");
}
```

The compilation and calling is done in the same way:

```
bp2 -c actcl1_2 -o actcl1_2.bpm
bp2 -r action actcl1_2
```

Replacing member functions (Redefinition)

Just as you can replace existing ordinary functions by simply redefining them, this is also possible for functions that are part of a class. The newly defined variant completely replaces the old one. This feature can be used to change the behaviour of a class at a later stage.
The example from the previous section is used to illustrate this. However, the `Class1` is now implemented immediately.

```
// doAction example  main module action.bpp

class Class1
{
public:
   Class1();
   doAction();
}

Class1::Class1() {}
Class1::doAction() { print("Class1::doAction called\n"); }

main()
{
   var obj = new Class1();
   obj->doAction();
}
```

The example should work straight away. Now write a patch module that changes the behaviour of `Class1's doAction` method:

```
// module patch.bpp replaces doAction method of Class1
Class1::doAction()
{
   print("patched version of Class1::doAction called\n");
}
```

Note that the patch module cannot be compiled separately because it does not contain a declaration of `Class1`, but the following call is possible:

```
bp2 action patch
```

Here the patch is compiled after the original program, so `Class1` already exists and the compiler is happy.

Note

- Redefining a method does not change its visibility level (`public`, `protected`, `private`).

- The same rules apply to redefinition as to "normal" implementation of a method outside the class definition. In particular, this applies to the need to re-specify default parameters.

Defining Operators

In B++ it is possible to redefine some operators for objects (or more precisely: their classes), or to define them at all.

An operator is defined indirectly by a class that provides a method (member function) whose name corresponds to one of the reserved identifiers of the form `OP_XXX` (see Chapter **Keywords and reserved identifiers**). Alternatively, a syntax based on C++ with the keyword `operator` and a trailing operator symbol can be used for the definition. This notation is preferred because of its common use in the examples in the following sections.

Note

- Even with `operator` notation, the compiler creates methods of the form `OP_XXX`. Therefore, explicit operator calls can only be made using these method names.
- Like all other methods, operator functions return a value. Therefore, an operator definition can optionally be preceded by a return type.
- Unlike C++, operators in B++ cannot be defined as global functions, but only as (non-static) member functions of classes.
- Like all other methods, operator functions can be declared c\ const if they do not change any properties of the calling object (exception: member variables declared as `mutable`). The arguments of operator functions may also be declared as `const`.

The rules for defining individual operators or groups of operators are explained below.

Defining the Function Operator

When a class defines the function operator `()`, its instances become *function objects* (sometimes called *functors*). Such objects can be used in the program like functions.

The operator is defined in the form

[returntype] `OP_CALL(`*argumentlist*`)` or
[returntype] `operator()(`*argumentlist*`)`,

where *argumentlist* is a comma-separated (and possibly empty) list of argument definitions to which all the rules of an argument list of "normal" functions apply.
In particular, argument lists of variable length or default parameters can also be used - see Section **Variable Parameter Lists and Default Parameters**.

Following example illustrates the procedure using a class whose instances are used as functions that write any number of strings to a file.

```
// operator () example
class StringWriter
{
public:
   StringWriter(string file);
   ~StringWriter();
   uint operator()();
private:
```

```
   FILE _fp;
}

StringWriter::StringWriter(string file)
{
   _fp = fopen(file,"wt");  // opens text file for write
}

StringWriter::~StringWriter()
{
   fclose(_fp);
}

uint StringWriter::operator()()
{
   uint n = argcnt();
   for (uint i=1;i<n; ++i)
     fputs(arg(i),_fp);
   return n;
}

main()
{
   StringWriter writer("txtfile.txt");
   writer("Hello","World");
}
```

Definition of the Index Operator

The index operator [ ] is normally used for indexed access to single elements of a string or container (types `vector`, `buffer` or `charbuffer`) or for access to entries of an associative container (`dictionary`) via a key. If the operator is defined for objects, two member functions are required - one for read access and one for write access. If only one of the functions is implemented, the operator works in one direction only.
Unlike for the standard variants, the index argument does not necessarily have to be an integer or a string.

The operator is defined in the form

[returntype] OP_VREF(*index*) *[const]* or
[returntype] operator[](*index*) *[const]*

for read access or

[returntype] OP_VSET(*index, value*) or
[returntype] operator[]=(*index, value*) or
[returntype] operator`var`

for write access.

> **Note**
>
> The last notation is mainly for compatibility to older versions of B++. The keyword `var` is used here for syntactic distinction when defining the operator. According to the definition of the postfix increment operators in C++, it is also possible to use `int` instead of **var**.

The following example shows a fragment of a `Point` class representing an n-dimensional point. The point's coordinates are read and written using the index operator.

```
class Point()
{
   Point(uint dim);
```

```
   ~Point();
   var operator[]=(uint index, var value)
   var operator[](uint index) const;
   vector _coords; // vector of coordinates
}

Point::Point(uint dim)
{
   _coords = newvector(dim);
   for (var i=0;i<dim;++i) _coords[i] = 0;
}

var Point::operator[]=(uint index, var value) { _coords[index] = value
   _coords[index] = value;
   return value;
}

var Point::operator[](uint index) const { return _coords[index]; }

void main()
{
   var point = new Point(2);
   point[0] = 12; // set x-coordinate
   point[1] = 4.7; // set y-coordinate
   print("x=",point[0]," y=",point[1],"\n");
}
```

Note

To preserve the semantics of the assignment operator, the write variant (OP_VSET) implementation should return the assigned value, as in the example above.

Defining the dot operator

As already **mentioned**, the dot in B++ is a common operator for accessing elements of associative containers (data type `dictionary`) and public data elements of objects, which can be explicitly defined for objects or their classes, i.e. its functionality can be extended. This allows, for example, to implement access to elements of an object in the sense of properties (as used in C#) or user-defined associative containers that can be used similarly to the dictionary type.

The semantics of the dot operator are very similar to the index operator, so it is defined in a similar way.

The definition has the form

[returntype] OP_PREF(*key*) *[const]* or
[returntype] operator.(*key*)[const]

for read access or

[returntype] OP_PSET(*key*, *value*) or
[returntype] operator.=(*key*, *value*)

for write access.

key must always be of type `string` and a valid **identifier**.

Once again, in the following example a class for a point (this time a two-dimensional one) is used to represent a possible implementation.

```
class Point2d
{
```

```
private:
   double _x;
   double _y;
public:
   Point2d(double x=0, double y=0) { _x=x; _y=y; }
   double get_x() const { return _x; }
   void set_x(double val) { _x = val; }
   double get_y() const { return _y; }
   void set_y(double val) { _y = val; }
   double operator . (string key) const { return this->("get_"+key)(); }
   double operator .= (string key, double val) {
      this->("set_"+key)(val);
      return val;
   }
};

void main() {
   Point2d pt(15,25);
   pt.x *= 3;
   pt.y *= 2;
   print("pt: x=",pt.x," y=",pt.y,"\n");
}
```

Note

A custom dot operator does not override the default behavior of the operator for the class that defines it.
This means that the custom operator function will only be called if there is no member variable of the name specified by `key` that is visible in the current context.

Defining the Increment and Decrement Operators

As alredy explained, the **increment and decrement operators** can be used in prefix and postfix notation. Both variants can be defined for classes.

The operators are defined in the form

`OP_INC()` or **`OP_DEC()`** or
`operator++()` or **`operator--()`**

for prefix notation or

`OP_PINC()` or **`OP_PDEC()`** or
`operator++(var)` or `operator--(var)` or
`operator++(int)` or **`operator--(int)`**

for postfix notation.

Similar to C++ notation, the `operator` variant uses a dummy argument to distinguish between prefix and postfix notation.

The following example shows a portion of a rational number class to demonstrate the definition of the operators.

```
class Rational
{
private:
   int _num;    // numerator
   uint _denom; // denominator
public:
   Rational(int n=0, uint d=1) { _num=n; _denom=d; }
   Rational operator++() { _num += _denom; return this; }
```

```
   Rational operator++(int) {
      return new Rational(_num+_denom,_denom);
   }
   Rational operator--() { _num -= _denom; return this; }
   Rational operator--(int) {
      Rational r(_num,_denom);
      return --r;
   }
   operator string() { return newstring("%d/%u",_num,_denom); }
}

main()
{
   Rational r(1,2);
   print(r,"\n");             // prints 1/2
   print(r++," -> ", r,"\n"); // prints 1/2 -> 3/2
   print(++r,"\n");           // prints 5/2
}
```

The prefix notation implementations modify the object and return the modified object as the result.

The postfix notation implementations create a copy of the original object, modify it, and return it as the result.

Note

- The implementation of the postfix variants is different from the usual C++ procedure! There you return a copy of the original object, in B++ the modified copy.
- If a class defines only the prefix variant, this implementation is used implicitly when the operator is used in the postfix form, but not reverse. Although the compiler does not require it, prefix and postfix variants of the operators should generally be implemented in pairs to achieve behavior equivalent to that of the standard types.

Defining the Assignment Operator

For values of reference types, which also include objects, a reference to the target variable is normally simply assigned.

However, in certain cases it is desirable to be able to assign values of other types to an object, in addition to initialization in the constructor or as a reversal of conversion operators.
Therefore, from version 1.8 onwards, the assignment operator can be redefined to behave similarly to C++.

The operator is defined in the form

[returntype] OP_ASSIGN(*[const] value*) or
[returntype] operator=(<i>const] value)),

where *value* is the value to be assigned, which may be preceded by a type specification.

If a user-defined assignment operator exists, it is called for every assignment to the object. It can then handle the assignment itself or, by returning the value null, activate the default handling (simple assignment to the target variable).

Note

- Assigning the value *null*, or an object of the same or a derived type, is a special case. In this case, the user-defined assignment operator is not invoked. This exception is necessary to allow

> *null* to be assigned to the corresponding object variable, thereby releasing it or replacing an object instance.
> - A user-defined assignment operator should return either the object itself or the assigned value, in order to replicate the default behaviour of the assignment operator for simple data types (value types).

Defining the Dereference Operator

The unary dereference operator * is primarily used to convert a weak reference into a "real" value (see **Unary reference operators & and *.**). In addition, it can be explicitly defined as a prefix operator for objects, as follows

[const] [returntype] **OP_UNREF()** *[const]* or
[const] [returntype] **operator*()***[const]*.

The operator should be used to return an embedded object or other reference without modifying the defining object itself. A typical use case for this would be an iterator on a custom container that returns the object pointed to by the iterator.

Defining other Unary Operators

Table below lists the additional unary operators that can be defined for B++ classes.

Operator	Function	Operator Notation	Meaning
+	**OP_PLUS()**	**operator**+()	Positive sign
-	**OP_NEG()**	**operator**-()	Negative sign (two's complement)
!	**OP_NOT()**	**operator!()**	Logical negation
~	**OP_BNOT()**	Operator~()	Binary negation (one's complement)

The implementation rules for these operators are largely the same as in C++. The corresponding operator function should leave the calling object unchanged and return a modified copy of that object.

An implementation of the negative sign for the **Rational** class from example **above** might look like this:

```
Rational operator-() const { return new Rational(-_nom, _denom); }
```

The operator functions of unary operators always have an empty argument list. Optionally, a return type can be specified explicitly and operator function can be declared as \t const.

Indirect definition of comparison operators using the methods `compare` and `equals`

Unlike C++, B++ does not allow the direct definition of **comparison operators**. However, this is possible indirectly by using the member functions

*[bool]*equals(*[typename] other*) *[const]*
and/or
[int] compare(*[typename] other*) *[const]*

of a class where the types of arguments and return values are optional.

The `equals` method should return `true` if there is an equality between the calling object and *other*, otherwise return `false`.

The `compare` method should return a negative value if the calling object is less than other, a positive value if it is greater than other, and `0` on equality.

Methods should change neither the calling object nor the *`other`* object.

If an object is used in an expression as the left operand of a comparison operator, then in the case of the `==` and `!=` operators, the object is searched first for an `equals` method, for all other comparison operators, or if `equals` does not exist, for a `compare` method to perform the comparison sought.

If an object occurs as the right operand, the procedure is followed accordingly and the result of the comparison is reversed. If both operands are objects, the left operand "wins" if it implements suitable comparison methods. If neither operand provides its own comparison method, the standard comparison rules apply.

Following example extends the **Rational** class example **above** by adding a \t compare method.

```
class Rational
{
   int _num; // numerator
   uint _denom; // denominator
   Rational(int n=0, uint d=1) { _num=n; _denom=d; }
   // other functions ...
   operator string() const { return newstring("%d/%u",_num,_denom); }
   operator double() const { return ::double(_num) / _denom; }
   int compare(const var other) const {
      double d = double() - ::double(other);
      return d > 0 ? 1 : d < 0 ? -1 : 0;
   }
}

main()
{
   Rational r(1,2);
   print (r<0,r>0,r==0,r!=0,0<r,0>=r); // prints 010110
}
```

The example also uses a type **conversion** to `double`. Note that to prevent a recursive call to the member function `double`, the global conversion function **`double`** is explicitly called with the range resolution operator (`::`).

Defining Binary Infix Operators

When defining the binary infix operators, note that the object (the operand) can be on the left or right side of the operator. Since operators in B++ are always defined as member functions, there are two functions for each operator (see Table below), with the right operand variant having the suffix `_R`.

In the `operator` notation, to distinguish the two forms between the keyword `operator` and the operator symbol, there is an `L` (object as left operand) or an `R` (object as right operand), although the `L` can also be omitted - the left variant is assumed by default.

Operator	Function		Operator notation
	left	right	
Arithmetic operators			
+			
`OP_ADD(arg)`			
`OP_ADD_R(arg)`			

```
operator [L|R] + (arg)
-
OP_SUB(arg)
OP_SUB_R(arg)
operator [L|R] - (arg)
*
OP_MUL(arg)
OP_MUL_R(arg)
operator [L|R] * (arg)
/
OP_DIV(arg)
OP_DIV_R(arg)
operator [L|R] / (arg)
%
OP_REM(arg)
OP_REM_R(arg)
operator [L|R] % (arg)
```

Bitwise operators

```
|
OP_BOR(arg)
OP_BOR_R(arg)
operator [L|R] | (bad)
&
OP_BAND(arg)
OP_BAND_R(arg)
operator [L|R] & (arg)
^
OP_XOR(arg)
OP_XOR_R(arg)
operator [L|R] ^ (arg)
```

Shift operators

<<

OP_SHL(arg)

OP_SHL_R(arg)

operator [L|R] << (**arg**)

>>

OP_SHR(arg)

OP_SHR_R(arg)

operator [L|R] >> (**arg**)

The principle of definition is the same for all these operators and is illustrated in the following example for the addition operator.
Here, the class `Point` (see **Definition of the Index Operator**) is modified and the + operator is implemented in such a way that it creates a new `Point` object, whose coordinates are formed as the sum of the coordinates of the operands, if the second operand is also of type `Point`. Otherwise, an attempt is made to convert the second operand to `double` type and add its value to all components of the first operand, with the result also being a new object.

```
class Point
{
   vector _coords;                              // vector of coordinates
   vector coords() const { return _coords };    // gets coordinates
   Point(uint dim);                             // ctor
   Point operator+(const var other) const;      // operator + (left)
   Point operator R +(const var other) const;   // operator + (right)
}

Point::Point(uint dim)
{
   _coords = newvector(dim);
   for (uint i=0; i<dim; ++i) _coords[i] = 0.0;
}

Point Point::operator+(const var other) const
{
   uint dim = size(_coords);
   if (dynamic_cast(Point, other))
   {
      vector ocoords =  other->coords();
      uint odim = size(ocoords);
      if (odim < dim) dim = odim;
      Point result(dim);
      vector rcoords = result->coords();
      for (uint i=0;i<dim;++i) rcoords[i] = _coords[i] + ocoords[i];
      return result;
   }
   double d = other;  // implicit conversion
   Point res(dim);
   vector rc = res->coords()
   for (uint j=0;j<dim;++j)
     rc[j] = _coords[j] + d;
   return res;
}

Point Point::operator R+(const var other) const
{
```

```
    return OP_ADD(other);
}
```

Note

- To achieve behavior analogous to the built-in data types for user-defined classes, the infix operators should always be defined in pairs (i.e., for the left and right operands).
- The operators should never modify one or both operands, but always return a new object. The type of the result should match the type of the defining object.
- If the behavior of an operator is symmetrical, as in addition, the implementation of one variant can call that of the other one (see `operator R+` in example above). The functional form must be used for the call.
- The type of the second operand passed as an argument is often (as in the example) not statically typed to allow use in mixed expressions. In such a case, the implementation of the operator function is responsible for performing appropriate type checking or conversions.
- Infix operators are evaluated from left to right. That is, the type of the left operand is considered first. If it is an object type, then the corresponding left operator function, if any, is called with the right operand as an argument. If the right operand is an object but the left operand is not, then the right operator function is called with the left operand as an argument.

Type Conversion Operators

As already **mentioned**, member functions can be defined for objects (classes) to convert them into other data types.
Syntactically, two variants are allowed:

```
[type] type() [const]
```

or

```
operator type() [const].
```

The first form is a 'normal' method definition, where the method name (*type*) corresponds to the return type. Optionally, the return type can also be specified. The second form is the C++ notation for conversion operators.

Conversion methods do not take arguments.

The example in section about **definition of comparison operators** defines conversion operators into the types `double` and `string` for the class `Rational`.

To explicitly convert values to object types, you can also use global functions that have the same name as the class whose type you want to convert, i.e., the form

```
[classtype] classtype([const] [type]arg)
```

Such a function should always return a result of the type of the corresponding class, or `null` if no conversion is possible. Otherwise, there are no strict implementation rules for such functions.

Note

- Technically, a global conversion function is an "ordinary" function that can take any arguments, perform any actions, and return any results, and whose only special feature is that its name

matches that of a class. It would also be conceivable to use such functions to create objects (instead of the `new` operator), i.e. in the role of factory functions.
- A class that defines a conversion to **`char8_t`** should always define a conversion to `uchar` too, since **`char8_t`** is internally an alias for `uchar`.

Namespaces

The concept of namespaces, which is used to structure identifiers according to their intended use and to avoid identifier conflicts, is used in several programming languages, including C++ and Java.

Namespaces are supported in B++ since version 1.5 and are syntactically analogous to C++ and defined in the form

```
namespace dentifier { ... }.
```

From a technical point of view, namespaces in B++ are classes with the following restrictions.

- All members (variables, functions) are class members (`static`).
- There are no constructors/destructors, so no instances can be created.
- All members are implicitly `public`.
- There is no inheritance of namespaces.

Namespaces can contain child namespaces, class definitions, variables, and constants. However, they cannot be part of a real class; defining a namespace as a child of a class is not allowed (unlike real classes).

Referencing elements of a namespace is done like referencing static elements of a class, using the scope resolution operator (`::`).
There is *no* \t using clause, so *elements* of a namespace *must always be addressed explicitly*, except in the context of the namespace itself or a subordinate namespace.

Synchronous timers and attached function extensions

B++ supports two concepts for executing function calls outside the regular program flow. Such function calls can simplify program logic, thus contributing to shorter and faster programs.

Synchronous timers

Synchronous timers allow the single or repeated execution of a function after a specified time interval, interrupting the normal program flow.

They can be installed at any position in code using the **`settimer`** function and uninstalled as needed using the **cleartimer** or **cleartimers** functions.

The bytecode interpreter periodically checks for expired timers and executes the associated functions if necessary during program execution. To reduce or control the potential slowdown of programme execution, the predefined global variable `__TMDELAY__` controls the number of bytecode execution cycles that occur between checks for expired timers. The default value is 10. The higher the value, the less frequently the timers are checked. This results in a shorter execution delay, but also reduces the timers' accuracy. If no timers are installed, the value of `__TMDELAY__` is irrelevant.

Exceptions in Timer Functions

An exception raised by a timer function is generally not passed to the executing program because the calling context of the timer function is undefined. Such an exception is therefore simply discarded. If the timer is cyclical, it is automatically uninstalled and the function will not be called again.

Note

- Synchronous timers cannot interrupt each other. While a timer function is being executed, no checks are performed for other expired timers.
- It is possible to use coroutines as timer functions, but this is only useful for cyclic timers.
- Timer functions cannot pass result values to the calling program.

Attached function extensions

Attached function extensions are functions that are executed immediately upon exiting another function, regardless of where or how the latter function exits. In certain situations, this can simplify the structure of the function flow.

There are three ways to execute appended function extensions:

1. Execution only if the primary function does not throw an exception,
2. Execution only if the primary function throws an exception and
3. Execution regardless of whether the primary function throws an exception.

Appended functions are installed using the predefined functions **callsucc** (Variant 1), **callfail** (Variant 2) or **callafter** (Variant 3)
As installation can occur at any point during implementation of the primary function, it can be based on specific conditions.

Attached function extensions and exceptions

If an attached function extension throws an exception, this exception is forwarded to the caller of the primary function. If the primary function itself threw an exception, it may be replaced by the exception thrown by the function extension.
If the primary function threw an exception, but the function extension did not, the exception from the primary function remains. Therefore, an attached function extension has no way of handling an exception from the primary function.

Note

- Attached function extensions do not have access to local variables of the primary function and cannot affect its return value.
- It is possible to install multiple attached function extensions for a function. They are then executed in the order in which they were installed. If a function extension throws an exception, execution is aborted.
- It is possible to use coroutines as attached function extensions.

Memory Management

In Section **Data Types** a distinction was made between the data types occurring in B++ according to **value types** and **reference types**.

Whereas a value type is directly bound to the existence of a value object, a reference type contains only a reference to a data area. This raises the question of how to manage the (dynamic) storage required for the actual data.

A special case is the type `FILE`, whose instances only contain a reference to a description structure provided by the operating system, which is requested or released by the standard functions **fopen** and **fclose**.

For all other reference types, B++ uses automatic memory management with reference counting by default. That is, each value of a reference type carries an internal reference count that is incremented when assigned to a variable and decremented when released (when the variable goes out of scope or is assigned a different value). When a reference counter reaches zero, it means that the value can no longer be accessed from anywhere in the program. In this case, it (and the memory used for the data area) is automatically freed.

> **Note**
>
> In contrast to many other scripting languages, B++ does not use a true garbage collector for reasons of runtime efficiency.

This behavior means that the lifetime of variables is strictly defined in B++. In particular, it is guaranteed that when you exit a function or throw an **exception**, all temporary variables in the respective context are freed, and the destructors of objects are called implicitly.

The example below illustrates this:

```
class A {
   static uint _id = 0;
   uint id;
   A() { id = ++_id; print(__func__," id:",id,"\n"); }
   ~A() { print(__func__," id:",id,"\n"); }
   operator string() { return newstring("class: %s id: %u",
                              getclassname(this),id); }
}
void testfunc() {
     A a;
}
void testfunc2() {
    A a;
    throw new A();
}

main()
{
   try {
      print("enter testfunc\n");
      testfunc();
      print("enter testfunc2\n");
      testfunc2();
   }
   catch(e) {s
      print(e," caught\n");
   }
}
```

Two functions (`testfunc` and `testfunc2`) are defined here, each of which creates an object of class `A`. While `testfunc` simply returns to the caller, `testfunc2` throws an exception with another instance of class `A` as the exception object. In the main program, both functions are called in a protected section (`try-catch`), producing the following output:

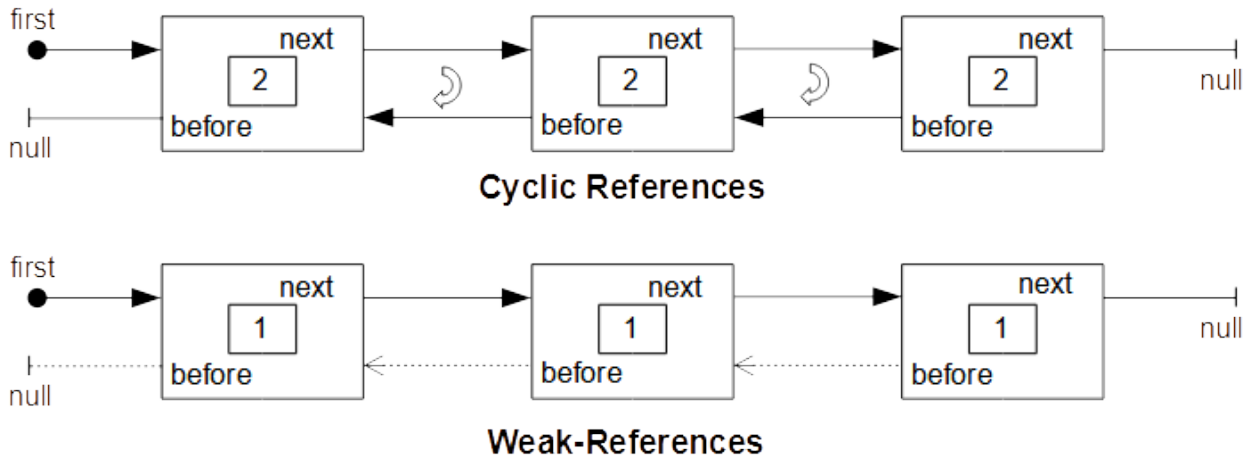
```
enter testfunc
A::A id:1
A::dtor id:1
enter testfunc2
A::A id:2
A::A id:3
A::dtor id:2
Class: A id: 3 caught
A::dtor id:3
```

The object created in `testfunc` (ID=1) is released when the function is exited. In `testfunc2`, an object is first created locally (ID=2) and then another instance of `A` (ID=3) is created as an exception object. The automatic release of the local object (ID=2) now occurs when `testfunc2` is exited (immediately after the exception is thrown), while the exception object (ID=3) is released when the `catch` block is exited.

> **Note**
>
> In the example, `testfunc2` could also use the local object `a` (ID=2) directly as the exception object. In this case, when `throw` is called, its reference count would be incremented so that a release does not occur until the `catch` block in main is exited.

Weak References and the Unary Operators & and *

One problem with memory management using reference counting is the occurrence of *cyclic references*, as illustrated in figure below (top portion) using a double-linked list:



Cyclic References and their Resolution

Because of the mutual references between the list elements, the reference counts of the elements are each two. If you release the reference to the beginning of the list (the variable `first`), the reference count of the first element is reduced, but does not reach zero, so that the element (or the entire list) is not released, even though there is no way to access it from the program.

This can be solved by using references that are explicitly excluded from the reference count (often called *weak references*), as shown in the bottom part of the figure.

In B++, such references are created with the unary operator &, which excludes the trailing value from reference counting when it is subsequently assigned to a variable.

The unary operator * is its counterpart. It converts a weak reference back into a "real" reference, i.e., a variable to which the result is assigned is included in the reference count again. Following example demonstrates the practical use of weak references in more detail.

```
// Double linked list class example
// --------------------------------

// List class
class List
{
   // List element class
   class element
   {
   private:
      var _prev = null; // Weak reference to previous element
      element _next = null; // Reference to next element
      var _value = null; // List elements value
   public:
      // ctor
      element(var v = null; ) { _value = v; }
      // Definition of operator. for member access
      var operator.(string key) {
         switch (key)
         {
         case "prev": return _prev;
         case "next": return _next;
         case "value": return _value;
         }
         throw "invalid key " + key;
      }
      void operator. = (string key, var val){
          switch (key)
          {
           case "prev":
               _prev = &val;
             break;
           case "next": _next = val;
              break;
           case "value": _value = val;
              break;
           default:
             throw "invalid key " + key;
         }
      }
   };
   // List iterator class
   class iterator
   {
   private:
      var _lst = null; // Weak reference to list
      var _item = null; // Weak reference to list item
   public:
      // ctor
      iterator(List lst, var item = null) { _lst = &lst; _item = &item; }
      // Increment
      iterator operator++() {
         if (_item) _item = &_item.next;
         else _item = &_lst->first();
         return this;
      }
      // Post-Increment
      iterator operator++(int) {
         iterator it(_lst, _item);
         return ++it;
      }
      // Decrement
      iterator operator--() {
         if (_item) _item = &_item.prev;
```

```
         else _item = &_lst->back();
         return this;
      }
      // Post-Decrement
      iterator operator--(int) {
         iterator it(_lst, _item);
         return --it;
      }
      var item() { return _item; }
      var lst() { return _lst; }

      // Equality operator
      bool equals(iterator other)
      {
         if (_lst != other._lst) return false;
         return _item == other._item;
      }
      // Gets referenced list item.
      element operator*() { return *_item; }
      // Delegates operator. to referenced list element
      var operator.(string key) { return _item->(element::OP_PREF)(key); }
      void operator. = (string key, var val){ _item->(element::OP_PSET)(key,val); }
   }

private:
   element _first = null;
   var _last = null;
   uint _size = 0;
   // Internal helpers
   // Initialization
   void _init(var first, var last, uint siz) {
      _first = first; _last = last; _size = siz;
   }
   // Removes an element.
   element _remove(var el)
   {
      if (!_size) throw "List is empty";
      if (!el) throw "Invalid argument";
      if (!el.prev) {
         var f = _first;
         _first = el.next;
      }
      else {
         var e = el.prev.next;
         el.prev.next = el.next;
      }
      if (!el.next) _last = el.prev;
      else el.next.prev = el.prev;
      --_size;
      return el.next;
   }
   // Inserts an element
   element _insert(var val, var beforeEl) {
      element el(val);
      if (!beforeEl)
      {
         if (_last)
         {
            el.prev = _last;
            _last.next = el;
         }
         else _first = el;
         _last = &el;
      }
      else {
         if (!_size) throw "List is empty";
         print("#insert ", val, "\n");
         el.next = beforeEl;
         if (!beforeEl.prev) _first = el;
         else {
```

```
            beforeEl.prev.next = el;
            el.prev = beforeEl.prev;
         }
         beforeEl.prev = &el;
      }
      ++_size;
      return el;
   }
public:
   // Ctor - allows initialization from vector
   List(vector init = null) {
      if (init) {
         uint n = size(init);
         for (uint i = 0; i < n; ++i) this->(_insert)(init[i], null);
      }
   }
   // STL-like functions
   element front() { return _first; }
   element back() { return _last ? *_last : null; }
   iterator begin() { return new iterator(this, _first); }
   iterator end() { return new iterator(this); }
   void push_back(var val) { this->(_insert)(val, null); }
   void push_front(var val) { this->(_insert)(val, _first); }
   void pop_back() { this->(_remove)(_last); }
   void pop_front() { this->(_remove)(_first); }
   iterator insert(iterator before, var val) {
      if (before->(iterator::lst)() != this) throw "Invalid iterator";
      return new iterator(this, this->(_insert)(val, before->(iterator::item)()));
   }
   iterator erase(iterator first, iterator last = null) {
      if (first->(iterator::lst)() != this) throw "Invalid iterator";
      if (!last) return new iterator(this, this->(_remove)(first->(iterator::item)()));
      if (last->(iterator::lst)() != this) throw "Invalid iterator";
      auto itend = end();

      while (first != itend) {
         auto item = *first;
         ++first;
         this->(_remove)(item);
         if (first == last) break;
      }
      return new iterator(this, item);
   }
   uint size() { return _size; }
   void clear() { _first = null; _last = null; _size = 0; }
   void swap(List other)
   {
      var ofirst = other->(List::front)();
      var olast = other->(List::back)();
      uint osize = other->(List::size)();
      other->(List::_init)(_first, _last, _size);
      _first = ofirst; _last = &olast; _size = osize;
   }
   // Conversion to string
   operator string() {
      string result = "";
      for (var el = _first; el; el = el.next)
      {
         result += ::string(el.value);
         if (el.next) result += ",";
      }
      return result;
   }
}

main()
{
   List l([1, 2, 3, 4, 5, 6, 7]);
   List l2([9, 8, 7, 6]);
```

```
   l->insert(l->end(), "00");
   auto i = l->begin();
   ++i;
   auto j = clone(i);
   ++j;

   l->erase(i, ++j);

   for (auto it = l->(List::begin)(); it != l->(List::end)(); ++it) print(*it.value, "\n");
   for (it = l2->(List::begin)(); it != l2->(List::end)(); ++it) print(*it.value, "\n");

   print(l, "\n", l2, "\n");
   print("--------------\n");
   l->swap(l2);
   for (it = l->(List::begin)(); it != l->(List::end)(); ++it) print(*it.value, "\n");
   for (it = l2->(List::begin)(); it != l2->(List::end)(); ++it) print(*it.value, "\n");
   print(l, "\n", l2, "\n");

   List l3 = clone(l2);

   l2 = null;
   print("reverse:\n");
   for (auto itr = l3->end(); (--itr) != l3->end();)
      print(*itr.value, " ");
   print("\n");
}
```

A double-linked list is implemented here, similar to that in the C++ standard library.
The example also illustrates some of the object-oriented programming techniques described **above**, in particular the definition of **user-defined operators** and the **static (not virtual) binding** of member functions.

Explicit Memory Management

In addition to creating **weak references** to resolve circular references, the unary & operator can also be used to create variables or objects (more precisely: values that are actually subject to reference counting) that are excluded from reference counting and therefore must be explicitly freed using the `delete` operator.
This can be useful in exceptional cases where the destruction of a particular object should occur at a precisely specified time, i.e. before the implicit destruction when the variable holding the value leaves the scope. This is especially true for global (or `static`) variables, which are normally destroyed only when the B++ VM is released, which may prevent destructors of objects from being called.

An explicitly managed variable is created by converting a value of a reference type into a weak reference the first time it is assigned to the variable, e.g:

```
var obj = &new MyClass();
var vec = &newvector(1,2,3);
```

When variables initialized in this way are no longer needed, they must be released using the `delete` operator:

```
delete obj;
delete vec;
```

or better

```
obj = delete obj;
vec = delete vec;
```

Unlike C++, the B++ delete operator always returns the value `null`, which can be reassigned to the variable to prevent later accidental use of the now invalid object reference.

Note

- Like all weak references, explicitly managed values cannot be used with static typing.
- It is possible to subsequently convert the value of an originally explicitly managed variable to an automatically managed value by using the dereference operator * and assigning it to another variable, but the reverse is not true.
  Caution: Such a dereference results in the immediate release of the object (analogous to calling delete) if there is no assignment to a variable made.

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.