**Dragan Milićev**

# The picoC++ Programming Language

***Instructor's Handbook***

**May 2018**

Language version: 1.0
Language date: May 2018
Document version: 1.0
Document date: May 2018

## Table of Contents

# Introduction

The *picoC++* (or *pC++* for short) is a simple object-oriented programming language designed for educational purposes. It represents a very simplified, tiny subset of standard C++, selected to meet the main design goals:

- simplicity of learning and implementation,
- suitability for teaching system software engineering disciplines, i.e., design and implementation of compilers, linkers, debuggers, and operating systems, and
- suitability for teaching basic procedural and object-oriented programming concepts for beginners, especially primary and high school students, using the flavor and concepts of C and C++, but avoiding the burden of the complexity of standard C/C++.

pC++ consists of only the very elementary procedural and object-oriented concepts, quite sufficient for effective programming of low-level software systems in educational environments, especially in academic courses on operating systems and other system software. Such systems include educational prototypes, tutorial implementations, and students' projects.

The syntax and semantics of pC++ are true subsets of those of standard C++: with a few minor exceptions, every regular program in pC++ is also a regular program in standard C++, with the same static and runtime semantics, meaning that it compiles and executes equally (after additional linking with a standard linker). (The opposite is, obviously, not true.) This allows for using standard programming tools, such as IDEs, compilers, linkers, and debuggers, for developing pC++ programs. On the other hand, developing a compiler and debugger for pC++ is dramatically simpler than for standard C++. This allows usage of pC++ for designing demonstrative programs and students' projects in teaching environments, such as simulators or virtual machines, providing much easier and more explicit and transparent insight into the background implementation of concepts and principles being taught.

## Motivation

I have been teaching operating systems at the University of Belgrade for 13 years now. The course is strongly oriented towards deep and thorough understanding of all covered concepts and their implementation, so that the students can become able to implement pieces of such kind of systems themselves. For the mandatory project in that course, the students have to develop their own small multithreaded, preemptive kernel with semaphores and a couple of other features.

Looking for a way to further improve the level of students' capabilities and achievements in understanding of complex concepts and features of modern computers and operating systems, I have concluded the following:

- Practice is crucial for understanding. Practical implementation of a certain concept, feature, or a piece of system, especially if it is done by the student him/herself, is much more effective than sole and pure theoretical explanation. Textbooks and courses that explain concepts of operating systems using concrete examples, such as implementations of educational kernels, are precious.

- The implementation is easier to understand if its dynamics, i.e., dynamic behavior and state of the system can be visualized and dynamically traced (for example, in a way a debugger helps understanding the behavior of a program).
- However, implementing even a small piece of an operating system's kernel so that it can run on today's hardware can be extremely complex. Contemporary processors, devices, and protocols are so complex that one has to devote at least a significant part of one's professional carrier in order to be familiar with all their details. For example, implementing just a bootstrap program for a contemporary IA-32/64 processor, or a device driver for a SATA disk drive is a huge task. Even explaining and understanding such a piece of software may be completely intractable and unsuitable for educational purposes.
- On the other hand, dealing with all these subtleties of modern hardware and software is even unnecessary. Core concepts are often much simpler and easier to understand in simpler contexts and environments. These concepts are usually rather old and well known in computing for a long time. The complexity of details of modern hardware, protocols, and software may completely blur or hide the main concepts and principles.
- Therefore, having a simple platform for executing software so that it does not impose unnecessary complexity may be invaluable. Unfortunately, a real hardware platform like that does not exist, or is at least not easily and widely available in normal educational environments.
- In the past, computers and software used to be much simpler and easier to understand and implement. In addition, contemporary hardware and software systems have emerged by an evolution of old, simpler systems. It is very often much easier to understand why a concept, mechanism, or a feature of hardware or operating system looks how it looks now by studying its evolution through history of development from root, simple systems.
- Therefore, having an opportunity to execute software on (a simulation of) old, simple systems (architectures, devices) seems also as very promising and beneficial for education.

As a result, I have conceived a Web-based, interactive educational platform for teaching and learning operating systems, which can also be useful for teaching other closely related subjects, such as computer architecture, compilers, and other system software (linkers and debuggers). The platform would provide the following features and services:

- simulation of hardware in an abstract, high-level architectural way (not on the register transfer level); the simulator would enable tracing of and breakpoints at all hardware events that are relevant to operating systems and understanding of their interaction with hardware, such as instruction execution, virtual or physical memory addressing, exceptions, interrupts, page faults, etc.
- processor, memory, and I/O device modeling and simulation would be simplified, enabling i.e. using a simplified RISC processor architecture or a block device with a very simple interface that is sufficient for focusing to and demonstrating core concepts and features of modern operating systems, abstracting away all the burden and complexity of irrelevant details;
- the platform would encourage modularization and encapsulation, so that specific and potentially overkilling details and parts of the architecture and devices would be abstracted

away and accessed through simplified interfaces, while their implementation can be implemented by a programmed simulator;

- the platform would also provide complete flexibility, allowing variations of those elements of the simulated computer architecture that are relevant for operating systems (e.g., interrupt handling and processor mode switching, operating memory organization, etc.);
- by the means of modularity and variability, the platform would enable development of a library of modules (simulators of processors, memory, devices), providing a case study of examples that illustrate gradual, evolutional development of certain concepts through history, by adding new desired features and solving problems one by one, according to the needs (e.g., from a simple single-process, batch operating system without virtual memory to multiprocessor systems with virtual memory).

Of course, one should be able to write software for that simulation platform in a standard, well known programming language, and the choice was easy to make – C/C++. All other popular languages are not suitable for this purpose, because they are too abstract and hide the details of implementation (e.g., memory layout of data) from the programmer. Some features of C that other languages do not have, such as arbitrary pointer casting without exceptions (e.g., from a raw memory pointer to a pointer to a structure or object, or vice versa), or pointers to functions (storing pure addresses, e.g. for interrupt vectors) are unique and necessary in this context. Although most operating systems are implemented in pure C, some object-oriented concepts of C++, especially the basic ones (abstract data types, encapsulation, inheritance, polymorphism), make the programming much easier and, more importantly, encourage proper modular and abstract thinking, which is also important for education.

Code written in C++ then has to be compiled to the language of the simulated processor (e.g. a simple RISC processor with load/store architecture). Such a compiler can be implemented by using the available open-source compilers that allow customization and retargeting. However, implementing a debugger for that platform, and a source code-level debugger is certainly a necessary tool for a comfortable environment, turns out to be an extremely tough task. All available open-source tools that I had studied turned out to be unsuitable for this need. Building a debugger for C++ from scratch, on the other hand, is too much complicated and would become an overkill: just interpreting the standard ELF format of executables, or interpreting the layout of all types of data of all memory storage categories would be a nightmare.

This is how I came up with the idea of designing pC++. pC++ would be constrained to only those concepts that are necessary for programming in the described context on one hand, and simple enough so that a compiler and a debugger can be implemented even through more advanced students' projects (e.g. for master theses). It seems to be the right solution.

I have also been teaching object-oriented programming and C++ since 1993 in both academy and industry. At that time, although already complex and difficult to master, C++ was still “small” enough to be covered fully by an ordinary textbook or academic course. In the meantime, standard C++, along with its standard library, has evolved into a monster of a language for which one has to devote a significant part of one's professional carrier, in order to master completely all its concepts, semantics, features, details, and peculiarities. This is obviously impossible, and also unreasonable for an ordinary academic course. On the other hand, I also faced the following problem when I started teaching my own kids the first steps of programming (once they mastered the level of programming with Scratch): we wanted to start with most basic concepts of procedural (and not yet OO) programming and go for C (instead of e.g. using Python, Java, or C#), and the problem with teaching beginners is again the burden of all complex details one cannot understand, but has to use from the very beginning. Take, for example, a simple console input/output in C (with *scanf/printf* and their complicated formats and the use of pointers) or in C++ (with *cin/cout* that can be taken abstractly, bat cannot be fully understood prior to describing objects, classes, and operator overloading).

Again, pC++ may be a solution to this problem too. It is a simple language for teaching basic procedural and object-oriented concepts, using the flavor of C and C++, but without having to cope with advanced features of those languages. These advanced concepts may be captured later more easily, once the basic features have been mastered.

## About This Book

This book is an informal description of pC++ for those who are interested in teaching it or implementing tools for it (compilers and debuggers). It assumes the reader is already familiar with C++ and knows its concepts and features (or at least those that are included in pC++).

Therefore, the book is not a tutorial for pC++ for absolute beginners: one has to be familiar with the basic procedural and object-oriented concepts in order to follow this book.

This book is not a complete reference or an authoritative specification of the language, either.

The text in this book describes the rules and design decisions of pC++. These are highlighted in the paragraphs styled like this:

*This is a language rule or design decision.*

The other text describes the rationale of the rules and design decisions, and provide examples or directions for practical use of the language.

As any product of human work, this book may contain errors. As of this writing, the pC++ language is only conceived, and not yet implemented, so errors or inconsistencies are even more likely. The reader should be aware of this. I would be grateful if you let me know about them.

This is also a very first draft of the language. I would be happy if it evolves, especially from its use and requests from its users and practice. I am open to questions and suggestions.

Belgrade, May 2018 *The author*

# Compilation

*The entire pC++ program is in a singe file, being a single compilation unit.*
*The compiled (single) file is immediately executable: there is no need for the linking phase.*

As opposed to standard C++, the entire program in pC++ is in one file. This is quite sufficient for small, simple programs, and especially suitable for teaching beginners: there is no need to explain the need for the linking phase, which might be too complicated, especially when it is not obvious (in case of small, single-module programs). In addition, this also simplifies the implementation of the described framework, as it removes the need for the linker (at least in a classical sense of resolving external references in object files).

In order to support modularity of bigger programs, pC++ provides a simple way for program composition at the source code level, by including the source code of modules (verbatim) into the (single) program file via the `#include` directive. Therefore, a program may look like follows:

```
#include "module1.cpp"  // Source code of module 1
#include "module2.cpp"  // Source code of module 2
#include "module3.cpp"  // Source code of module 3

int main () {  // Main program
  ....
}
```

Sometimes circular dependences exist between modules: the implementation (e.g., a function body) from one module may refer to an identifier declared in the other module, and vice versa. To resolve such cases, a usual style should be to divide the interface declarations from the implementation of a module into .h and .cpp files, respectively, and include all .h files first, and then the rest:

```
#include "module1.h"  // Interface of module 1
#include "module2.h"  // Interface of module 2
#include "module3.h"  // Interface of module 3

#include "module1.cpp"  // Implementation of module 1
#include "module2.cpp"  // Implementation of module 2
#include "module3.cpp"  // Implementation of module 3

int main () {  // Main program
  ....
}
```

Obviously, including the entire program's code and compiling it all at once may have negative impact on the compilation time for large programs. However, this drawback is not expected to be too problematic in practice, because:

- very often in the context where pC++ is expected to be used, programs will be small,
- larger programs will most likely be developed using standard IDEs, and compiled with a pC++ compiler infrequently, at the time of deployment to the simulation platform.

*There are no standard libraries.*

All necessary libraries will be provided in the framework or implemented ad hoc, according to the concrete context of use. As a consequence, there are no standard header files.

# Lexical Analysis and Preprocessing

*Comments, tokens (but only those covered by pC++), and white spaces are extracted as in standard C++.*

Many sequences of characters that are recognized by standard C++ as single tokens, are not recognized as legal tokens, but as illegal sequences of tokens in pC++. For example: `.*`, `->*`, `<=>` etc. are illegal sequences of tokens in pC++.

*Keywords and reserved words in standard C++ are reserved words in pC++.*

Many keywords that exist in standard C++ are not keywords in pC++, such as `explicit`, `const`, `new`, `delete`, or `void`, but are still reserved words, meaning that they cannot be used for identifiers in pC++, although they have no any meaning and are thus not allowed in pC++.

*Identifiers are formed as in standard C++, without Unicode characters.*

Identifiers in pC++ are case-sensitive, arbitrarily long sequences of digits, underscores, lowercase and uppercase Latin letters. Unlike in standard C++, identifiers in pC++ cannot contain Unicode characters. A valid identifier must begin with a non-digit character (Latin letter or underscore). Every character is significant. There are no reserved identifiers.

## Literals

*Literals are of one of the following kinds:*

- *integer literals are decimal, octal, hexadecimal, or binary numbers of an integer type supported by pC++;*
- *character literals are individual ASCII characters enclosed in single-quotes;*
- *string literals are sequences of ASCII characters enclosed in double-quotes.*

There are no floating-point, Boolean, or user-defined literals.

Integer literals are one of the following:

- decimal literal is a non-zero decimal digit (1, 2, 3, 4, 5, 6, 7, 8, 9), followed by zero or more decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9); for example:

```
321
1000
12399
```

- octal literal is the digit zero (0) followed by zero or more octal digits (0, 1, 2, 3, 4, 5, 6, 7); for example:

```
0321
01000
01237
```

- hexadecimal literal is the character sequence `0x` or the character sequence `0X` followed by one or more hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F); for example:

```
0xff
0x1A
0X1de
```

- binary literal is the character sequence `0b` or the character sequence `0B` followed by one or more binary digits (0, 1); for example:

```
0b10
0b0010
0B11111110
```

Each integer literal may be immediately followed by a suffix, which may contain one or both of the following (if both are provided, they may appear in any order):

- unsigned suffix (the character `u` or the character `U`);
- long suffix (the character `l` or the character `L`); for example:

```
321U
1000UL
12399L
```

The type of an integer literal is the first type in which the value can fit, unless the long suffix is present (then it is `long`), modified with the unsigned suffix, if present.

Character literals are one of the following, enclosed in single-quotes (`'`):

- a single-byte character from the base character set (i.e., ASCII), minus single-quote (`'`), backslash (`\`), or the newline character,
- escape sequence (a sequence beginning with a backslash), as in standard C++, except from Unicode. For example:

```
'3'
'\0'
'\n'
'\\'
'\0xff'
```

The type of a character literal is always `char`.

String literals are sequences of characters (the same as in character literals, i.e., optionally escape sequences), enclosed in double-quotes (`"`). For example:

```
"Hello world!"
"Hello\nworld!"
```

String literals are of type `char[]`. The null character (`'\0'`) is always appended at the end of the sequence of characters of a string literal. For example, the following string literal has six characters:

```
"Hello"
```

String literals have static storage duration, and thus exist in memory for the entire life of the program. The compiler is allowed, but not required, to combine (overlap) storage for equal or overlapping string literals. For example:

```
char* p = "Hello world!";
char* q = “world”;
if ((p+6) == q) ... // may and may not be true
```

Attempting to modify a string literal results in undefined behavior: string literals may be stored in read-only memory, where an attempt to modify it may cause an exception raised by the hardware and abnormal termination of the program, or combined (overlapped) with other string literals, where an attempt to modify it may also modify another string literal.

There is one minor discrepancy from the compatibility with standard C++11 and later. Namely, in C, just like in pC++, string literals are of type `char[]`, and can be assigned directly (without an explicit cast) to a (non-const) `char*`. C++03 allowed it as well (but deprecated it, as literals are `const` in C++). C++11 no longer allows such assignments without a cast. Since pC++ does not recognize `const`, this is a legal code in pC++ (as well as in C++03 and older), but illegal in C++11 and later:

```
char* str = "Hello world!”;
```

To avoid such exceptions, an explicit cast should be used:

```
char* str = (char*)"Hello world!”;
```

## Preprocessor

> *The preprocessing phases take place before compilation and transform the source text only, as in standard C+.*
> *There is only one preprocessor directive:* `#include “filepath”`

The `#include` directive can have the file name in double-quotes only (recall that there are no standard libraries and headers). The effect of this directive is the same as in standard C++: it is replaced by the preprocessor with the contents of the included file, with its own nested `#include` directives expanded recursively.

# Program Structure

*A program may consist of a sequence of:*

- *#include directives*
- *forward declarations of structs and classes*
- *definitions of structs and classes*
- *definitions of static global and member objects*
- *forward declarations of global functions*
- *definitions of global and member functions*

*and nothing but these.*

Forward declarations (which are not definitions) of structs and classes are of a simple form:

```
struct Time;
class Clock;
```

They are used to resolve circular dependencies between definitions of structs and classes, i.e., in cases when two or more definitions refer to each other:

```
class Part;

class Whole {
  ...
private:
  Part* myPart;  // Requires a declaration of Part
};

class Part {
  ...
private:
  Whole* myOwner;  // Requires a declaration of Whole
};
```

Definitions of structs and classes contain declarations of their members within curly braces `{}`. They will be explained in detail later.

Definitions of static objects may be definitions of global objects or of static class data members:

```
int numberOfProcesses; // Global static object

Process* Process::procListHead; // Static class member
```

Forward declarations (which are not definitions) of global functions contain their signatures only:

```
int max (int a, int b);
```

Similar to structs and classes, they are used to resolve circular dependencies between definitions of functions, i.e., in cases when two or more functions call each other.

Any program must contain one and only one definition of a function `main`. It is the starting point of program execution. The program terminates implicitly when the flow of control returns from this function.

# Types

There are the following types in pC++:

- Object types are:
  - Scalar types
  - Composite types
- Functions are:
  - global functions
  - member function

Scalar types are:

- Integral types
- Pointers

For the sake of simplicity of implementation, there are no floating point types in pC++. If necessary, they can be supported by libraries.

Composite types are:

- arrays
- structs
- classes

## Integral Types

> *Integral types are:*
> - `char`*: 8 bits*
> - `short`*: 16 bits*
> - `int`*: 32 bits*
>
> *Each of these types can have the unsigned version by using the* `unsigned` *modifier.*

Unlike in standard C++, integral types have a guaranteed size in bits. Except for `char`, which is limited to 8-bit values (Unicode is not supported in pC++), the size of other types is compatible with the minimal requirements of the standard. This simplifies programming and teaching, and makes programs portable. In addition, this reduces the calculation of data sizes to the unit of (8-bit) bytes, since `sizeof(char)==1`. For the context of usage of pC++, this is quite convenient, because there is a great need for clear and easy data alignment, without introducing the complexity of the alignment specifiers in standard C++.

There are no bigger integers (`long` and `long long`), because the envisioned educational platforms are most likely to be 32-bit. 64-bit platforms do not bring significant conceptual value (except from a huge, practically unlimited address space), but may add complexity that is inconvenient for teaching and learning.

The assumed representation of all integral types is signed (there is no explicit `signed` modifier). Each of these types can be modified with the `unsigned` modifier, which indicates that the values will have the unsigned representation. The unsigned type has the same size in bits as the base type.

Note that the keyword `short` is the full name of the type, and is not a modifier in pC++: `short int` is an irregular construct in pC++, although allowed in standard C++.

For example:

```
char c;  // 8 bits, signed
short s; // 16 bits, signed
int i;   // 32 bits, signed
unsigned char uc;  // 8 bits, unsigned
unsigned short us; // 16 bits, unsigned
unsigned int ui;   // 32 bits, unsigned
```

## Pointers

*Pointer types are:*

- *pointers to integral object types,*
- *pointers to structs and classes,*
- *pointers to functions (global or static member functions),*

*and nothing but these.*

For the sake of simplicity of parsing declarations, interpreting and managing types, and understanding pointers by beginners, pointer types are limited to the given types only.

There are no pointers to pointers. Pointers to pointers complicate declarations and understanding of code. In addition, other newer object-oriented languages use references instead of pointers and do not allow references to (bare) references, but only to objects. This also seems to be easily avoidable by the following style, which improves readability of code: when a pointer has to be pointed to by another pointer, the former has to be promoted to an object of a class or struct, i.e., wrapped into at list a simple struct. For example:

```
struct InterruptVector {
  int (*vector)();  // Pointer to a function
};

InterruptVector* ivtEntry; // A pointer to a struct
...
ivtEntry->vector = &f;
```

Pointers to functions are still supported by pC++, because they are needed for low-level implementation of some operating system features, such as interrupt vectors. In addition, in order to demonstrate the means of implementing polymorphism through dynamic binding and virtual function tables in popular operating systems written in C, one has to use pointers to functions.

There are no pointers to arrays, because a pointer to an array can be easily replaced with a pointer to the first element of the array (an implicit conversion). Pointers to arrays, arrays, and pointers to elements of arrays introduce unnecessary confusion and are usually replaceable. PC++ supports only pointers to objects that are elements of arrays.

There are no pointers of type `void*`.

> *A pointer may have a null value*, meaning that it does not point to any object. The literal 0 is used as the literal for the null pointer value.

There are no other literals that represent a null value of a pointer in pC++.

As in standard C++, a pointer may have one of the following values:

- a value that points to an object or function, or
- a value that points past the end of an object, or
- the null pointer value, or
- an invalid pointer value.

A pointer that points to an object represents the address of the first byte in memory occupied by the object. A pointer past the end of an object represents the address of the first byte in memory after the end of the storage occupied by the object. Note that two pointers that represent the same memory address may nonetheless have different values, as in e.g. segment memory organizations with overlapping segments.

Any use of an invalid pointer value has either undefined or implementation-defined behavior.

## Arrays

> *Array types are:*
> - *arrays of integral object types,*
> - *arrays of structs and classes,*
>
> *and nothing but these.*

Due to the same reasons as for pointers, arrays can hold only objects of integral types, structs, or classes. There are no arrays of arrays or arrays of pointers. If such a thing is required, the elements of the array must be wrapped into classes or structs:

```
struct InterruptVector {
  int (*vector)();  // Pointer to a function
};

InterruptVector interruptVectorTable[N]; // An array of structs
...
ivtEntry[i].vector = &f;

struct Row {
  int row[N];
};

Row matrix[M];
...
matrix[i].row[j] = 5;
```

An array is a package of its elements, an ordered collection of contiguously allocated objects, with the semantics as in standard C++. The elements of an array with *n* elements are indexed with indexes from 0 to *n*-1. The elements can be accessed through the subscript operator `[]`. There is a built-in implicit conversion form an array to the pointer to its first element:

```
int a[N];
int* p;
p = a;  // p points to a[0]
```

> *The dimension of the array has to be specified by a constant expression in each array declaration.*

Since arrays in pC++ are always single-dimensional, and cannot be used as incompletely defined types as in standard C++ (e.g., as types of function arguments or with initializers), their dimension has to be specified by a constant expression having an integer result (e.g., an integer literal).

## Structs

> *Structs may only have data members of any object type.*
> *Structs have all members public by default; access specifiers are not allowed.*

Structs in pC++ can have data members only. Data members of structs can be static or non-static and can be of any object type.

Structs cannot have member functions. Consequently, there are no constructors or destructors in structs. There are no derivation for structs (no base struct).

All data members of structs are public by default. Access specifiers (`public`, `protected`, `private`) are not allowed within struct definitions.

As it can be seen, except that they can have data members of class types, structs in pC++ are simple data structures as they used to be in C, i.e., they are “plain old data” types (POD-types) unless they contain a data member of a class with member functions.

```
struct ListElement {
  ListElement* next;
  char* contents;
};
```

## Classes

> *Classes may have data members, static or non-static, of any object type.*

Classes in pC++ may have data members of any object type. A data member can be static or non-static. Once declared, a static data member must be defined somewhere in the program, outside the definition of the class. For example:

```
class Process {
public:
  ...
private:
  static int numOfProcesses; // Static data member
  int pid; // Non-static data member
  ...
};

int Process::numOfProcesses; // Definition of the static member
```

*Classes may have static and non-static member functions.*
*All non-static member functions are always virtual; the keyword `virtual` is mandatory.*

Classes may have static and non-static member functions. Once declared, each member function must be defined somewhere in the program, outside the definition of the class:

```
class Process {
public:
  static int getNumOfProcesses ();
  ...
private:
  static int numOfProcesses; // Static data member
  int pid; // Non-static data member
  ...
};

int Process::numOfProcesses; // Definition of the static data member

int Process::getNumOfProcesses () { // Definition of the static member fun
  return numOfProcesses;
}
```

All non-static member functions are always virtual. For the sake of compatibility with standard C++ and improved clarity of code, the keyword `virtual` is mandatory in every declaration of a non-static member function. For example:

```
class IOStream {
public:
  virtual int getc (char*);
  virtual int putc (char);
  ...
};
```

There are no constructors and destructors in pC++; all member functions are equal in their role. Therefore, there are no implicit function invocations upon object initialization or destruction. If necessary, such activities have to be supported by (ordinary) member functions and their explicit calls:

```
class Process {
public:
  int init (int parentPID, int uid);
  ...
};

...
processes[i].init(pid,uid);
```

*Access specifiers `public`, `protected`, and `private` declare sections of member declarations, and can appear in that order only; every section is optional.*

In order to simplify the grammar and the compiler, and to improve clarity and uniformity of coding style, each of the access specifiers `public`, `protected`, and `private` always starts the entire section of member declarations of the corresponding access level. Each of these specifiers can occur only once in a definition of a class. Each section is optional, but the sections must occur in the given order (public, protected, private) only. A member declaration has to belong to a section, meaning that a member declaration cannot occur before any access specifier. For example, the following class definitions are correct:

```
class User {
public:
  ... // some member declarations
protected:
  ... // some member declarations
private:
  ... // some member declarations
};


class UserGroup {
public:
  ... // some member declarations
private:
  ... // some member declarations
};

class IOStream {
public:
  ... // some member declarations
};
```

while these are incorrect:

```
class User {
public:
  ... // some member declarations
protected:
  ... // some member declarations
public:
  ... // some member declarations
};


class UserGroup {
private:
  ... // some member declarations
public:
  ... // some member declarations
};

class IOStream {
  ... // some member declarations
public:
  ... // some member declarations
};
```

*A class can be (publicly) derived from only one class.*

A class can have only one base class – pC++ supports single inheritance only. The base class is always public, but for the sake of compatibility with standard C++, the keyword `public` is mandatory:

```
class FStream : public IOStream {
  ...
};
```

In practice, if a class has to inherit several base classes, or implement several interfaces, a traditional and explicit approach with embedding objects of base classes (which is, in fact, how multiple inheritance is implemented anyway) has to be applied:

```
class Derived {
public:
  Base1* asBase1 ();
  Base2* asBase2 ();
private:
  Base1 b1;
  Base2 b2;
};

Base1* Derived::asBase1 () {
  return &(this->b1);
}

Base2* Derived::asBase2 () {
  return &(this->b2);
}
```

Apart from declarations of data members and member functions, a class definition cannot hold any other declaration. In particular, there are no nested classes or other types with class scope.

## Functions

Depending on their declaration, functions can be global (i.e., having global scope) or class members (i.e., having class scope).

*The return type of a function is mandatory and can be one of scalar types only, except pointers to functions.*
*Arguments of functions can be of scalar types only and must all be declared.*

The return type of each function is mandatory and can be of integral type or a pointer to object type. Declarations of functions returning pointers to functions are considered confusing and are thus avoided. There is no `void` in pC++. Arguments of functions can also be of scalar types only and all arguments must be declared. Consequently, a function cannot accept or return an object of a class or struct; only pointers to such objects can be passed into and out from a function:

```
int List::add (ListElem* newElem) {
  newElem->next = 0;
  if (this->tail) this->tail->next = newElem;
  else this->head = this->tail = newElem;
```

```
    return 0;  // Has to return something, at least an int
  }
```

Parameters of functions cannot have default values.

*Each function must have a unique name within its scope.*

There is no function overloading in pC++. Functions are bound to call operations via their identifiers only. Thus, a function has to have a unique name in its scope (class or entire program).

If a derived class declares a non-static member function having the same name as an inherited function (i.e., a function declared in the base class, or its base classes recursively), the entire signature of the function has to be exactly the same, and the new declaration overrides the inherited function (polymorphism). In all other cases, the declaration is incorrect and the program is ill formed. For example:

```
  class Base {
  public:
    virtual int f1 (X*, int);
    static  char f2 (Y*, char);
  };

  class Derived : public Base {
  public:
    virtual int f1 (X*, int);  // Correct. Overrides f1.
    virtual int f1 (X*, char); // Incorrect. Different signatures.
    static  char f2 (Y*, char);// Incorrect. Redefinition of a static member.
  };
```

A forward function declaration is a declaration of a function without its body. Such declarations are allowed to reoccur in the same scope, if they have completely equal signatures. They are used to resolve circular dependencies between functions, when two or more functions call each other in a circular manner.

As in standard C++, pointers to functions are of the same type if and only if the pointed function types have exactly the same signatures:

```
  int f1 (X*, int);
  char f2 (X*, int);
  int f3 (X*);

  int (*pf) (X*, int);
  pf = &f1; // Correct. Pointed function types are equal.
  pf = &f2; // Incorrect. Different pointed function types.
  pf = &f3; // Incorrect. Different pointed function types.
```

*Each non-static member function has an implicitly declared pointer `this`. An access to a non-static member requires the specification of the target object; `this` cannot be omitted.*

Each non-static member function of a class `X` has an implicitly declared pointer `this` of type `X*` of a local scope in that member function. At the time of the function invocation, that pointer is bound to the object whose member function is called (the target object). Each access to a non-static member of a class has to be done over the operator `.` or `->`, i.e., it requires the specification of the object whose member is accessed (the target object). Unlike in standard C++, `this` cannot be omitted and is not assumed, but has to be explicitly stated. This improves readability of the code and resembles some other newer languages, like JavaScript. For example:

```
class Timer {
public:
  virtual int elapsed ();
  static  Timer* clock ();
private:
  int time;
  static Timer* clk;
  ...
};

int Timer::elapsed () {
  return this->time; // this-> is mandatory.
  // 'return time' would be incorrect.
}

Timer* Timer::clock () {
  return clk; // Correct, because clk is static.
  // 'this->clk' and 'Timer::clk' would also be correct.
}
```

Access to a static member does not require the specification of the target object, although it is allowed.

Static member functions do not have the pointer `this`.

# Declarations and Scopes

*A declaration introduces one and only one name (identifier) into a scope.*
*A declaration cannot have initializers.*

A declaration introduces a *name* (in pC++, a name is always an identifier) into a *scope*; a scope is a possibly discontiguous portion of the source code where the name is valid. A name cannot be used before it has been declared.

Declarations in standard C and C++ have very complex syntax and may declare several names. However, the standard C and C++ syntax for declarations of pointers, due to the binding of the token `*` to the declared identifier and not to the identifier of the pointed type, reduces readability and may cause confusion to beginners in interpreting declarations containing several names, especially when some are and some are not pointers. The idea of pC++ is to avoid that confusion and simplify parsing and human interpretation of pointer declarations, but to preserve compatibility with the standard. For that reason, pC++ takes a much simplified approach: a declaration may introduce one and only one name. Together with simplified types allowed in pC++ (e.g., no pointers to pointers), this makes the syntax and parsing of declarations very simple:

```
int i; // Object of type int
int* pi; // Pointer to an object of type int
int* (*pf)(int); // Pointer to a function that accepts int and returns int*
int* f (int); // Function that accepts int and returns int*
int* a[100]; // Array of pointers to int
```

The restriction to a single name in a declaration makes coding a bit more tedious, indeed, because several names of the same simple type require as many declarations, but this is traded off for the simplicity:

```
int i, j, k; // This is incorrect
int i; int j; int k; // Each name requires a separate declaration
```

Again for the sake of simplicity and avoidance of implicit behavior, pC++ does not allow initializers in declarations. (Recall that there are no constructors in pC++ either.) Instead:

```
int i = 0; // Incorrect
Time t(12, 44, 00); // Incorrect
```

one has to make every initialization explicit through assignment or function call:

```
int i;
// and then somewhere in a function body:
i = 0;
Time t;
// and then somewhere in a function body:
t.init(12, 44, 00);
```

*There are four kinds of scopes in pC++, which allow declarations of the following names:*

- *Global: functions, objects of all types, structs, and classes;*
- *Block: objects of scalar and pointer types; arguments have local scope;*

- *Struct: data members;*
- *Class: data members and member functions.*

The *global* (or file) scope have names whose declarations are placed outside all definitions of structs and classes and outside all function bodies. A global name is visible from its declarations to the end of the program. Global can be:

- functions (which are not members of classes);
- objects of any object type;
- structs;
- classes.

The *block* (or local) scope have names declared in a compound statement (block). The scope of a local name begins at the point of declaration and ends at the end of the block enclosing the declaration. For the sake of simplicity of debugger implementation (i.e., for interpreting automatic objects placed on the stack), this scope can have only objects of:

- scalar types;
- pointer type.

Arguments of functions also have local scope. For example:

```
int x; // A global x
int y; // A global y

int f (int x) {  // A local x
  int y = 3;  // A local y
  return x+y;  // Refers to the local x and y
} // Scope of local x and y ends

int main () {
  x = 0;  // Refers to the global x
  f(1); // Returns 4
}
```

Structs and classes also represent scopes. The scope of a name declared in a struct or class begins at the point of declaration and includes the rest of the class body and all bodies of its member functions (although they are defined outside the class definition).  A name of a struct or class member can only be used:

- in its own class scope or in the scope of a class derived from its class;
- after the `.` operator applied to an expression of the type of its struct/class or a class derived from it;
- after the `->` operator applied to an expression of the type of pointer to its struct/class or pointers to a class derived from it;
- after the `::` token applied to the name of its struct/class or the name of a class derived from it.

When a name is encountered in a program by the compiler, the *name lookup* procedure tries to find and associate a declaration that introduced it. The name lookup procedure is the same as in standard C++, limited to the features supported by pC++. It can be:

- *qualified name lookup*: when a name is qualified with a class name and the token `::`, the lookup starts from the scope of that class and then examines the scope of its base class, recursively; a call to a qualified member function is never polymorphic:

```
class Base {
public:
  virtual int f ();
};

class Derived : pubic Base {
public:
  virtual int f (); // Overrides B::f
};

Derived d;
Base* p = &d;

int main () {
  p->f(); // Polymorphic
  p->B::f(); // Not polymorphic, Base::f is called
}
```

- *unqualified name lookup*: unqualified name is a name that does not appear to the right of a token `::`; unqualified name lookup examines the scopes until it finds at least one declaration of any kind, at which time the lookup stops and no further scopes are examined; names declared in nested blocks hide the names declared in the enclosing blocks and global names; for member function bodies, the entire scope of the owner class and its base class (recursively) is also examined; since there are no namespaces and nested types in pC++, there is no recursive search for a name in any other class.

# Objects

*Objects can have only one of these storage categories:*

- *static storage have global objects or static data members;*
- *automatic storage have local objects (including arguments).*

*Built-in operators, including function calls, return pure values of scalar types only and have temporary lifetime.*
*Objects embedded in other objects (data members, base class subobjects, and elements of arrays) have their storage and lifetime bound to the enclosing object/array, which is ultimately always static.*

Objects are instances of object types: integral types, pointers, arrays, structs, and classes. A *storage category* of an object indicates how and when the space for the object in memory is allocated.

In order to simplify the implementation of the compiler and debugger, one of the most restrictive design decisions underpinning pC++ is that there are only two *storage categories* for objects:

- static and
- automatic.

Static storage have all globally defined objects and static data members. They can be of any object type, including composite types – structs, classes, and arrays. For each definition of a static objects, there is exactly one object associated at runtime. Consequently, the space for static objects is allocated at compilation time, in the product of compilation (i.e., in the executable file defining the program's initial memory layout).

Automatic storage have local objects (including arguments). An object with automatic storage is allocated each time the control flow reaches the beginning of the enclosing block and deallocated when the control flow exits the block. Since all such objects are defined within function bodies and there is no initialization, the allocation may actually take place at the invocation of the function and deallocation at the return from the function, which is a usual implementation (using a stack frame for all local objects of a function on the call stack).

As already said, local automatic objects can be of scalar types only, meaning that objects of structs and classes cannot have automatic storage and lifetime. This design decision simplifies the implementation of the debugger (i.e., the interpretation of the object structure on the stack). Moreover, this makes pC++ more similar to most other popular OO programming languages, where objects of classes have always dynamic storage allocation, i.e., cannot have automatic storage and cannot be used by value, but are always accessed over references (pointers). For that reason, objects of structs and classes in pC++ cannot be used and passed by value, as in standard C++, which is one of the key roots of the huge part of the tremendous complexity of standard C++. Consequently, there is no copy or move semantics for objects of classes, including passing of arguments and returning objects by value (scalar types have trivial copy semantics); only pointers to objects of structs and classes can be passed as arguments to or returned from functions.

As all built-in operators and functions return objects of scalar types only (integral types or pointers), there are no temporary objects of structs and classes in pC++. Again, there is no need for copy or move semantics for objects of classes.

There are no dynamic objects built in pC++ either. The semantics of dynamic objects can still be provided with the support of ad-hoc class implementation or a library that dynamically allocates and deallocates objects from a preallocated piece of memory or a (static) array of slots. This approach makes the implementation of dynamic object lifetime more explicit, which is suitable for educational purposes, while still allowing convenient use if properly abstracted behind a class interface or library. For example:

```
class Object {
public:
  static int startup ();  // To be called on program's startup
  static Object* create (int a, int b, int c); // Some arguments
  virtual int destroy ();  // May also have arguments
  ...
protected:
  virtual int constructor  (int a, int b, int c);  // Some arguments
  virtual int destructor ();   // May also have arguments
  ...
private:
  Object* next;
  static Object* freeSlotsHead;
  static Object storage[];
  ...
};

Object* Object::freeSlotsHead;
Object Object::storage[10000];

int Object::startup () {
  freeSlotsHead = 0;
  int i;
  for (i=0; i<10000; i++) {
    storage[i].next =  freeSlotsHead;
    freeSlotsHead = &storage[i];
  }
  return 0;
}

Object* Object::create (int a, int b, int c) {
  if (freeSlotsHead) {
    Object* obj =  freeSlotsHead;
    freeSlotsHead = freeSlotsHead->next;
    obj->constructor(a,b,c);
    return obj;
  } else
    return 0;
}

int Object::destroy () {
  this->destructor();
  this->next =  freeSlotsHead;
  freeSlotsHead = this;
  return 0;
}

int Object::constructor (int a, int b, int c) {
  // Perform initialization of *this using the arguments
}
```

```
int Object::destructor () {
  // Perform clean-up of *this
}

int main () {
  ...
  Object::startup();
  ...
}
```

Now, objects of this class can be used as dynamic objects only, as it is usual in some other languages, but the implementation of dynamic allocation (and its memory layout) is completely under control:

```
Object* p = Object::create(a,b,c);
...
p->destroy();
```

After all, the dynamic storage (freestore, heap) of a program is ultimately allocated either statically or using an operating system call for dynamic memory allocation, so it can also be used here to provide memory for allocation of objects of classes by casting pointers to raw memory (`char*`) to pointers to classes. Therefore, static objects of structs and classes are a sufficient language support in pC++.

The lifetime and storage kind of objects embedded into other objects (elements of arrays, data members or base class subobjects) is always bound to the lifetime of the enclosing object. Eventually, since (independent, i.e., not embedded) objects of compound types are always static, embedded objects also always have static storage.

As a result, the lifetimes of objects of different types are:

- objects of scalar types (integral and pointer types) can have any lifetime, static, automatic, temporary, or embedded (within an array or an object of a class or struct);
- objects of composite types (arrays, structs, and classes) may have static storage and lifetime only.

As there are no initializers, all objects have to be explicitly set to the required initial state either by assignment operators (for scalar objects and objects of structs) or by calling their member function(s) (for objects of classes) from a statement in a function body. The only exception is the virtual table pointer (VTP) in objects of classes: the implementer of a compiler has to take care that the VTPs of static objects and embedded objects of classes (recursively) are properly initialized at compilation time, so that all objects of classes having at least one non-static member function are always polymorphic. The support for the initialization of VTPs dynamically, e.g., when converting raw memory to an object of class, is not defined yet.

# Conversions

*There are only built-in conversions; there are no user-defined conversions.*
*Explicit conversions (cast operator) are allowed only if defined in the language:*

- *all implicit conversions;*
- *between all scalar types.*

*Implicit conversions:*

- *integral promotions (`char` and `short` to `int`);*
- *pointer to derived class to pointer to a base class (including transitivity);*
- *array to the pointer to its (first) element;*
- *function to the pointer to function.*

pC++ supports only simple built-in conversions, there are no user-defined conversions.

Conversions can be performed implicitly or explicitly.

Explicit conversions are performed using the C-like cast operator `(new_type)expression`. They can be performed from type T1 to type T2 when:

- there is a sequence of allowed implicit conversions from T1 to T2;
- when T1 and T2 are any of the scalar types (integral types and pointers).

Implicit conversions are performed whenever an expression of some type T1 is used in a context that does not accept that type, but accepts some other type T2:

- when the expression is used as the actual argument when calling a function that is declared with T2 as the corresponding formal parameter;
- when the expression is used as an operand with an operator that expects T2;
- when the expression is used in a `return` statement in a function whose return type is T2;
- when the expression is used in an `if` statement or a loop (T2 is int).

The program is well-formed (compiles) only if there exists one unambiguous *implicit conversion sequence* from T1 to T2 of one or more of the following built-in conversions:

- integral promotions: `char` to `int` and `short` to `int` (and their unsigned counterparts);
- pointer to a derived class to the pointer to its base class;
- array of elements of type `T` to the pointer of type `T*`, which points to the first element of that array;
- a global function or static member function to a pointer to function, which points to that very function.

The semantics of these conversions is quite simple:

- integral promotion: when a value of an unsigned integral type is promoted to a larger type, it is zero-extended and preserves the value; when a value of a signed integral type is promoted, it may change the value if promoted to an unsigned type;
- integral demotion: when a value of a larger integral type is converted to a smaller integral type, it is simply truncated and may change the value;
- all other conversions do not compile to any instructions and thus have no runtime effects – the binary value is preserved.

# Operators and Expressions

*The number of operands, priority, and associativity of the operators supported in pC++ are the same as in standard C++.*

pC++ supports a reduced set of operators from standard C++, with simplified rules and semantics. The number of operands, priority, and associativity for these operands is the same as in standard C++.

*The order of expression evaluation is the same as in standard C++, including short-circuit evaluation of logical operators `&&` and `||`.*

Expressions can be enclosed with parentheses as subexpressions of larger expressions. As usual, parentheses may change the default order of evaluation. Logical operators `&&` and `||` are short-circuiting (if the first operand determines the result of the operation, the second operand is not evaluated). Taking into account the same priority, associativity, and default order evaluation as in standard C++, the order of expression evaluation in general is the same as in standard C++, including the unspecified order of evaluation (e.g., the order of evaluation of operands of most operators is unspecified).

*Lvalue property is defined and checked in a simplified manner, as in older versions of C++.*

Because there are no references and no temporary objects and objects passed by value, there is no move semantics in pC++. Consequently, the notion of *rvalues* and *prvalues* from C++11 (and later) are not needed in pC++. Instead, there is a very simplified notion of *lvalue*, as it used to be in C and in older versions of C++ (prior to C++98):

- a name of an object or a function is an lvalue;
- a result of an operator is or is not an lvalue, depending on the concrete operator and its operands.

Arrays and functions are not *modifiable* lvalues, and cannot be left-hand operands of assignment operators and increment/decrement operators. As there are no constants, all other lvalues are modifiable.

Operators return lvalues as in standard C++.

*There is no operator overloading in pC++.*

There is no operator overloading in pC++. Hence, except for some built-in operators, operators cannot work with objects of classes and structs, and cannot be (re)defined for classes and structs.

The set of supported operators, grouped according to their priority, is as follows:

```
::
( )<function call>      [ ]   ->    .     x++   x--
```

```
!     ~     +<unary>    -<unary>    ++x   --x   *<dereference>

&<address of>     sizeof      (type)<cast>

*     /     %

+     -

<<    >>

<     <=    >     >=

==    !=

&

^

|

&&

||

?:

=  +=  -=  *=  /=  %= >>=  <<=  &=  ^=  |=

,
```

In the text that follows, some additional comments are given for specific operators. For all other supported operators, the standard rules apply, limited to the types of operands supported by pC++.

There is only binary operator `::` , there is no its unary version. Since there are no nested namespaces and types, this operator is not associative, i.e., `x::y::z` is an illegal construct; this operator can only have a single use: `class_name::member_name`.

Arithmetic operators perform integral promotions of their operands. So, they always operate on (`unsigned`) `int`. Pointer arithmetic is performed as in standard C++.

*Assignment and equality operators are allowed for scalar object types only.*

Assignment and equality operators are allowed for scalar object types only. Objects of structs and classes cannot be passed by value, and cannot be assigned or copied otherwise. If such a thing is necessary, an explicit function that copies their members, one by one, has to be explicitly defined and called. This supports the idea of objects being treated as separate identities, as in most other popular languages.

# Statements

*There are the following statements in pC++:*

- *Composite statement (block)*
- *Local (automatic) object definition*
- *Expression*
- *`if`*
- *`for`, `while`, `do`*
- *`break`, `continue`*
- *`return`*
- *`asm`*

pC++ supports a reduced and simplified set of statements from standard C++. Their semantics is the same as in standard C++.

A block of statements enclosed in `{}` is a composite statement.

Definitions of local automatic objects are the only declarations that can appear within function bodies; syntactically, they are also statements.

Expressions terminated with `;` can also appear as statements.

The statements `if`, `for`, `while`, and `do` can only have a simple, C-like form:

```
if (expression) statement
if (expression) statement else statement
for (expression; expression; expression) statement
while (expression) statement
do statement while expression;
```

The expressions used in them must have an integral type or a pointer type. Note that the `for` statement does not allow a declaration as the initial statement. So, this is not allowed:

```
for (int i=0; i<N; i++) ...
```

Instead, one must write:

```
int i;
for (i=0; i<N; i++) ...
```

The `break` and `continue` statements are allowed only within loops (there is no `switch` statement).

The `return` statement must have an expression of an integral type or a pointer to object (not a pointer to function), because each function returns a (scalar) value.

The `asm` statement is a block of assembly statements (instructions and directives), translated verbatim by the compiler. These assembly statements can reference identifiers of accessible static objects (of global and class scope) and functions (global and static members) by using their identifiers.

# Appendix A: The pC++ Grammar

## *Translation Unit*

*translation-unit*:
    *global-declaration-seq*$_{opt}$
*global-declaration-seq*:
    *global-declaration*
    *global-declaration-seq  global-declaration*
*global-declaration*:
    *class-declaration*
    *struct-declaration*
    *global-function-declaration*
    *member-function-definition*
    *global-object-definition*
    *static-member-object-definition*

## *Classes*

*class-declaration*:
    **class** *class-name*  **;**
    *class-definition*
*class-name:*
    *identifier*
*class-definition*:
    **class**  *class-name  base-class-spec*$_{opt}$  **{**  *class-def-body*  **}**  **;**
*base-class-spec*:
    **:**  **public**  *class-name*
*class-def-body*:
    *public-member-decl-seq*$_{opt}$  *protected-member-decl-seq*$_{opt}$  *private-member-decl-seq*$_{opt}$
*public-member-decl-seq*:
    **public**  **:**  *member-decl-seq*
*protected-member-decl-seq:*
    **protected**  **:**  *member-decl-seq*
*private-member-decl-seq:*
    **private**  **:**  *member-decl-seq*
*member-decl-seq*:
    *member-declaration*
    *member-decl-seq  member-declaration*
*member-declaration:*
    *data-member-declaration*
    *member-function-declaration*
*data-member-declaration:*
    **static**$_{opt}$  *object-definition*
*member-function-declaration:*
    **static**  *nonmember-function-decl*
    **virtual**  *nonmember-function-decl*

## Structs

*struct-declaration*:

  **struct** *class-name* **;**
  *struct-definition*

*struct-definition*:

  **struct** *class-name* **{** *struct-member-decl-seq* **}** **;**

*struct-member-decl-seq:*

  *object-definition*
  *struct-member-decl-seq object-definition*

## Functions

*global-function-declaration:*

  *nonmember-function-decl* **;**
  *nonmember-function-decl compound-statement*

*member-function-definition:*

  *member-function-decl compound-statement*

*nonmember-function-decl:*

  *return-type identifier* **(** *parameter-decl-seq* **)**

*member-function-decl:*

  *return-type class-name* **::** *identifier* **(** *parameter-decl-seq* **)**

*parameter-decl-seq:*

  *parameter-declaration*
  *parameter-decl-seq* **,** *parameter-declaration*

## Object Declarations

*object-definition:*

  *primitive-object-definition*
  *object-of-class-definition*
  *ptr-to-object-definition*
  *array-definition*
  *ptr-to-function-definition*

*local-object-definition:*

  *primitive-object-definition*
  *ptr-to-object-definition*
  *ptr-to-function-definition*

*global-object-definition:*

  *object-definition*

*static-member-object-definition:*

  *primitive-object-member-definition*
  *object-of-class-member-definition*
  *ptr-to-object-member-definition*
  *array-member-definition*
  *ptr-to-function-member-definition*

*parameter-declaration:*

  *primitive-object-declaration*
  *ptr-to-object-declaration*
  *ptr-to-function-declaration*

*primitive-object-definition:*

  *primitive-type identifier* **;**

*primitive-object-member-definition:*

  *primitive-type class-name* **::** *identifier* **;**

*primitive-object-declaration:*
  *primitive-type identifier* $_{opt}$
*primitive-type:*
  **char**
  **unsigned char**
  **short**
  **unsigned short**
  **int**
  **unsigned int**
*object-of-class-definition:*
  *class-name identifier* **;**
*object-of-class-member-definition:*
  *class-name class-name* **::** *identifier* **;**
*ptr-to-object-definition:*
  *primitive-type* ***** *identifier* **;**
  *class-name* ***** *identifier* **;**
*ptr-to-object-member-definition:*
  *primitive-type* ***** *class-name* **::** *identifier* **;**
  *class-name* ***** *class-name* **::** *identifier* **;**
*ptr-to-object-declaration:*
  *primitive-type* ***** *identifier*$_{opt}$
  *class-name* ***** *identifier*$_{opt}$
*array-definition:*
  *object-type identifier* **[** *constant-expression* **] ;**
*array-member-definition:*
  *object-type class-name* **::** *identifier* **[** *constant-expression* **] ;**
*object-type:*
  *primitive-type*
  *class-name*
*ptr-to-function-definition:*
  *return-type* **( *** *identifier* **) (** *parameter-decl-seq* **) ;**
*ptr-to-function-member-definition:*
  *return-type* **( *** *class-name* **::** *identifier* **) (** *parameter-decl-seq* **) ;**
*ptr-to-function-declaration:*
  *return-type* **( *** *identifier*$_{opt}$ **) (** *parameter-decl-seq* **)**
*return-type:*
  *primitive-type*
  *object-type* *****

## Statements

*compound-statement:*
  **{** *statement-seq*$_{opt}$ **}**
*statement-seq:*
  *statement*
  *statement-seq statement*
*statement:*
  *declaration-statement*
  *expression-statement*
  *compound-statement*
  *selection-statement*
  *iteration-statement*
  *jump-statement*

*declaration-statement:*
    *local-object-definition*
*expression-statement:*
    *expression*$_{\text{opt}}$ **;**
*selection-statement:*
    **if** **(** *condition* **)** *statement*
    **if** **(** *condition* **)** *statement* **else** *statement*
*condition:*
    *expression*
*iteration-statement:*
    **while** **(** *condition* **)** *statement*
    **do** *statement* **while** **(** *expression* **)** **;**
    **for** **(** *for-init-statement* *condition*$_{\text{opt}}$ **;** *expression*$_{\text{opt}}$ **)** *statement*
*for-init-statement:*
    *expression-statement*
*jump-statement:*
    **break** **;**
    **continue** **;**
    **return** *expression* **;**

## ***Expressions***

*expression:*
        *assignment-expression*
        *expression* **,** *assignment-expression*
*constant-expression:*
        *conditional-expression*
*assignment-expression:*
        *conditional-expression*
        *logical-or-expression* *assignment-operator* *assignment-expression*
*assignment-operator:*
        =
        *****=
        /=
        **%**=
        +=
        **-**=
        >>=
        <<=
        **&**=
        ^=
        |=
*conditional-expression:*
        *logical-or-expression*
        *logical-or-expression* **?** *expression* **:** *assignment-expression*
*logical-or-expression:*
        *logical-and-expression*
        *logical-or-expression* **||** *logical-and-expression*
*logical-and-expression:*
        *inclusive-or-expression*
        *logical-and-expression* **&&** *inclusive-or-expression*

*inclusive-or-expression:*
*exclusive-or-expression*
*inclusive-or-expression* **|** *exclusive-or-expression*
*exclusive-or-expression:*
*and-expression*
*exclusive-or-expression* ^ *and-expression*
*and-expression:*
*equality-expression*
*and-expression* **&** *equality-expression*
*equality-expression:*
*relational-expression*
*equality-expression* == *relational-expression*
*equality-expression* **!=** *relational-expression*
*relational-expression:*
*shift-expression*
*relational-expression* < *shift-expression*
*relational-expression* > *shift-expression*
*relational-expression* <= *shift-expression*
*relational-expression* >= *shift-expression*
*shift-expression:*
*additive-expression*
*shift-expression* << *additive-expression*
*shift-expression* >> *additive-expression*
*additive-expression:*
*multiplicative-expression*
*additive-expression* + *multiplicative-expression*
*additive-expression* **-** *multiplicative-expression*
*multiplicative-expression:*
*multiplicative-expression* * *cast-expression*
*multiplicative-expression* / *cast-expression*
*multiplicative-expression* **%** *cast-expression*
*cast-expression:*
*unary-expression*
**(** *type-id* **)** *cast-expression*
*type-id:*
*primitive-type*
*primitive-type* *
*class-name* *
*return-type* **(** * *identifier*$_{opt}$ **)** **(** *parameter-decl-seq* **)**
*unary-expression:*
*postfix-expression*
++ *cast-expression*
**--** *cast-expression*
*unary-operator* *cast-expression*
**sizeof** *unary-expression*
**sizeof** **(** *type-id* **)**
*unary-operator:*
*
**&**
+
**-**

**!**
**~**

*postfix-expression:*
*primary-expression*
*postfix-expression* **[** *expression* **]**
*postfix-expression* **(** *expression-list*$_{\text{opt}}$ **)**
*postfix-expression* **.** *id-expression*
*postfix-expression* **->** *id-expression*
*postfix-expression* **++**
*postfix-expression* **--**

*expression-list:*
*assignment-expression*
*expression-list* **,** *assignment-expression*

*id-expression:*
*unqualified-id*
*qualified-id*

*unqualified-id:*
*identifier*

*qualified-id:*
*class-name* **::** *unqualified-id*

*primary-expression:*
*literal*
**this**
**(** *expression* **)**
*id-expression*

## Literals

*literal:*
*integer-literal*
*character-literal*
*string-literal*

*integer-literal:*
*decimal-literal integer-suffix*$_{\text{opt}}$
*octal-literal integer-suffix*$_{\text{opt}}$
*hexadecimal-literal integer-suffix*$_{\text{opt}}$

*decimal-literal:*
*nonzero-digit*
*decimal-literal digit*

*octal-literal:*
**0**
*octal-literal octal-digit*

*hexadecimal-literal:*
**0x** *hexadecimal-digit*
**0X** *hexadecimal-digit*
*hexadecimal-literal hexadecimal-digit*

*nonzero-digit:*
**1**
**2**
**3**
**4**
**5**
**6**

**7**
**8**
**9**

*octal-digit:*
**0**
**1**
**2**
**3**
**4**
**5**
**6**
**7**

*hexadecimal-digit:*
**0**
**1**
**2**
**3**
**4**
**5**
**6**
**7**
**8**
**9**
**a**
**b**
**c**
**d**
**e**
**f**
**A**
**B**
**C**
**D**
**E**
**F**

*integer-suffix:*
*unsigned-suffix long-suffix*$_{\text{opt}}$
*long-suffix unsigned-suffix*$_{\text{opt}}$

*unsigned-suffix:*
**u**
**U**

*long-suffix:*
**l**
**L**

*character-literal:*
**'** *c-char* **'**

*c-char:*
*any member of the base character set except the single quote ', backslash \, or new-line character*
*escape-sequence*

*escape-sequence:*
*simple-escape-sequence*

*octal-escape-sequence*
*hexadecimal-escape-sequence*

*simple-escape-sequence:*
**\'**
**\"**
**\?**
**\**
**\a**
**\b**
**\f**
**\n**
**\r**
**\t**
**\v**

*octal-escape-sequence:*
\ *octal-digit*
\ *octal-digit octal-digit*
\ *octal-digit octal-digit octal-digit*

*hexadecimal-escape-sequence:*
**\x** *hexadecimal-digit*
*hexadecimal-escape-sequence hexadecimal-digit*

*string-literal:*
**"** *s-char-sequenceopt* **"**

*s-char-sequence:*
*s-char*
*s-char-sequence s-char*

*s-char:*
*any member of the base character set except the double-quote ", backslash \, or new-line character*
*escape-sequence*