Cmajor Programming Language Specification
Version 3.4

Seppo Laakko

January 6, 2016

Contents

1 Grammar Notation

The lexical and syntactical structure of Cmajor programs is presented in this text using grammars that are in extended Backus-Naur form. A grammar consists of

1. *Terminal* symbols that are the elementary symbols of the language generated by the grammar. Terminals are presented like `this` in a type-writer font.

2. *Nonterminal* symbols also called syntactic variables that represent sets of strings of terminal symbols. Nonterminals are presented like ⟨*this*⟩ in italic font within angle brackets.

3. *Operators* ( ), |, −, *, +, ?, [ ], and \ that operate on strings of terminals and nonterminals. If an operator character is ment to be a terminal symbol, it is quoted like this: '`*`'. The meaning of each operator appears in table 1.

4. *Productions* each of which consists of a nonterminal symbol called the *head* of the production, symbol ::= (pronounced "produces"), and a sequence of terminals, nonterminals and operators collectively called the *body* of the production.

Table 1: Grammar Operators

Expression	**Meaning**
$\alpha \mid \beta$	α or β
$\alpha - \beta$	α but not β
$\alpha*$	zero or more α's
$\alpha+$	one or more α's
$\alpha?$	zero or one α's
$(\alpha\ \beta)$	grouping of α and β together
`[a-z]`	a terminal character in range a ... z
`[^\r\na-c]`	any terminal character except a carriage return, a newline or a character in range a ... c

2 Lexical Structure

2.1 White Space and Comments

Lexical elements in program texts can be separated by one or more instance of white space characters and comments.

⟨*spaces-and-comments*⟩ ::= (⟨*space*⟩ | ⟨*comment*⟩)+

⟨*space*⟩ ::= 'any ASCII character classified as space character according to C isspace function'

⟨*comment*⟩ ::= ⟨*line-comment*⟩ | ⟨*block-comment*⟩

⟨*line-comment*⟩ ::= `//` `[^\r\n]`* ⟨*new-line*⟩

⟨*new-line*⟩ ::= `\r\n` | `\n` | `\r`

⟨*block-comment*⟩ ::= '`/*`' (⟨*any-char*⟩ − '`*/`') '`*/`'

⟨*any-char*⟩ ::= 'any 8-bit ASCII character'

2.2 Keywords

Keywords have reserved context-dependent meaning in a program and they cannot be used as identifiers.

⟨*keyword*⟩ ::= 'see table 2'

Table 2: Keywords

abstract	**and**	**as**	**axiom**	**base**	**bool**
break	**byte**	**case**	**cast**	**catch**	**cdecl**
char	**class**	**concept**	**const**	**construct**	**continue**
default	**delegate**	**delete**	**destroy**	**do**	**double**
else	**enum**	**explicit**	**extern**	**false**	**float**
for	**goto**	**if**	**inline**	**int**	**internal**
is	**long**	**namespace**	**new**	**not**	**nothrow**
null	**operator**	**or**	**override**	**private**	**protected**
public	**return**	**sbyte**	**short**	**sizeof**	**static**
suppress	**switch**	**this**	**throw**	**true**	**try**
typedef	**typename**	**uint**	**ulong**	**unit_test**	**ushort**
using	**virtual**	**void**	**where**	**while**	

2.3 Identifiers

Identifiers are used to name entities in a program. Identifiers that begin with double underscores are reserved for the compiler.

⟨*identifier*⟩ ::= ⟨*id-char-sequence*⟩ − ⟨*keyword*⟩2.2

⟨*id-char-sequence*⟩ ::= (⟨*letter*⟩ | _) (⟨*letter*⟩ | _| ⟨*digit*⟩)*

⟨*letter*⟩ ::= [A-Za-z]

⟨*digit*⟩ ::= [0-9]

⟨*qualified-id*⟩ ::= ⟨*identifier*⟩ (. ⟨*identifier*⟩)*

2.4 Literals

Literals are used to write values in a program.

⟨*literal*⟩ ::= ⟨*boolean-literal*⟩2.4.1
| ⟨*integer-literal*⟩2.4.2
| ⟨*floating-literal*⟩2.4.3
| ⟨*character-literal*⟩2.4.4
| ⟨*string-literal*⟩2.4.5
| ⟨*raw-string-literal*⟩2.4.5
| ⟨*null-literal*⟩2.4.6

2.4.1 Boolean Literals

⟨*boolean-literal*⟩ ::= `true` | `false`

The type of a boolean literal is **bool**.

2.4.2 Integer Literals

⟨*integer-literal*⟩ ::= ⟨*decimal-digit-sequence*⟩ ⟨*unsigned-suffix*⟩?
| (`0x` | `0X`) ⟨*hexadecimal-digit-sequence*⟩ ⟨*unsigned-suffix*⟩?

⟨*decimal-digit-sequence*⟩ ::= ⟨*decimal-digit*⟩+

⟨*decimal-digit*⟩ ::= [`0-9`]

⟨*unsigned-suffix*⟩ ::= 'u' | 'U'

⟨*hexadecimal-digit-sequence*⟩ ::= ⟨*hexadecimal-digit*⟩+

⟨*hexadecimal-digit*⟩ ::= [`0-9a-fA-F`]

If integer literal has no unsigned suffix, the type of the integer literal is first of the following types that can represent the value: **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**.

If integer literal has an unsigned suffix, the type of the integer literal is first of the following types that can represent the value: **byte**, **ushort**, **uint**, **ulong**.

2.4.3 Floating Literals

⟨*floating-literal*⟩ ::= (⟨*fractional-floating-literal*⟩ | ⟨*exponent-floating-literal*⟩) ⟨*floating-suffix*⟩?

⟨*fractional-floating-literal*⟩ ::= ⟨*decimal-digit-sequence*⟩? `.` ⟨*decimal-digit-sequence*⟩ ⟨*exponent-part*⟩?
| ⟨*decimal-digit-sequence*⟩ `.`

⟨*exponent-part*⟩ ::= (`e` | `E`) ⟨*sign*⟩? ⟨*decimal-digit-sequence*⟩

⟨*sign*⟩ ::= `+` | `-`

⟨*exponent-floating-literal*⟩ ::= ⟨*decimal-digit-sequence*⟩ ⟨*exponent-part*⟩

⟨*floating-suffix*⟩ ::= 'f' | 'F'

If a floating literal has a floating suffix, the type of it is **float**, otherwise the type of the floating literal is **double**.

2.4.4 Character Literals

⟨*character-literal*⟩ ::= `'` (`[^\r\n]` | ⟨*escape*⟩) `'`

⟨*escape*⟩ ::= `\` (⟨*hex-escape*⟩ | ⟨*decimal-escape*⟩ | ⟨*char-escape*⟩)

⟨*hex-escape*⟩ ::= `[xX]` ⟨*hexadecimal-digit-sequence*⟩

⟨*decimal-escape*⟩ ::= `[dD]` ⟨*decimal-digit-sequence*⟩

⟨*char-escape*⟩ ::= `[^dDxX\r\n]` 'see table 3'

Table 3: Character Escape Sequences

Escape Sequence	Character Name	Character Code
'\n'	new line	10
'\t'	horizontal tab	9
'\v'	vertical tab	11
'\b'	backspace	8
'\r'	carriage return	13
'\f'	form feed	12
'\a'	alert	7
'\\'	backslash	92
'\0'	null	0
'\x'	any other character x stands for itself	

The type of a character literals is **char**.

2.4.5 String Literals

The backslash character provides an escaping mechanism for ordinary string literals.

⟨*string-literal*⟩ ::= `"` (`[^"\r\n]` | ⟨*escape*⟩)* `"`

There is no escapes in raw string literals. The content is taken literally.

⟨*raw-string-literal*⟩ ::= `@"` (`[^"]`)* `"`

The type of a string literal and raw string literal is **const char***.

2.4.6 Null-literal

⟨*null-literal*⟩ ::= `null`

The null literal can be implicitly converted to any pointer or delegate type.

3 Basic Types

Cmajor has basic types for representing Boolean, integer, floating-point and character values, and for expressing lack of type.

⟨*basic-type*⟩ ::= `bool`
| `sbyte`
| `byte`
| `short`
| `ushort`
| `int`
| `uint`
| `long`
| `ulong`
| `float`
| `double`
| `char`
| `void`

3.1 Boolean Type

The Boolean type **bool** is an 8-bit integral type with a value 1 representing **true** and value 0 representing **false**.

Operators unary !, and binary ||, && and == operate on Boolean type operands.

The default value of a **bool** type object is **false**.

A Boolean type value can be converted to an integer type, floating-point type or **char** type with an explicit **cast**.

3.2 Integer Types

sbyte is an 8-bit signed integer type for representing values in range $-128 \ldots 127$.
byte is an 8-bit unsigned integer type for representing values in range $0 \ldots 255$.
short is a 16-bit signed integer type for representing values in range $-32768 \ldots 32767$.
ushort is a 16-bit unsigned integer type for representing values in range $0 \ldots 65535$.
int is a 32-bit signed integer type for representing values in range $-2147483648 \ldots 2147483647$.
uint is a 32-bit unsigned integer type for representing values in range $0 \ldots 4294967295$.
long is a 64-bit signed integer type for representing values in range $-9223372036854775808 \ldots 9223372036854775807$.
ulong is a 64-bit unsigned integer type for representing values in range $0 \ldots 18446744073709551615$.

The default value of an integer type is 0.

Unary operators −, +, ∼, prefix operators ++ and −−, binary operators +, −, ∗, /, %, ==, <, &, |, ^, <<, and >> operate on integer type operands.

There is an implicit conversion from a signed or unsigned integer type whose bit width is smaller to a signed integer type whose bit width is larger. For example, a **sbyte** can be implicitly converted to an **int**, because the bit width of **sbyte** is 8 and the bit width of **int** is 32. There is an implicit conversion from an unsigned integer type whose bit width is smaller to an unsigned integer type whose bit width is larger. For example, a **ushort** can be implicitly

converted to an **uint**, because the bit width of **ushort** is 16 and the bit width of **uint** is 32. Other conversions between integer types are explicit (they need a **cast**).

There is an implicit conversion from integer types whose bit width is less than or equal to 32 bits to **float** floating-point type.

There is an implicit conversion from all integer types to **double** floating-point type.

An integer type value can be converted to a **char** or **bool** value with an explicit **cast**.

3.3 Floating-point Types

Floating-point type **float** is a 32-bit IEEE 754 type. The default value of a **float** type object is 0.0f.

Floating-point type **double** is a 64-bit IEEE 754 type. The default value of a **double** type object is 0.0.

There is an implicit conversion from **float** to **double**.

A **double** type value can be converted to **float** with an explicit **cast**.

A floating-point type value can be converted to integer type, **char** type or **bool** type with an explicit **cast**.

Operators unary $-$, $+$, binary $+$, $-$, $*$, $/$, $==$ and $<$ operate on floating-point type operands. A floating-point type value can be converted to an integer type with an explicit **cast** operator.

3.4 Character Type

The character type **char** is a 8-bit integral ASCII code type for representing character code values in range $0 \ldots 255$. The default value of a **char** type object is '\0'.

Character code values can be compared for equality with operator $==$ and for less-than relationship with operator $<$.

A **char** value can be converted to an integer type, floating-point type or **bool** type with an explicit **cast**.

3.5 Void Type

The type **void** represents lack of type. It is an incomplete type that cannot be completed, meaning you cannot have an object of type **void**.

4 Expressions

An expression represents a computation that usually produces a value [1]. This grammar accepts a superset of valid expressions. The implementation contains special disambiguation rules that reject syntactically valid (according to the grammar) but meaningless expressions. The type checking phase of the compiler further rejects semantically invalid expressions.

⟨*expression*⟩ ::= ⟨*equivalence*⟩4.1

[1] Calling a void function does not produce a value.

4.1 Equivalence

⟨*equivalence*⟩ ::= ⟨*implication*⟩4.2 (`<=>` ⟨*implication*⟩)*

The <=> operator is used in axioms (10.4) to state that operand expressions are logically equivalent. It groups operands from left to right.

4.2 Implication

⟨*implication*⟩ ::= ⟨*disjunction*⟩4.3 (`=>` ⟨*implication*⟩)?

The => operator is used in axioms (10.4) to state that the right operand is logical implication of the left operand. It groups operands from right to left.

4.3 Disjunction

⟨*disjunction*⟩ ::= ⟨*conjunction*⟩4.4 (|| ⟨*conjunction*⟩)*

The || operator takes boolean operands and yields a boolean value. It groups operands from left to right. Expression $a||b$ is **true** when a is **true**, or a is **false** and b is **true** (in the first case operand b is not evaluated); otherwise **false**.

A user-defined class cannot overload the || operator.

4.4 Conjunction

⟨*conjunction*⟩ ::= ⟨*bit-or-expr*⟩4.5 (`&&` ⟨*bit-or-expr*⟩)*

The && operator takes boolean operands and yields a boolean value. It groups operands from left to right. Expression $a\&\&b$ is **false** when a is **false**, or a is **true** and b is **false** (in the first case operand b is not evaluated); otherwise **true**.

A user-defined class cannot overload the && operator.

4.5 Bitwise OR-expression

⟨*bit-or-expr*⟩ ::= ⟨*bit-xor-expr*⟩4.6 (| ⟨*bit-xor-expr*⟩)*

The | operator with integer operands yields an integer value. It groups operands from left to right. The result is a bitwise OR of its operands: that is, each bit of the result will be 1 if one or both of the corresponding bits of the operands is 1; 0 otherwise.

A user-defined class can overload the | operator by implementing the **operator|** function.

4.6 Bitwise XOR-expression

⟨*bit-xor-expr*⟩ ::= ⟨*bit-and-expr*⟩4.7 (^ ⟨*bit-and-expr*⟩)*

The ^ operator with integer operands yields an integer value. It groups operands from left to right. The result is a bitwise XOR of its operands: that is, each bit of the result will be 1 if either but not both of the corresponding bits of the operands is 1; 0 otherwise.

A user-defined class can overload the ^ operator by implementing the **operator^** function.

4.7 Bitwise AND-expression

⟨*bit-and-expr*⟩ ::= ⟨*equality-expr*⟩4.8 (`&` ⟨*equality-expr*⟩)*

The & operator with integer operands yields an integer value. It groups operands from left to right. The result is a bitwise AND of its operands: that is, each bit of the result will be 1 if both of the corresponding bits of the operands is 1; 0 otherwise.

A user-defined class can overload the & operator by implementing the **operator&** function.

4.8 Equality-expression

⟨*equality-expr*⟩ ::= ⟨*relational-expr*⟩4.9 ((`==` | `!=`) ⟨*relational-expr*⟩)*

The == operator compares equality. It groups operands from left to right.

The compiler will automatically implement the != operator for a type if it implements the == operator. Then expression a != b will be evaluated as **operator!(operator==(a,b))**.

A user-defined class can overload the == and != operators by implementing the **operator==** function.

4.9 Relational Expression

⟨*relational-expr*⟩ ::= ⟨*shift-expr*⟩4.10 ((`<` | `>` | `<=` | `>=`) ⟨*shift-expr*⟩)*
| ⟨*shift-expr*⟩ ((`is` | `as`) ⟨*type-expr*⟩4.21)*

The < operator compares less-than relationship. It groups operands from left to right.

The compiler will automatically implement operators >, <= and >= for a type if it implements the < operator. Then expression $a > b$ will be evaluated as **operator<(b,a)**, expression $a <= b$ as **operator!(operator<(b,a))**, expression $a >= b$ as **operator!(operator<(a,b))**.

A user-defined class can overload the < and >, <= and >= operators by implementing the **operator<** function.

4.9.1 Is and As Operators

Consider following code:

```
1  public class Base
2  {
3      public virtual ~Base()
4      {
5      }
6      //  ...
7  }
8
9  public class Derived : Base
10 {
11     // ...
12 }
13
14 public Base* GetBasePtrFromSomewhere()
15 {
```

```
16      return new Derived();
17  }
18
19  void main()
20  {
21      Base* b1 = GetBasePtrFromSomewhere();
22      if (b1 is Derived*)
23      {
24          // do something with b1
25      }
26
27      Base* b2 = GetBasePtrFromSomewhere();
28      Derived* d = b2 as Derived*;
29      if (d != null)
30      {
31          // do something with d
32      }
33  }
```

The **is** operator tests if the left operand can be legally casted to a pointer type on the right-hand side. That is: if the pointer *b1* in fact points to a *Derived* class object or an object of class derived from *Derived*. The left operand of the **is** operator must be a pointer to a virtual class object [2] and the right operand must be a pointer to virtual class type. The **is** operator yields a Boolean result.

The **as** operator tries to convert the left operand to a pointer type on the right-hand side. If the conversion succeeds you got a non-null pointer to *Derived* class, otherwise the result is **null**. In the case of previous example the conversion succeeds if pointer *b2* in fact points to *Derived* class object or an object of class derived from *Derived*. The left operand of the **as** operator must be pointer to a virtual class object and the right operand must be a pointer to virtual class type. The **as** operator yields **null** or non-null pointer result. It performs an operation similar to C++ **dynamic_cast**.

Note: in the **debug**, **release** and **profile** configurations the **is** and **as** operators generate a call to a function that traverses the class hierarchy, but in **full** configuration the class identifiers are chosen so that the **is** and **as** generate only a single modulo operation in addition to retrieving the class identifier from the run-time type information table. In the **full** configuration the compiler does whole-program analysis and implements a scheme described in http://www.stroustrup.com/fast_dynamic_casting.pdf by Michael Gibbs and Bjarne Stroustrup.

4.10 Shift Expression

⟨*shift-expr*⟩ ::= ⟨*additive-expr*⟩4.11 ((<< | >>) ⟨*additive-expr*⟩)*

The << operator with integer operands will yield an integer value. It groups operands from left to right. Expression $a << b$ with integer operands a and b will yield a value of a shifted b bit positions left and filling the vacant bit positions of a with 0-bits from right. If a is non-negative and b small enough the result will be $2^b a$.

[2] to a class object containing virtual, abstract or overridden member functions

The >> operator with integer operands will yield an integer value. It groups operands from left to right. Expression $a >> b$ with integer operands a and b will yield a value of a shifted b bit positions right and filling the vacant bit positions of a with 0-bits from left. If a is non-negative the result will be $a/2^b$ ('/' meaning integer division).

A user-defined class can overload the << and >> operators by implementing the **operator**<< and **operator**>> functions respectively.

4.11 Additive Expression

⟨*additive-expr*⟩ ::= ⟨*multiplicative-expr*⟩4.12 ((`+` | '-') ⟨*multiplicative-expr*⟩)*

The + and − operators with arithmetic type operands will yield same arithmetic type values. If one of the operands is integer and one is floating-point type the integer type operand will be first converted to a floating-point type. Resulting values with operands a and b will be $a + b$ and $a - b$ respectively, if possible overflow or loss of precision is not taken account.

For + and − operators it is also possible that the left operand is of a pointer type (p) and the right operand is of integer type (i). In that case the expression $p+i$ yields a pointer value pointing i objects after p in memory and expression $p - i$ yields a pointer value pointing i objects before p in memory.

For − operator it is also possible that both operands are of a pointer type (p and q). In that case the expression $p - q$ yields an integer counting the number of objects between pointers p and q.

A user-defined class can overload the + and − operators by implement the **operator**+ and **operator**− functions respectively.

4.12 Multiplicative Expression

⟨*multiplicative-expr*⟩ ::= ⟨*prefix-expr*⟩4.13 (('*' | `/` | `%`) ⟨*prefix-expr*⟩)*

The ∗ and / operators with arithmetic type operands will yield same arithmetic type values. They groups operands from left to right. If one of the operands is integer and one is floating-point type the integer type operand will be first converted to a floating-point type. Resulting values with operands a and b will be ab and a/b respectively, if possible overflow or loss of precision is not taken account.

The % operator with integer operands will yield integer type values. It groups operands from left to right. The resulting value will be remainder of integer division a/b.

A user-defined class can overload the ∗, / and % operators by implement the **operator**∗, **operator**/, and **operator**% functions respectively.

4.13 Prefix Expression

⟨*prefix-expr*⟩ ::= (`++` | `--` | `+` | '-' | `!` | `~` | `&` | '*') ⟨*prefix-expr*⟩ | ⟨*postfix-expr*⟩4.14

The prefix ++ and −− operators can have an integer or pointer type operand. The operand shall be an lvalue [3]. When the operand is of integer type, the ++ operator will increment its operand by 1 and the operator −− will decrement its operand by 1. When the operand is of pointer type with pointer p pointing to an object of type T, the expression $++p$

[3] A variable, a reference or a result of a dereference operator.

increments pointer p so that p points to next object of type T in memory. Correspondingly $--p$ decrements pointer p so that p points to previous object of type T in memory. The result of an increment or decrement operator is a value of the same type as the operand after incrementing or decrementing it.

The prefix + and - operators can have an arithmetic type operand. Expression $+p$ will yield p and $-p$ will yield the negation of p.

The prefix ! operator can have a boolean operand. The result of !**true** is **false** and !**false** is **true**.

The $\sim$ operator can have an integer operand. It will yield a bitwise complement of its operand.

The & operator shall have an lvalue operand. If the operand is an object of type is T, it returns an address or memory location of the object. The resulting value will be a pointer pointing to the object and it will have a type T$*$.

The $*$ operator can have a pointer operand. If the operand p is of a type pointer to T, the expression $*p$ dereferences p yielding an lvalue of type T.

A user-defined class can overload any prefix operator *op* by implement the **operator*op*** function.

4.14 Postfix Expression

⟨*postfix-expr*⟩ ::= ⟨*primary-expr*⟩4.15 (`++`4.14.1 | `--`4.14.1
| `.`4.14.2 ⟨*identifier*⟩2.3
| `->`4.14.3 ⟨*identifier*⟩
| '`(`' ⟨*argument-list*⟩ '`)`'4.14.4
| '`[`' ⟨*expression*⟩4 '`]`'4.14.5)*

⟨*argument-list*⟩ ::= ⟨*expression-list*⟩?

⟨*expression-list*⟩ ::= ⟨*expression*⟩ (`,` ⟨*expression*⟩)*

4.14.1 Postfix Increment and Decrement Operators

The postfix ++ and −− operators can have an integer or pointer type operand. The operand shall be an lvalue. The result of the operator is the value of the operand before incrementing or decrementing it. When the operand is of integer type, the ++ operator will increment its operand by 1 and the operator −− will decrement its operand by 1. When the operand is of pointer type with pointer p pointing to an object of type T, the expression $p++$ increments pointer p so that p points to next object of type T in memory. Correspondingly $p--$ decrements pointer p so that p points to previous object of type T in memory.

A user-defined class cannot overload the postfix forms of ++ or −− operators. They are automatically implemented by the compiler if the prefix versions of those operators are implemented.

4.14.2 Member Access Operator

The member access operator . will access the member named by the right operand from the entity specified by the left operand. The left operand can be a namespace, a class type, or an enumerated type. The type of the expression is the type of the right operand.

4.14.3 Pointer Member Access Operator

The pointer member access operator $->$ will access a member named by the right operand from the entity specified by the left operand. The left operand can be a pointer type or class type object. In the latter case the class type shall implement the **operator**$->$ function that returns a pointer or another class type object. The type of the expression is the type of the right operand.

4.14.4 Invocation Operator

The invocation operator () will invoke a function specified by the left operand with the specified list of arguments. The left operand can be a function, a member, an object of a class type that implements the **operator**() function or a type. The type of the expression is the return type of the function.

4.14.5 Indexing Operator

The the left operand of the indexing operator [] shall be of a pointer type, an array type or a class type object. When the left operand is pointer p, the expression $p[i]$ is evaluated as *(p + i). When the left operand is an array, the expression $a[i]$ returns a reference to the i'th element of array a, where i can range 0 to N - 1, where N is the dimension of the array a (see 4.21.2). When the left operand is of a class type, it shall implement the **operator**[] function taking one parameter.

4.15 Primary Expression

⟨*primary-expr*⟩ ::= '(' ⟨*expression*⟩4 ')'
| ⟨*literal*⟩2.4
| ⟨*basic-type*⟩3
| ⟨*identifier*⟩2.3
| `typename` '(' ⟨*expression*⟩ ')'
| `this`
| `base`
| ⟨*sizeof-expr*⟩4.16
| ⟨*cast-expr*⟩4.17
| ⟨*new-expr*⟩4.18
| ⟨*construct-expr*⟩4.19
| ⟨*template-id*⟩4.20

An expression in parenthesis, a literal (2.4), a name of a basic type (3) and an identifier are primary expressions.

Keyword **typename** followed by an expression in parenthesis yields the full name of the dynamic type of the expression. The type of it is **const char***.

Keyword **this** represents a pointer to the current class object in class member functions.

Keyword **base** represents a pointer to the base class object in class member functions.

4.16 Size-of Expression

⟨*sizeof-expr*⟩ ::= `sizeof` '(' ⟨*expression*⟩4 ')'

The operand of the **sizeof** operator can be a type or an entity having a type. If the type in question is T, the operator yields constant of type **ulong** that is equal to the size of object of type T in bytes.

4.17 Cast Expression

⟨*cast-expr*⟩ ::= `cast` `<` ⟨*type-expression*⟩4.21 `>` '(' ⟨*expression*⟩4 ')'

The **cast** operator performs explicit type conversion. In order to the cast operation to succeeed, there shall be a built-in conversion, a conversion function(9.2.11) in the source type, or a converting constructor in the target type taking one parameter of type of the source expression.

4.18 New Expression

⟨*new-expr*⟩ ::= `new` ⟨*type-expression*⟩4.21 '(' ⟨*argument-list*⟩4.14 ')'

The **new** operator reserves memory from the free store for an object of the requested type, constructs the object by issuing the **construct** operator with the specified arguments, and returns a pointer to the newly constructed object.

4.19 Construct Expression

⟨*construct-expr*⟩ ::= `construct` `<` ⟨*type-expression*⟩ `>`
'(' ⟨*expression*⟩4 (`,` ⟨*expression-list*⟩4.14)? ')'

The **construct** operator takes a type, a generic pointer to memory and constructor arguments. It constructs an object of requested type to the memory by calling the constructor with arguments, and returns a pointer to the newly constructed object.

4.20 Template Identifier

⟨*template-id*⟩ ::= ⟨*function-or-class-name*⟩ `<` ⟨*type-expression*⟩4.21 (`,` ⟨*type-expression*⟩)* `>`

⟨*function-or-class-name*⟩ ::= ⟨*qualified-id*⟩2.3

A template identifier is a primary expression.

4.21 Type Expression

⟨*type-expression*⟩ ::= ⟨*prefix-type-expr*⟩

4.21.1 Prefix Type Expression

⟨*prefix-type-expr*⟩ ::= `const` ⟨*postfix-type-expr*⟩
| ⟨*postfix-type-expr*⟩

A prefix type expression is either a **const** qualified postfix type expression, or sole postfix type expression.

4.21.2 Postfix Type Expression

⟨*postfix-type-expr*⟩ ::= ⟨*primary-type-expr*⟩ ('.' ⟨*identifier*⟩2.3
| `&&`
| `&`
| `*`
| `[` ⟨*array-dimension*⟩ `]`)*

⟨*array-dimension*⟩ ::= ⟨*constant-expression*⟩4.22

A postfix type expression is a primary type expression optionally followed by sequence of the following: a member selection operator '.' and an identifier, an rvalue reference type operator (see 8.3), an lvalue reference type operator (see 8.2), a pointer type operator (see 8.1), or an array type operator (see 8.4).

4.21.3 Primary Type Expression

⟨*primary-type-expr*⟩ ::= ⟨*basic-type*⟩3
| ⟨*template-id*⟩4.20
| ⟨*identifier*⟩
| '(' ⟨*prefix-type-expr*⟩ ')'

A primary type expression consists of basic types, template identifiers, identifiers and parenthesized prefix type expressions.

4.22 Constant Expression

⟨*constant-expression*⟩ ::= ⟨*expression*⟩4

A constant expression is an expression whose value can be obtained at compile time. A constant expression can contain literals, constants, enumeration constants and operators that do not involve taking an address of an object.

5 Statements

Statements control the program execution logic, declare type aliases, evaluate expressions, assign values to variables, or construct or destroy them. A statement can be labeled by an identifier.

⟨*statement*⟩ ::= (⟨*identifier*⟩ `:`)? (⟨*control-statement*⟩5.1
| ⟨*typedef-statement*⟩5.2
| ⟨*simple-statement*⟩5.3
| ⟨*assignment-statement*⟩5.4
| ⟨*construction-statement*⟩5.5
| ⟨*delete-statement*⟩5.6
| ⟨*destroy-statement*⟩5.7
| ⟨*throw-statement*⟩5.8
| ⟨*try-catch-statement*⟩5.9
| ⟨*assert-statement*⟩5.10
| ⟨*conditional-compilation-statement*⟩)5.11

5.1 Control Statements

The control statements control statement execution order.

⟨*control-statement*⟩ ::= ⟨*return-statement*⟩5.1.1
| ⟨*conditional-statement*⟩5.1.2
| ⟨*switch-statement*⟩5.1.3
| ⟨*while-statement*⟩5.1.4
| ⟨*do-statement*⟩5.1.5
| ⟨*range-for-statement*⟩5.1.6
| ⟨*for-statement*⟩5.1.7
| ⟨*compound-statement*⟩5.1.8
| ⟨*break-statement*⟩5.1.9
| ⟨*continue-statement*⟩5.1.10
| ⟨*goto-statement*⟩5.1.11

5.1.1 Return Statement

The **return** statement returns control from a function to its caller. When the return type of the function is other than **void** the return statement is mandatory, and there shall be a return value expression, otherwise there shall be no return value expression, and the return statement is optional.

⟨*return-statement*⟩ ::= `return` ⟨*expression*⟩4? `;`

5.1.2 Conditional Statement

The conditional statement executes a statement if its condition expression is **true**. Otherwise, if there is an else statement, it is executed. The condition shall be a Boolean expression.

⟨*conditional-statement*⟩ ::= `if` '(' ⟨*expression*⟩4 ')' ⟨*statement*⟩5 (`else` ⟨*statement*⟩)?

5.1.3 Switch Statement

The **switch** statement evaluates its condition expression and executes the matching case statement. Each control path of the **case** statement shall end with **break**, **goto case** or **goto default** statement, that is: the control is not allowed to "fall through". Multiple cases can be grouped into one **case** statement, though. The condition expression and each case expression shall evaluate to a compile time constant.

⟨*switch-statement*⟩ ::= `switch` '(' ⟨*constant-expression*⟩4.22 ')'
'{' (⟨*case-statement*⟩ | ⟨*default-statement*⟩)* '}'

⟨*case-statement*⟩ ::= (`case` ⟨*constant-expression*⟩ `:`)+
((⟨*identifier*⟩ `:`)? (⟨*goto-case-statement*⟩ | ⟨*goto-default-statement*⟩) | ⟨*statement*⟩5)*

⟨*goto-case-statement*⟩ ::= `goto case` ⟨*constant-expression*⟩ `;`

⟨*goto-default-statement*⟩ ::= `goto default ;`

⟨*default-statement*⟩ ::= `default :` ((⟨*identifier*⟩ `:`)? ⟨*goto-case-statement*⟩) | ⟨*statement*⟩)*

5.1.4 While Statement

The **while** statement repeatedly evaluates its condition expression and executes a statement as long as the condition evaluates to **true**. The condition shall be a Boolean expression.

⟨*while-statement*⟩ ::= `while` '(' ⟨*expression*⟩4 ')' ⟨*statement*⟩5

5.1.5 Do Statement

The **do** statement repeatedly executes a statement and then evaluates its condition expression until the condition evaluates to **false**. The condition shall be a Boolean expression.

⟨*do-statement*⟩ ::= `do` ⟨*statement*⟩5 `while` '(' ⟨*expression*⟩4 ')' `;`

5.1.6 Range-for Statement

The **range-for** statement iterates through a container. For each iteration, a variable is bound to each value in the container and then a statement is executed. The *container* shall be an expression that will yield an object that is a container having *Begin()* and *End()* member functions that return iterators to the beginning and one-past-the-end of a container respectively.

⟨*range-for-statement*⟩ ::= `for` '(' ⟨*type-expression*⟩4.21 ⟨*identifier*⟩2.3 ':' ⟨*container*⟩ ')' ⟨*statement*⟩5

⟨*container*⟩ ::= ⟨*expression*⟩4

5.1.7 For Statement

The **for** statement first initializes a variable and then repeatedly evaluates a condition expression and executes a statement as long as the condition evaluates to **true**. After each execution cycle an expression that typically increments a condition variable is evaluated. The condition shall be a Boolean expression. The initialization statement is optional as are the condition and increment expressions in which case they are evaluated as **true**.

⟨*for-statement*⟩ ::= `for` '(' ⟨*for-init-statement*⟩ ⟨*expression*⟩4? `;` ⟨*expression*⟩? ')' ⟨*statement*⟩5

⟨*for-init-statement*⟩ ::= ⟨*assignment-statement*⟩5.4 | ⟨*construction-statement*⟩5.5 | `;`

5.1.8 Compound Statement

The compound statement executes a sequence of statements in order. The statement sequence can be empty.

⟨*compound-statement*⟩ ::= '{' ⟨*statement*⟩5* '}'

5.1.9 Break Statement

The **break** statement transfers control to the statement coming after its closest containing **switch**, **while**, **do** or **for** statement, if there is one; otherwise to the end of the function.

⟨*break-statement*⟩ ::= `break ;`

5.1.10 Continue Statement

The **continue** statement transfers control to the end of the closest containing **while**, **do** or **for** loop.

⟨*continue-statement*⟩ ::= `continue ;`

5.1.11 Goto Statement

The **goto** statement transfers control to the statement labeled with the matching identifier.

⟨*goto-statement*⟩ ::= `goto` ⟨*identifier*⟩2.3 `;`

5.2 Typedef Statement

The **typedef** statement declares an alias name to a type expression.

⟨*typedef-statement*⟩ ::= `typedef` ⟨*type-expression*⟩4.21 ⟨*identifier*⟩2.3 `;`

5.3 Simple Statement

The simple statement evaluates an expression if there is one, otherwise it does nothing. Typically the expression is a function call, or incrementation or decrementation of a variable.

⟨*simple-statement*⟩ ::= ⟨*expression*⟩4? `;`

5.4 Assignment Statement

The assignment statement assign a value to an target object. The target expression shall be an lvalue expression.

⟨*assignment-statement*⟩ ::= ⟨*expression*⟩4 `=` ⟨*expression*⟩ `;`

5.5 Construction Statement

The construction statement constructs a local variable. If the variable is not of a class type, it fits into a register, its address is not taken, and it is not passed by reference argument, a register for the variable is allocated, otherwise space for the variable is reserved from the execution stack. If an initialization expression or expression list is given, the constructor for the variable is called with the given arguments, otherwise the variable is default-constructed.

⟨*construction-statement*⟩ ::= ⟨*type-expression*⟩4.21 ⟨*identifier*⟩2.3 ⟨*initialization*⟩? `;`

⟨*initialization*⟩ ::= '(' ⟨*expression-list*⟩4.14 ')' | `=` ⟨*expression*⟩4

5.6 Delete Statement

The **delete** statement calls the destructor of an object if there is one, and then releases the memory reserved for the object back to the free store. The expression shall evaluate to a pointer to the object. If the object has a virtual destructor, it is called, otherwise the destructor for the pointee type of the pointer type is called.

⟨*delete-statement*⟩ := `delete` ⟨*expression*⟩4 `;`

5.7 Destroy Statement

The **destroy** statement calls the destructor for an object but does not release memory reserved for the object. The expression shall evaluate to a pointer to the object. If the object has a virtual destructor, it is called, otherwise the destructor for the pointee type of the pointer type is called.

⟨*destroy-statement*⟩ ::= `destroy` ⟨*expression*⟩4 `;`

5.8 Throw Statement

The **throw** statement causes an exception object to be thrown. The expression shall construct a class object to throw. The exception object includes the source line number and the source file name of the throw statement.

⟨*throw-statement*⟩ ::= `throw` ⟨*expression*⟩4 `;`

5.9 Try-Catch Statement

The **try-catch** statement executes statements in a try-block and if an exception is thrown, control is transfered to a matching exception handler. Each exception handler handles exceptions that are the same type as or a type derived from the type of the variable of the exception handler. First matching handler is executed, so the handlers should be ordered from the most specific to the most general.

⟨*try-catch-statement*⟩ ::= `try` ⟨*try-block*⟩ ⟨*exception-handler*⟩+

⟨*try-block*⟩ ::= ⟨*compound-statement*⟩5.1.8

⟨*exception-handler*⟩ ::= `catch` '(' ⟨*type-expression*⟩4.21 ⟨*identifier*⟩2.3 ')' ⟨*catch-block*⟩

⟨*catch-block*⟩ ::= ⟨*compound-statement*⟩

5.10 Assert Statement

The **# assert** statement tests a condition that should always be true. If the condition is false, an error message containing the failed expression, a source line number and a source file name is issued and the program is exited.

Assert statements are skipped when compiling a program using **release** configuration.

⟨*assert-statement*⟩ ::= `'#' assert` '(' ⟨*expression*⟩4 ')' ';'

5.11 Conditional Compilation Statement

The conditional compilation statement makes it possible to include statements in compilation based on conditional compilation symbols.

When compiling a program using **debug** configuration (default), the symbol DEBUG is defined. When compiling a program using **release** configuration, the symbol RELEASE is defined. When compiling a program using Cmajor compiler in Windows, a symbol WINDOWS is defined. When compiling a program using Cmajor compiler in Linux, a symbol LINUX is defined.

Users can define other conditional compilation symbols in IDE or using the -D command line option.

⟨*conditional-compilation-statement*⟩ ::= ⟨*cond-comp-if-statememt*⟩

⟨*cond-comp-if-statememt*⟩ ::= '#' `if` '(' cond-comp-expr ')' ⟨*statement-list*⟩
('#' `elif` '(' cond-comp-expr ')' ⟨*statement-list*⟩)*
('#' `else` ⟨*statement_list*⟩)? '#' `endif`

⟨*cond-comp-expr*⟩ ::= ⟨*cond-comp-disjunction*⟩

⟨*cond-comp-disjunction*⟩ ::= ⟨*cond-comp-conjunction*⟩ (`||` ⟨*cond-comp-conjunction*⟩)*

⟨*cond-comp-conjunction*⟩ ::= ⟨*cond-comp-prefix*⟩ (`&&` ⟨*cond-comp-prefix*⟩)*

⟨*cond-comp-prefix*⟩ ::= ⟨*cond-comp-not*⟩ | ⟨*cond-comp-primary*⟩

⟨*cond-comp-not*⟩ ::= '!' ⟨*cond-comp-prefix*⟩

⟨*cond-comp-primary*⟩ ::= ⟨*identifier*⟩2.3 | '(' ⟨*cond-comp-expr*⟩ ')'

⟨*statement-list*⟩ ::= ⟨*statement*⟩5*

The **cond-comp-if** statement includes a list of statements in compilation if its conditional compilation expression evaluates true, otherwise, if one of the **elif** conditional compilation expressions evaluates true, the statements in that **elif**-part is included in compilation, otherwise, if the **else**-part is given, the list of statements in **else**-part is included to compilation.

Conditional compilation disjunction expression evaluates to true, if one of conditional compilation conjunction expressions it contains evaluates to true.

Conditional compilation conjunction expression evaluates to true, if all conditional compilation prefix expressions it contains evaluate to true.

Conditional compilation prefix expression can be either conditional compilation not-expression or conditional compilation primary expression.

Conditional compilation not-expression evaluates to true if its conditional compilation prefix expression evaluates to false.

Conditional compilation primary expression can be an conditional compilation identifier or parenthesized conditional compilation expression.

Conditional compilation identifier evaluates to true if it is defined.

6 Access Specifiers

Most entities defined within a class or a namespace can be given access specifiers that grant or reject access for another entity.

- **public** access grants access from everywhere.
- **protected** access specifier grants access for an entity defined within the same or derived class and rejects it for other entities.

- **private** access specifier grants access for an entity defined within the same class and rejects it for other entities.
- **internal** access specifier grants access for an entity defined in the same program or library and rejects it for other entities.

⟨*access-specifiers*⟩ ::= ⟨*access-specifier*⟩*

⟨*access-specifier*⟩ ::= `public` | `protected` | `private` | `internal`

If no access specifiers are given, the default access for a namespace level entity is **internal** access, and for a class level entity it is **private** access.

7 Functions

A function represents a unit of computation that can be executed from the other parts of the program or from the function body itself. When the function is called, each of the argument expressions of the function call is evaluated and bound to the corresponding parameter of the function. Then the function body is executed and after that the control returns to the caller.

⟨*function-definition*⟩ ::= ⟨*function-specifiers*⟩ ⟨*type-expression*⟩4.21 ⟨*function-name*⟩ ⟨*template-parameter-list*⟩? ⟨*parameters*⟩ ⟨*where-constraint*⟩10.5? ⟨*function-body*⟩

⟨*function-specifiers*⟩ ::= (⟨*access-specifier*⟩6 | `inline` | `cdecl` | `nothrow` | `throw` | `unit_test`)*

⟨*function-name*⟩ ::= ⟨*identifier*⟩2.3 | ⟨*operator-function-name*⟩

⟨*operator-function-name*⟩ ::= `operator` (`<<` | `>>` | `==` | `<` | `=` | `++` | `--` | `->` | `+` | '-' | `*` | `/` | `%` | `&` | '|' | `!` | `~` | `[]` | `()`)

⟨*template-parameter-list*⟩ ::= `<` ⟨*template-parameter*⟩ (`,` ⟨*template-parameter*⟩)* `>`

⟨*template-parameter*⟩ ::= ⟨*identifier*⟩ ⟨*default-template-parameter*⟩?

⟨*default-template-parameter*⟩ ::= '=' ⟨*type-expression*⟩

⟨*parameters*⟩ ::= '(' ⟨*parameter-list*⟩? ')'

⟨*parameter-list*⟩ ::= ⟨*parameter*⟩ (`,` ⟨*parameter*⟩)*

⟨*parameter*⟩ ::= ⟨*type-expression*⟩ ⟨*identifier*⟩?

⟨*function-body*⟩ ::= ⟨*compound-statement*⟩5.1.8

7.1 Function Specifiers

7.1.1 Inline Specifier

inline specifier is a hint to optimizer that it should replace function body in place of function call when applicable.

7.1.2 CDecl Specifier

cdecl specifier suppresses name mangling [4], so that the function can easily called from C language.

7.1.3 Nothrow Specifier

nothrow specifier states that the function will not throw Cmajor exceptions directly or indirectly, or that the function handles all exceptions (catches *System.Exception*) and does not deliberately throw any exception from catch blocks. [5]

7.1.4 Throw Specifier

throw specifier emphasizes that the function might throw a Cmajor exception.

7.1.5 Unit Test Specifier

unit_test specifier has a meaning in programs intended as input to the **cmunit** program.

7.2 Function Templates

A function that contains a template parameter list is said to be a *function template*. A function template is parameterized by one or more type parameters. An optional default value can be provided for a template parameter. There cannot be a non-default template parameter coming after a default template parameter.

7.2.1 Constrained Function Templates

A function template that has a where-constraint (10.5) is said to be a *constrained function template*. The where-constraint sets requirements for template type parameters.

7.2.2 Function Template and Overload Resolution

Function templates participate overload resolution process. In overload resolution the template parameters are deduced from the types of function call arguments. The type parameters can also be explicitly specified using a template identifier (4.20). After the type parameters are deduced or explicitly specified, the function template is instantiated, and it becomes a concrete function. The type-checking of an unconstrained function template is deferred until instantiation time.

8 Derived Types

8.1 Pointer Types

If τ is a type expression (4.21), $\tau*$ creates a new type representing a pointer to an object of type τ. The default value of a pointer type object is **null**.

[4] name mangling encodes parameter types to the function name
[5] semantics changed from version 2.0.

Unary operators $*$ and $->$, prefix operators $++$ and $--$, binary operators $+$, $-$, $==$ and $<$, and the indexing operator [] operate on pointer type operands.

A pointer type value can be implicitly converted to a generic pointer type **void**$*$.

8.1.1 Generic Pointer Type

The type **void**$*$ represents a generic pointer to a memory location or it is **null**. The memory location can hold an object or it can be a location in the free store that may or may not hold an object. The default value of a generic pointer type object is **null**.

Generic pointers can be compared for equality using the $==$ operator. The generic pointer type can be converted to any other pointer type with an explicit **cast**.

8.1.2 Null Pointer Type

The null pointer type represents a pointer that does not point to any object or any location in the free store. The value of a null pointer type object is **null**. Any pointer type can be compared with **null** for equality using the $==$ operator.

8.2 Lvalue Reference Types

If τ is a type expression (4.21), τ& creates a new type representing an lvalue reference to an object of type τ. Lvalue reference type objects do not have a default value and they must be initialized to refer to an object in some memory location.

If T is an object type, type T& supports the same operators as T does.

Type T& (an lvalue reference to an object of type T) can be implicitly converted to type **const** T& (an lvalue reference to a constant object of type T).

8.3 Rvalue Reference Types

If τ is a type expression (4.21), τ&& creates a new type representing an rvalue reference to an object of type τ. Rvalue references are used to implement move semantics.

8.4 Array Types

If τ is a type expression (4.21), $\tau[N]$ creates a new type representing an array whose element type is τ and dimension is N.

Currently only single-dimensional arrays whose element type is a basic type (3) or a class type (9.2) are supported. The dimension N must be a constant expression (4.22).

Indexing operator [] operates on array type operands (4.14.5).

8.5 Constant Types

If τ is a type expression, **const** τ creates a new type representing a constant object of type τ. The default value of type **const** τ is the same as the default value of type τ.

If T is an object type, type **const** T supports those operators of type T that do not change the value of object of type T.

Type T can be implicitly converted to type **const** T.

9 User-defined Types

9.1 Enumerated Types

An enumerated type is used to create an integral type with a set of named constants. Each constant will have a value of type **int** that is distinct from the values of other constants within the same enumerated type.

⟨*enumerated-type-definition*⟩ ::= ⟨*access-specifiers*⟩6 `enum` ⟨*identifier*⟩2.3
'`(`' ⟨*enumeration-constant*⟩ (`,` ⟨*enumeration-constant*⟩)* '`)`'

⟨*enumeration-constant*⟩ ::= ⟨*identifier*⟩ (`=` ⟨*constant-expression*⟩4.22)?

The default value of an enumerated type object is 0, whether it is represented by a named constant or not.

If the first constant is not explicitly given a value, it will have a value 0. If any other constant is not explicitly given a value, it will have the value of the previous constant + 1.

9.2 Class Types

A class type is used to define a user-defined type that can contain data members, type aliases, constants, member functions and subtypes.

⟨*class-definition*⟩ ::= ⟨*class-specifiers*⟩ `class` ⟨*identifier*⟩2.3 ⟨*template-parameter-list*⟩7?
⟨*inheritance*⟩? ⟨*where-constraint*⟩10.5? '{' ⟨*class-content*⟩ '}'

⟨*class-specifiers*⟩ ::= (⟨*access-specifier*⟩6 | `static` | `abstract`)*

⟨*inheritance*⟩ ::= `:` (⟨*template-id*⟩4.20 | ⟨*qualified-id*⟩2.3)

⟨*class-content*⟩ ::= (⟨*static-constructor*⟩9.2.7
| ⟨*constructor*⟩9.2.8
| ⟨*destructor*⟩9.2.9
| ⟨*member-function*⟩9.2.10
| ⟨*conversion-function*⟩9.2.11
| ⟨*enumerated-type-definition*⟩9.1
| ⟨*constant-declaration*⟩9.2.13
| ⟨*member-variable-declaration*⟩9.2.12
| ⟨*typedef-declaration*⟩9.2.14
| ⟨*delegate-definition*⟩9.3.1
| ⟨*class-delegate-definition*⟩9.3.2
| ⟨*class-definition*⟩)*

A class type can be regular, **static** or **abstract**.

9.2.1 Regular Class Types

A regular class can contain each kind of class content members.

9.2.2 Static Class Types

A **static** class can contain

- a static constructor
- static data members
- static member functions
- constants
- typedef declarations
- delegate and class delegate definitions
- subtypes

9.2.3 Abstract Class Types

A class that contains one or more abstract member functions shall be declared **abstract**. One cannot create an object of an abstract class. Each concrete class that derives from an abstract class shall override each of the abstract member functions of an abstract base class.

9.2.4 Class Templates

A class that contains a template parameter list is said to be a *class template*. A class template is parameterized by one or more type parameters. When a template name (4.20) is specified with the type argument expressions, the class template is instantiated, and becomes a concrete class. The type-checking of a class template is deferred until instantiation time.

9.2.5 Inheritance

A class or a class template can derive from a class or a class template. Derived class *inherits* members of the *base class*. A derived class can *override* virtual, overridden and abstract member functions of the base class.

9.2.6 Constrained Class Templates

A class template that has a where-constraint (10.5) is said to a *constrained class template*. The where-constraint sets requirements for template type parameters.

9.2.7 Static Constructor

The identifier of the static constructor shall be equal to the name of the class type. The purpose of the static constructor is to initialize the static member variables of a class object on its first call. The static constructor is called from each constructor and each static member function of a class type before taking other actions.

⟨*static-constructor*⟩ ::= `static` ⟨*static-constructor-class-name*⟩ '(' ')'
(`:` ⟨*static-initializer-list*⟩)? ⟨*function-body*⟩7

⟨*static-constructor-class-name*⟩ ::= ⟨*identifier*⟩2.3

⟨*static-initializer-list*⟩ ::= ⟨*static-initializer*⟩ (',' ⟨*static-initializer*⟩)*

⟨*static-initializer*⟩ ::= ⟨*identifier*⟩ '(' ⟨*argument-list*⟩4.14 ')'

9.2.8 Constructors

The identifier of the constructor shall be equal to the name of the class type. The purpose of a constructor is to initialize non-static member variables of a class type, and to establish a class invariant.

An initializer of a constructor can delegate initialization to another constructor of the same class using the **this** keyword. The initializer can call a base class constructor using the **base** keyword.

If a non-static class has no user-defined constructor, the compiler will implement a default constructor that constructs each non-static member variable to its default value.

If a non-static class has no user-defined copy constructor, no user-defined move constructor, no user-defined copy assignment, no user-defined move assignment and no user-defined destructor, the compiler will implement a copy constructor that copy-constructs each non-static member variable from the corresponding member variable of the constructor argument. The compiler will also implement a move constructor that moves each non-static member variable from the corresponding member variable of the constructor argument in that case.

This automatic generation of default constructor, copy constructor and move constructor can be suppressed using the **suppress** keyword.

Automatic generation of default constructor, copy constructor and move constructor can be requested by using the **default** keyword.

⟨*constructor*⟩ ::= ⟨*constructor-specifiers*⟩ ⟨*constructor-class-name*⟩
'(' ⟨*parameter-list*⟩7? ')' (`:` ⟨*initializer-list*⟩)? ⟨*function-body*⟩7

⟨*constructor-specifiers*⟩ ::= (⟨*access-specifier*⟩6 | `suppress` | `default` | `inline` | `nothrow` | `throw`)*

⟨*constructor-class-name*⟩ ::= ⟨*identifier*⟩2.3

⟨*initializer-list*⟩ ::= ⟨*initializer*⟩ (',' ⟨*initializer*⟩)*

⟨*initializer*⟩ ::= (⟨*identifier*⟩ | `this` | `base`) '(' ⟨*argument-list*⟩4.14 ')'

```
1  public class Foo
2  {
3      public Foo()                    // default constructor
4      {
5          //  ...
6      }
7      public Foo(const Foo& that) // copy constructor
8      {
9          //  ...
10     }
```

```
11      public Foo(Foo&& that)        // move constructor
12      {
13          //  ...
14      }
15  }
16
17  public class Bar
18  {
19      default Bar();                // compiler will implement memberwise
            default constructor
20      default Bar(const Bar&);      // compiler will implement memberwise
            copy constructor
21      default Bar(Bar&&);           // compiler will implement memberwise
            move constructor
22  }
23
24  public class Baz
25  {
26      suppress Baz();               // automatic generation of default
            constructor is suppressed
27      suppress Baz(const Baz&);     // automatic generation of copy
            constructor is suppressed
28      suppress Baz(Baz&&);          // automatic generation of move
            constructor is suppressed
29  }
```

9.2.9 Destructor

The identifier of the destructor shall be equal to the name of the class type. The purpose of a destructor is to destroy non-static member variables of a class type, and to tear down the class invariant.

⟨*destructor*⟩ ::= ⟨*destructor-specifiers*⟩ ~ ⟨*destructor-class-name*⟩ '(' ')' ⟨*function-body*⟩7

⟨*destructor-specifiers*⟩ ::= (⟨*access-specifier*⟩6 | `virtual` | `override` | `default` | `inline`)*

⟨*destructor-class-name*⟩ ::= ⟨*identifier*⟩2.3

If the class type has a base class that has a virtual destructor, the destructor shall be declared with override specifier; otherwise, if the class type contains virtual functions, the destructor is automatically set virtual.

If a non-static class has no user-defined destructor, no user-defined copy constructor, no user-defined move constructor, no user-defined copy assignment, no user-defined move assignment, and (1) the class has virtual functions, or (2) at least one member variable has a non-trivial destructor, or (3) a class has a base class that has a non-trivial destructor, the compiler will implement a destructor that destroys all non-static member variables.

Automatic generation of destructor can be requested by using the **default** keyword.

```
1   public class Foo
2   {
3       public ~Foo()    // destructor
```

```
4       {
5           // ...
6       }
7   }
8
9   public class Bar
10  {
11      default ~Bar(); // compiler will implement memberwise destructor
12  }
```

9.2.10 Member Functions

A member function operates on member variables a class object or class type and can call other member functions of its class. A member function can be virtual, overridden, abstract, static or regular.

⟨*member-function*⟩ ::= ⟨*member-function-specifiers*⟩ ⟨*type-expression*⟩4.21
⟨*function-name*⟩7 '(' ⟨*parameter-list*⟩7? ')' `const`? (⟨*function-body*⟩7 | `;`)

⟨*member-function-specifiers*⟩ ::= (⟨*access-specifier*⟩6
| `virtual` | `override` | `abstract` | `static` | `suppress` | `default` | `inline` | `nothrow` | `throw`)*

A regular member function may have access specifiers but no other member function specifiers. A regular member function operates on a class object.

Member function can be declared *virtual*, in which case the actual member function called through a pointer or reference variable depends on the actual type of the class object instead of the formal type of the pointer or reference variable.

Member function can be declared *abstract*, in which case it shall not have an implementation, and it has to be overridden in each concrete class type derived from the abstract class type.

A member function of a derived class type can *override* a virtual or abstract member function of a base class type.

Member function can be declared *static*, in which case it do not operate on a class object (it has no **this** pointer). A static member function can call other static member functions of its class and can operate on static member variables of its class. A static member function is called using ClassName.StaticMemberFunc(arguments) syntax.

A member function can be declared *const*, in which case the type of the **this** pointer for a class C is "pointer to **const** C". A constant member function cannot directly alter its class object.

If a non-static class has no user-defined copy assignment, no user-defined copy constructor, no user-defined move assignment, no user-defined move constructor and no user-defined destructor, the compiler will implement a copy assignment that assigns each non-static member variable from its corresponding member variable of the function argument. The compiler will also implement a move assignment that swaps each non-static member variable with the corresponding member variable of the function argument in that case.

This automatic generation of copy assignment and move assignment can be suppressed using the **suppress** keyword.

Automatic generation of copy assignment and move assignment can be requested by using the **default** keyword.

```
public class Foo
{
    public void operator=(const Foo& that)  // copy assignment
    {
        // ...
    }
    public void operator=(Foo&& that)        // move assignment
    {
        // ...
    }
}

public class Bar
{
    default void operator=(const Bar&); // compiler will implement
        memberwise copy assignment
    default void operator=(Bar&&);      // compiler will implement
        memberwise move assignment
}

public class Baz
{
    suppress void operator=(const Baz&);     // automatic generation of
        copy assignment is suppressed
    suppress void operator=(Baz&&);          // automatic generation of
        move assignment is suppressed
}
```

9.2.11 Conversion Functions

A conversion function converts a class type object to the specified target type. Conversion functions participate type conversions in cast-expressions(4.17) and in function overloading.

⟨*conversion-function*⟩ ::= ⟨*conversion-function-specifiers*⟩ `operator` ⟨*type-expression*⟩4.21 '(' ')' `const`? (⟨*function-body*⟩7 | `;`)

⟨*conversion-function-specifiers*⟩ ::= (⟨*access-specifier*⟩6 | `nothrow` | `throw` | `inline`)*

9.2.12 Member Variable Declarations

A class can have non-static and static member variables.

⟨*member-variable-declaration*⟩ ::= ⟨*member-variable-specifiers*⟩ ⟨*type-expression*⟩4.21 ⟨*identifier*⟩2.3 `;`

⟨*member-variable-specifiers*⟩ ::= (⟨*access-specifier*⟩6 | `static`)*

A non-static member variable is a member of each class object of its class.
A static member variable is a member of its class type.

9.2.13 Constant Declarations

A constant is an object whose value can be obtained at compile time.

⟨*constant-declaration*⟩ ::= ⟨*access-specifiers*⟩6
`const` ⟨*type-expression*⟩4.21 ⟨*identifier*⟩2.3 `=` ⟨*constant-expression*⟩4.22 `;`

The type of a constant can be a basic value type or an enumerated type (see 9.1).

9.2.14 Typedef Declarations

The **typedef** declaration inserts an alias name for a type expression into its scope.

⟨*typedef-declaration*⟩ ::= ⟨*access-specifiers*⟩6 `typedef` ⟨*type-expression*⟩4.21 ⟨*identifier*⟩2.3 `;`

9.3 Delegate Types

Delegate types make possible to delegate execution of a function to another entity. Delegates can be passed as parameters and called much the same way ordinary functions can be called. Delegates are typically used to implement callbacks and events.

Delegate types come in two flavors: regular delegates that capture a function pointer and class delegates that capture a pointer to a class object and a pointer to a member function.

9.3.1 Delegates

A delegate type is defined using the keyword **delegate** followed by a return type, the name of the delegate type and a list of function parameters. An object of a delegate type is a pointer to a free function or to a static member function.

⟨*delegate-definition*⟩ ::= (⟨*access-specifier*⟩6 | `throw` | `nothrow`)* `delegate`
⟨*type-expression*⟩4.21 ⟨*identifier*⟩2.3 ⟨*parameters*⟩7 `;`

9.3.2 Class Delegates

A class delegate type is defined using the keyword pair **class delegate** followed by a return type, the name of the class delegate type and a list of member function parameters. An object of a class delegate type is a pair consisting of a pointer to a class object and a pointer to a member function.

⟨*class-delegate-definition*⟩ ::= (⟨*access-specifier*⟩6 | `throw` | `nothrow`)* `class delegate`
⟨*type-expression*⟩4.21 ⟨*identifier*⟩2.3 ⟨*parameters*⟩7 `;`

10 Concepts

Concepts are used to set requirements for template type arguments in function and class templates. The concept design is influenced by the Concept Design for the STL ([6]).

A concept definition consists of the name of the concept followed by a list of type parameters. A concept body consists of syntactic requirements for type parameters that take form of constraints, and semantic requirements for type parameters that take form of axioms.

A concept can *refine* another concept by setting additional requirements and overriding existing requirements. When overriding existing requirements, the refining concept can set an associated type equal to a more constrained type, or to model a more constrained concept than in the refined concept.

⟨*concept-definition*⟩ ::= ⟨*access-specifiers*⟩6
    `concept` ⟨*identifier*⟩2.3 `<` ⟨*identifer*⟩ (`,` ⟨*identifier*⟩)* `>`
    ⟨*refinement*⟩? ⟨*where-constraint*⟩10.5? `{` ⟨*concept-body*⟩ `}`

⟨*refinement*⟩ ::= `:` ⟨*concept-name*⟩ `<` ⟨*identifer*⟩ (`,` ⟨*identifier*⟩)* `>`

⟨*concept-name*⟩ ::= ⟨*qualified-id*⟩2.3

⟨*concept-body*⟩ ::= (⟨*typename-constraint*⟩10.1
  | ⟨*signature-constraint*⟩10.2
  | ⟨*embedded-constraint*⟩10.3
  | ⟨*axiom*⟩10.4)*

10.1 Typename Constraint

A typename constraint sets a requirement that an associated type with the specified name is found.

⟨*typename-constraint*⟩ ::= `typename` ⟨*type-expression*⟩4.21 `;`

Typically type expression is of the form <*concept type parameter*> `.` <*associated type name*>.

10.2 Signature Constraint

A signature constraint sets a requirement that either the first concept type parameter contains a member function with the specified signature or there exists a nonmember function with the specified signature. The signature of the function does not have to match exactly.

⟨*signature-constraint*⟩ ::= ⟨*constructor-constraint*⟩
  | ⟨*destructor-constraint*⟩
  | ⟨*member-function-constraint*⟩
  | ⟨*function-constraint*⟩

⟨*constructor-constraint*⟩ ::= ⟨*identifier*⟩2.3 ⟨*parameters*⟩7 `;`

⟨*destructor-constraint*⟩ ::= `~` ⟨*identifier*⟩ '(' ')' `;`

⟨*member-function-constraint*⟩ ::= ⟨*type-expression*⟩4.21
    ⟨*identifier*⟩ `.` ⟨*function-name*⟩7 ⟨*parameters*⟩ `;`

⟨*function-constraint*⟩ ::= ⟨*type-expression*⟩4.21 ⟨*function-name*⟩7 ⟨*parameters*⟩ `;`

10.3 Embedded Constraint

An embedded constraint sets additional requirements for concept parameter types.

⟨*embedded-constraint*⟩ ::= ⟨*where-constraint*⟩10.5 `;`

10.4 Axioms

A concept can contain *axioms* that represent semantic requirements for constrained template type arguments. The axioms are not processed in any way by the compiler (except parsing their syntax). They are only documentation for the programmer.

⟨*axiom*⟩ ::= `axiom` ⟨*identifier*⟩?2.3 ⟨*parameters*⟩?7 `{` ⟨*axiom-body*⟩ `}`

⟨*axiom-body*⟩ ::= ⟨*axiom-statement*⟩*

⟨*axiom-statement*⟩ ::= ⟨*expression*⟩4 `;`

10.5 Where Constraint

A where constraint sets requirements for template type arguments or concept parameter types.

⟨*where-constraint*⟩ ::= `where` ⟨*constraint-expr*⟩

⟨*constraint-expr*⟩ ::= ⟨*disjunctive-constraint-expr*⟩10.5.1

10.5.1 Disjunctive Constraint Expressions

A disjunctive constraint expression states that the template type arguments and concept parameters must satisfy the requirements of one of the conjunctive constraint expressions separated by **or** connectives.

⟨*disjunctive-constraint-expr*⟩ ::= ⟨*conjunctive-constraint-expr*⟩10.5.2 (`or` ⟨*conjunctive-constraint-expr*⟩)*

10.5.2 Conjunctive Constraint Expressions

A conjunctive constraint expression states that the template type arguments and concept parameters must satisfy the requirements of all primary constraint expressions separated by **and** connectives.

⟨*conjunctive-constraint-expr*⟩ ::= ⟨*primary-constraint-expr*⟩10.5.3 (`and` ⟨*primary-constraint-expr*⟩)*

10.5.3 Primary Constraint Expressions

A primary constraint can be either an atomic constraint or a constraint expression enclosed in parenthesis.

⟨*primary-constraint-expr*⟩ ::= ⟨*atomic-constraint*⟩10.5.4 | '(' ⟨*constraint-expr*⟩10.5 ')'

10.5.4 Atomic Constraints

An atomic constraint can be either an is-constraint or a multiparameter constraint.

⟨*atomic-constraint*⟩ ::= ⟨*is-constraint*⟩10.5.5 | ⟨*multiparam-constraint*⟩10.5.6

10.5.5 Is-Constraint

An is-constraint sets a requirement that a either (1) the specified type satisfies the requirements of the specified concept, or (2) the specified type is trivially convertible to another type. For example, type **const T&** is trivially convertible to type **T**. When specifying the is-constraint in case (2), the so far less constrained type should be put to the left and the more constrained type to the right.

⟨*is-constraint*⟩ ::= ⟨*type-expression*⟩4.21 `is` ⟨*concept-or-typename*⟩

⟨*concept-or-typename*⟩ ::= ⟨*type-expression*⟩

10.5.6 Multiparameter Constraint

A multiparameter constraint sets a requirement that the specified types satisfy the requirements of the specified concept.

⟨*multiparam-constraint*⟩ ::= ⟨*concept-name*⟩10 `<` ⟨*type-expression*⟩4.21 (`,` ⟨*type-expression*⟩)*
`>`

10.6 Built-in Concepts

There are four built-in multiparameter concepts implemented by the compiler.

10.6.1 Same Concept

The **Same<T, U>** concept sets a requirement that types T and U are exactly the same type. The so far less constrained type should be put to the left and the more constrained type to the right.

10.6.2 Derived Concept

The **Derived<T, U>** concept sets a requirement that type T is derived from type U, or in other words that the type U is the base class of type T. Type T is trivially derived from type T.

10.6.3 Convertible Concept

The **Convertible<T, U>** concept sets a requirement that type T is convertible to type U. Either there should be a constructor with signature `U(const T&)` or there should be a built-in conversion from type T to type U. Type T is trivially convertible to type T.

10.6.4 Common Concept

The **Common<T, U>** concept sets a requirement that types T and U have a common type to which they both are convertible to. The common type is inserted as a typedef *CommonType* to the scope of concept **Common<T, U>**. The common type of T and T is trivially T. The so far less constrained type should be put to the left and the more constrained type to the right.

11 Namespaces

Namespaces are used to disambiguate equal names belonging to different libraries. A namespace consists of optional using directives followed by optional definitions.

⟨*namespace-definition*⟩ ::= `namespace` ⟨*qualified-id*⟩2.3 '{' ⟨*namespace-content*⟩ '}'

⟨*namespace-content*⟩ ::= ⟨*using-directive*⟩* ⟨*definition*⟩*

⟨*using-directive*⟩ ::= ⟨*using-alias-directive*⟩ | ⟨*using-namespace-directive*⟩

⟨*using-alias-directive*⟩ ::= `using` ⟨*identifier*⟩2.3 `=` ⟨*qualified-id*⟩ `;`

⟨*using-namespace-directive*⟩ ::= `using` ⟨*qualified-id*⟩ `;`

⟨*definition*⟩ ::= ⟨*enumerated-type-definition*⟩9.1
| ⟨*constant-declaration*⟩9.2.13
| ⟨*function-definition*⟩7
| ⟨*class-definition*⟩9.2
| ⟨*delegate-definition*⟩9.3.1
| ⟨*class-delegate-definition*⟩9.3.2
| ⟨*typedef-declaration*⟩9.2.14
| ⟨*concept-definition*⟩10
| ⟨*namespace-definition*⟩

A using alias directive brings a single alias name to the current file scope.
A using namespace directive brings contents of given namespace to the current file scope.

12 Source Files

A source file consists of namespace content.

⟨*source-file*⟩ ::= ⟨*namespace-content*⟩11

13 Programs

A Cmajor program consists of source files. One of the source files must contain a main function.

A main function can have four possible signatures:

- `void main()`
- `int main()`
- `void main(int argc, const char** argv)`
- `int main(int argc, const char** argv)`

In the two latter signatures parameter argc contains the number of program arguments, and parameter argv contains program argument strings. By convention the first argument (argv[0]) contains the name of the program.

If the main function is declared to return a value, the main function shall contain a return statement, otherwise it shall not contain a return statement.

If the main function is declared void, it returns exit code 0 to the environment.

14 Solution and Project File Formats

14.1 Solution File Format

⟨*solution-file*⟩ ::= ⟨*solution-declaration*⟩ ⟨*project-file-declarations*⟩ ⟨*project-dependencies*⟩

⟨*solution-declaration*⟩ ::= `solution` ⟨*solution-name*⟩ ';'

⟨*solution-name*⟩ ::= ⟨*qualified-id*⟩2.3

⟨*project-file-declarations*⟩ ::= ⟨*project-file-declaration*⟩*

⟨*project-file-declaration*⟩ ::= `project` ⟨*project-file-path*⟩ ';'

⟨*project-file-path*⟩ ::= `'<' [^>]+ '>'`

⟨*project-dependencies*⟩ ::= ⟨*project-dependency*⟩*

⟨*project-dependency*⟩ ::= `dependency` ⟨*project-name*⟩ ⟨*dependent-project-list*⟩ ';'

⟨*dependent-project-list*⟩ ::= '(' ⟨*dependent-project-name*⟩ (',' ⟨*dependent-project-name*⟩)* ')'

⟨*dependent-project-name*⟩ ::= ⟨*qualified-id*⟩

14.2 Project File Format

⟨*project-file*⟩ ::= ⟨*project-declaration*⟩ ⟨*project-file-declarations*⟩

⟨*project-declaration*⟩ ::= `project` ⟨*project-name*⟩ ';'

⟨*project-name*⟩ ::= ⟨*qualified-id*⟩2.3

⟨*project-file-declarations*⟩ ::= ⟨*project-file-declaration*⟩*

⟨*project-file-declaration*⟩ ::= ⟨*target-declaration*⟩
  | ⟨*assembly-declaration*⟩
  | ⟨*library-reference*⟩
  | ⟨*executable-declaration*⟩
  | ⟨*c-library-declaration*⟩
  | ⟨*add-library-path-declaration*⟩
  | ⟨*source-file*⟩
  | ⟨*asm-source-file*⟩
  | ⟨*c-source-file*⟩
  | ⟨*cpp-source-file*⟩
  | ⟨*text-file*⟩

⟨*target-declaration*⟩ ::= `target` '=' (`program` | `library`) ';'

⟨*assembly-declaration*⟩ ::= `assembly` ⟨*file-path*⟩ ⟨*properties*⟩? ';'

⟨*library-reference*⟩ ::= `reference` ⟨*file-path*⟩ ⟨*properties*⟩? ';'

⟨*executable-declaration*⟩ ::= `executable` ⟨*file-path*⟩ ';'

⟨*c-library-declaration*⟩ ::= `clib` ⟨*file-path*⟩ ⟨*properties*⟩? ';'

⟨*add-library-path-declaration*⟩ ::= `addlibrarypath` ⟨*file-path*⟩ ⟨*properties*⟩? ';'

⟨*source-file*⟩ ::= `source` ⟨*file-path*⟩ ⟨*properties*⟩? ';'

⟨*asm-source-file*⟩ ::= `asmsource` ⟨*file-path*⟩ ⟨*properties*⟩? ';'

⟨*c-source-file*⟩ ::= `csource` ⟨*file-path*⟩ ⟨*properties*⟩? ';'

⟨*cpp-source-file*⟩ ::= `cppsource` ⟨*file-path*⟩ ⟨*properties*⟩? ';'

⟨*text-file*⟩ ::= `text` ⟨*file-path*⟩ ⟨*properties*⟩? ';'

⟨*properties*⟩ ::= '[' ⟨*property*⟩ (',' ⟨*property*⟩)* ']'

⟨*property*⟩ ::= ⟨*property-name*⟩ '=' ⟨*propterty-value*⟩

⟨*property-name*⟩ ::= ⟨*identifier*⟩

⟨*property-value*⟩ ::= ⟨*identifier*⟩ | ⟨*long*⟩

⟨*file-path*⟩ ::= `'<' [^>]+ '>'`

Target declaration <*target-declaration*> sets whether we are building a program or a library.

Assembly declaration <*assembly-declaration*> sets the name of the archive that contains project's object code. By convention its extension is ".cma".

Library reference <*library-reference*> imports a Cmajor library to the project. By convention its extension is ".cml". The .cml file contains the metadata (symbol table etc.) of a Cmajor project.

If target is "program", an executable declaration <*executable-declaration*> sets the name of the executable file. In Windows ".exe" extension is appended automatically to the executable file name.

A C-library declaration <*c-library-declaration*> requests that named object code library must be linked to the final executable. Thus C-library declaration can be used in a library project or an executable project. Usually the object code library file is named "libfoo.a", but the link name needed here is just "foo", because the object code library is linked using gcc's -l option.

Library path declaration <*add-library-path-declaration*> adds a directory path to the library search paths when final executable is linked. Thus library path declaration can be used in a library project or an executable project. It is specified using gcc's -L option by the compiler when the program is linked.

Source file declaration <*source-file*> adds a Cmajor source file to the project. By convention its extension is ".cm".

Assembly source file declaration <*asm-source-file*> adds LLVM source file to the project. By its extension is ".ll". Assembly source file is compiled using the LLVM compiler **llc**.

C source file declaration <*c-source-file*> adds a C source file to the project. By convention its extension is ".c". C source file is compiled using GNU C compiler **gcc**.

C++ source file declaration <*cpp-source-file*> adds a C++ source file to the project. By convention its exension is ".cpp" or ".cxx". C++ source file is compiled using the GNU C++ compiler **g++**.

Text file declaration <*text-file*> includes some text file to the project. The text file can have any extension. It is not processed by the compiler.

Properties contains comma-separated list of property declarations. Properties can be used to include a declaration only in selected configurations. If the properties do not match current compilation properties, the associated declaration has no effect.

Currently property name can be **backend**, **os** or **bits**. Backend values can be **c** and **llvm**. Os values can be **windows** and **linux**. Bits values can be **32** and **64**.

References

[1] Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman: Compilers: Principles, Techniques, & Tools. Second Edition. Addison-Wesley, 2007.

[2] Boost C++ libraries, *http://www.boost.org/*

[3] Ellis, M. A., and B. Stroustrup: The Annotated C++ Reference Manual. Addison-Wesley, 1990.

[4] Gibbs, M., and B. Stroustrup: Fast dynamic casting, 2005, *http://www.stroustrup.com/fast_dynamic_casting.pdf*

[5] Han The Thanh: pdfTeX, *http://www.tug.org/applications/pdftex/*

[6] JTC1/SC22/WG21 - The C++ Standards Committee: A Concept Design for the STL, *http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf*

[7] JTC1/SC22/WG21 - The C++ Standards Committee: Working Draft, Standard for Programming Language C++, *http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf*

[8] Laakko, S.: Soul Parsing Framework, *http://sourceforge.net/projects/soulparsing/*

[9] Microsoft Corporation: The C# Language Specification. Version 4.0, *http://go.microsoft.com/fwlink/?LinkId=199552*

[10] Patterson, D. A., and J. L. Hennessy: Computer Organization and Design. The Hardware / Software Interface. Fourth Edition. Morgan Kaufmann, 2012.

[11] Stepanov, A., and P. McJones: Elements of Programming. Addison-Wesley, 2009.

[12] Stroustrup, B.: The C++ Programming Language. Fourth Edition. Addison-Wesley, 2013.

[13] Sundaresan V., and C. Razafimahefa, R. Vallee-Rai, L. Hendren, P. Lam, E. Cagnon, C. Godin: Practical Virtual Method Call Resolution for Java, 1999, *http://web.cs.ucla.edu/~palsberg/tba/papers/sundaresan-et-al-oopsla00.pdf*