Entwurf einer EC++-Spezifikation und -Implementierung auf Basis von ISO C

Diplomarbeit im Fach Informatik

vorgelegt von

Christoph Dietze

geb. 03.06.1980 in Erlangen

angefertigt am

Institut für Informatik
Lehrstuhl für Informatik 2
Programmiersysteme
Friedrich-Alexander-Universität Erlangen–Nürnberg
(Prof. Dr. M. Philippsen)

Betreuer: Volker Barthelmann, Michael Klemm

Beginn der Arbeit: 02.01.2006
Abgabe der Arbeit: 03.07.2006

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch die Informatik 2 (Programmiersysteme), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Diplomarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 03.07.2006

Christoph Dietze

Diplomarbeit

Thema: Entwurf einer EC++–Spezifikation und –Implementierung auf Basis von ISO C

Hintergrund: EC++ (siehe `http://www.caravan.net/ec2plus`) ist eine eingeschränkte Variante von C++ für eingebettete Systeme. Da viele C++–Konstrukte versteckt großen und/oder ineffizienten Code generieren, wurde eine Untermenge von C++ spezifiziert, die derartige Konstrukte verbietet (z.B. Templates, RTTI oder Mehrfachvererbung).

Die EC++–Spezifikation ist verfasst als Liste von Einschränkungen gegenüber der ISO C++–Spezifikation. Die derzeit verfügbaren Implementierungen von EC++ sind i. d. R. eingeschränkte Versionen von vollen C++–Compilern. Da EC++ jedoch sehr viel der "schwierigen" C++–Fähigkeiten eliminiert, ist anzunehmen, dass EC++ sowohl bzgl. Spezifikation als auch Implementierung ähnlich nahe an ISO C liegt wie an ISO C++.

Aufgabenstellung: Im Rahmen dieser Diplomarbeit soll eine alternative EC++–Spezifikation erarbeitet werden, die nicht aus Änderungen des ISO C++ Standards, sondern aus Erweiterungen und Änderungen von ISO C besteht.

Die gänderte C–Spezifikation ist anschließend in einem ISO–C–Compiler zu implementieren und überprüfen. Die erarbeitete Spezifikation ist zusammen mit dem Compiler auf Vollständigkeit zu prüfen. Weiterhin sollen Metriken aufgestellt werden, die es erlauben, die bereits existierende EC++–Spezifikation und die im Rahmen der Arbeit erstellte zu vergleichen und zu beurteilen. Für die Implementierung des EC++–Compilers sind ebenfalls Metriken zu entwickeln, die einen Vergleich zwischen dem EC++–Compiler und dem ursprünglichen ISO–C–Compiler ermöglichen.

Die Ergebnisse der Arbeit sollen sein:

- EC++ Spezifikation auf Basis von ISO C
- um EC++ Unterstützung erweiterter C Compiler
- Erarbeitung von Metriken zur Beurteilung der Spezifikation
- Vergleich (mit den entwickelten Metriken) der auf C basierenden Spezifikation mit der originalen, auf C++ basierenden
- quantitative Analyse des Aufwands der Implementierung

Betreuung: Volker Barthelmann, Michael Klemm, Michael Philippsen

Bearbeiter: Christoph Dietze

Zusammenfassung

EC++ ist eine Programmiersprache, die speziell für eingebettete Systeme entworfen wurde. Sie ist als eine Untermenge der Sprache C++ definiert. Features von C++, welche für eingebettete Systeme als ungeeignet erachtet werden, wurden bei EC++ entfernt. Zum Beispiel werden Templates, Ausnahmebehandlungen und Mehrfachvererbung in EC++ nicht unterstützt. Dadurch, dass ein beachtlicher Teil der C++–Funktionalität entfernt wurde, nähert sich EC++ der Sprache C an. Daher verfolgen wir in dieser Arbeit den Ansatz, EC++ auf Basis von C, an Stelle von C++ zu definieren und betrachten EC++ unter verschiedenen Gesichtspunkten. Erstens geben wir die Definition einer alternativen Grammatik im selben Stil, wie die der offiziellen EC++–Spezifikation beiliegenden Grammatik an. Als Zweites definieren und untersuchen wir die funktionalen Unterschiede und Inkompatibilitäten zwischen C und EC++. Drittens analysieren wir den Umfang einer alternativen, formalen Spezifikation. Unsere Absicht, eine solche Spezifikation in derselben Weise, wie die offizielle Spezifikation zu erstellen, stellte sich als nicht durchführbar heraus. Wir erstellen teilweise eine solche auf C basierende EC++–Spezifikation und stellen fest, dass es nicht möglich ist, diese in einer sinnvollen Art zu definieren. Wir wenden verschiedene Metriken an, um den Umfang der offiziellen und der alternativen Spezifikation zu quantifizieren und kommen zu dem Schluss, dass sowohl die Anwendbarkeit, als auch das Erstellen einer EC++–Spezifikation basierend auf C nicht in einem sinnvollen Rahmen möglich ist.

Die meisten vorhandenen EC++–Übersetzer sind reduzierte C++–Übersetzer. Wir haben einen anderen Ansatz verfolgt, und einen C–Übersetzer zu einem EC++–Übersetzer erweitert. Wir beschreiben, welche Änderungen und Erweiterungen in den C–Übersetzer eingebracht werden müssen, um die Inkompatibilitäten aufzulösen und die neue Funktionalität von EC++ einzuführen. Wir verwenden Metriken, um den Aufwand der Implementierung quantitativ zu bestimmen und betrachten den Arbeitsaufwand als angemessen. Wir wenden auch eine Metrik an, um die Komplexität der Implementierung zu bestimmen. Dies zeigt Funktionen von hoher Komplexität auf, welche schwieriger zu verstehen, zu testen und zu warten sind. Diese Funktionen sind Kandidaten für eine Umstrukturierung.

Abstract

EC++ is a programming language designed for embedded systems that is specified as a subset of the C++ language. Features of C++ that are considered inappropriate for embedded systems are removed from EC++, e. g., templates, exception handling, and multiple inheritance are not supported. By the removal of these significant C++ features, EC++ comes closer to the C language. Hence, we evaluate a different approach on EC++ and define it in multiple respects based on C instead of C++. Firstly, we give the definition of an alternative EC++ grammar in an analogous way to the one accompanied by the official EC++ specification. Secondly, we define and examine the functional differences and incompatibilities between C and EC++. Thirdly, we analyze the extent of an alternative formal specification. Our intention to create such a specification in the same manner as the official one emerged as unmanageable. We partially create such a EC++ specification based on C and observe that it is not possible to define it in a feasible way. We apply different metrics to quantify the extent of both the official and the alternative EC++ specification and come to the conclusion, that the extent of such a specification will be magnitudes larger than the original. Thus, both the creation and the use of a EC++ specification based on C are not sensible.

Most existing EC++ compilers are restricted C++ compilers. We took a different approach and implemented a EC++ compiler by extending a C compiler. We describe what modifications and extensions must be injected into the C compiler to resolve incompatibilities and to insert the new functionality of EC++. We use metrics to quantify the effort of the implementation and evaluate the required work to upgrade an existing C compiler to support EC++ to be moderate. We also use a metric to determine the complexity of the implementation. This reveals functions with high complexity, which are harder to understand, test, and maintain. We identify those functions as candidates for future restructuring.

Contents

List of Figures

List of Tables

1. Introduction

An embedded system is described on Wikipedia as "a special–purpose system, in which the computer is completely encapsulated by the device it controls" [41]. Embedded systems are used in various different environments and utilized in many modern devices. The most prominent usage areas include household devices, vehicles, entertainment electronics, and mobile communication devices.

The microprocessors employed in these devices make up a big market share. In fact, about 98% of all 32–bit processors are used in embedded systems and not in PCs [38]. The market for embedded systems is big and is growing steadily. The worldwide chip market is to grow by 8% in 2006 and by 10.6% in 2007 [49]. Embedded systems are employed in special–purpose environments; general–purpose PCs, on the contrary, must be flexible to perform many different tasks. Naturally, requirements for embedded systems differ from those for PCs. Typical requirements for embedded systems are reliability, meeting real–time constraints, and low price because of mass–production [41].

To meet these requirements, also the demands on the programming language used to program such systems differ from those for PC applications. Major demands are efficient memory usage and small program size. Since savings in these two areas often lead to lower costs, it is important for a programming language not to impose unnecessary or unwanted overhead (so–called code bloat).

What is the best fitting programming language, depends on the application of the embedded system. In many cases, low–level assembly languages are used to program embedded systems. Such languages give direct control over the resulting machine code and thus enable the programmer to write highly efficient software. When the programming task is small and of little complexity, there often is no need for a high–level language. The use of a high–level language with more abstraction capabilities can yield many advantages, which include better portability, reusability, and of course easier management of complex systems [5]. Programming in high–level languages also is less error–prone than assembly programming.

Probably the most commonly used high–level programming language for embedded systems is C. This is due to C being well understood, having little potential to create overhead, and good C compilers being widely available. C++, which offers even more abstraction mechanisms—by object–oriented, generic programming, and metaprogramming [43]—is also often used for embedded systems programming. However, some of C++'s language features are less appropriate for embedded systems. For example, when using template features without care, the program size can increase immensely. Another example: when exception handling is used, asserting real–time constraints can become difficult [28].

The EC++ (Embedded C++) language [13] is especially designed to meet the requirements of embedded systems. It is derived from C++ and removes features from C++ that are less suited for embedded systems programming. Thus, EC++ forms a proper subset of C++. The basic idea when designing EC++ was to create a language that offers the object–oriented programming paradigm of C++, and to remove the features inappropriate for embedded systems. The most prominent features that were removed from C++ are templates, exception handling, and multiple inheritance. In addition to the language, there is also a library specified for EC++. This thesis focuses solely on the language, not the library.

Historically, Dennis Ritchie began work on the language called “C” in 1969, many features of which were derived from a language named “B” by Ken Thompson. By 1978 Ritchie and Brian Kernighan published the first edition of “The C Programming Language” [22], which served as the first informal specification of C—this version of C is commonly referred to as “Kernighan and Ritchie C” or as “K&R C”. In 1979, Bjarne Stroustrup began work on a language extending C, which was called “C with classes” at the time, 1983 its name was changed into “C++”. In 1989, the American National Standards Institute (ANSI) completed a standard of C, which with minor modifications was also adopted by the International Organization for Standardization (ISO) as ISO/IEC 9899:1989 [18], commonly referred to as “C89” and “C90”. Over the years, many new features found their way into C++ and in 1998 a joint ANSI–ISO committee ratified a standardized C++ as ISO/IEC 14882:1998 [20]. Throughout this thesis, when we refer to C++, we refer to this standard. A new revision of the C standard was released in 1999 as ISO/IEC 9899:1999 [21] and was adopted as an ANSI standard in 2000. When we use the name C, we actually mean C as specified in the C99 standard.

Since 1979, C and C++ have evolved independently. Changes and new features were introduced in both languages. C adopted some features from C++ and vice versa. Also, incompatibilities between the two languages came up. Figure 1.1 views the three languages as sets and depicts common functionality, where the sets overlap. The gradient between C and EC++ symbolizes the incompatibilities between C and EC++, which are also incompatibilities between C and C++.

In this thesis, we investigate the issues of an alternative EC++ specification based on C rather than C++. We show, what parts of C have to be removed, which incompatibilities have to be resolved, and which features of C++ have to be added. Our initial intention was also to create a formal alternative specification based on C using the same style as the official specification does. However, this specification has shown to be too complex to be created in a sensible way. Most available EC++ compilers are stripped–down C++ compilers. In this thesis, we also examine the steps required to upgrade a C compiler to compile EC++. We also implemented this functionality into an existing C compiler. Then, we compare the official with the alternative spec-

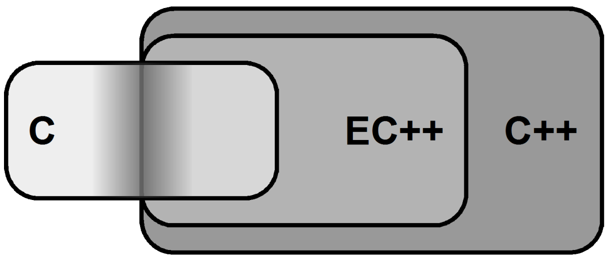


Figure 1.1.: Relationship between C, EC++, and C++.

ification using metrics to quantify the extent and complexity. We also use metrics to determine the complexity of the implementation if EC++ based on a C compiler.

In Chapter 2 we first present the specification of EC++ as it is officially defined, based on C++. Afterwards we give an alternative specification, based on C. In Chapter 3 we extend an existing C compiler to also support EC++, and describe the required steps to do so. In Chapter 4 we quantify and compare the extent of the two alternative specifications and also the effort required to extend the C compiler to support EC++. In Chapter 5 we have a look at related work. Chapter 6 lists some topics for future work and in Chapter 7 we give a conclusion of the thesis.

2. Specification of EC++

In Section 2.1 of this chapter we will present the official EC++ specification which is based on C++. In Section 2.2 we will look into what is necessary to specify EC++ not based on C++, but on C. Many adjustments of the specification are required and we will examine some in greater detail. Then we investigate the extent and complexity of defining a formal specification of EC++ based on C. Concluding we will give a summary of the differences between C and EC++ in Section 2.3.

2.1. Official Specification of EC++ based on C++

The official Embedded C++ specification [13] is defined by the Embedded C++ Technical Committee. The committee basically is a collaboration of Asian companies. Its most important members are Fujitsu, Motorola, NEC and Toshiba. The most recent version is WP-AM-003, which was released in October 1999.

The objective of EC++ as stated by the committee is to "provide embedded systems programmers with a subset of C++ that is easy for the average C programmer to understand and use", to "retain the major advantages of C++", and to "fulfill the particular requirements of embedded systems designs", as well as to "make the specification as small as possible while retaining the object–oriented features" [11].

The EC++ specification is defined as a list of differences to the C++ specification [20], rather than a document to be read for itself. Almost all of the differences are brief instructions to omit a chapter, paragraph, subparagraph, or footnote from the C++ specification. It defines the EC++ language as well as the EC++ library. However, in this thesis we will only be concerned with the language specification.

With embedded systems and their small memory capabilities being the target platform, features that unavoidably or potentially introduce overheads are candidates to be omitted from EC++. As well, features that are difficult to use sensibly for a programmer having only experience in C, are candidates for removal. The following list gives brief descriptions of the features omitted from C++ as described in the EC++ Rationale [11]:

- **Mutable specifiers** are used to declare a class member `mutable`, which allows that member to change its value, even when the class object is declared as `const`. Especially for embedded systems, it is important to be able to decide whether an object is really constant and thus can safely be put into ROM. Since the `mutable` specifier might confuse an embedded systems programmer, it is excluded from the EC++ specification.

- **Exception handling** enables more elaborate handling of errors by the concept of throwing and catching exceptions. When an exception is thrown, the stack is unwound until an appropriate exception handler is found that catches the exception. In C, errors are usually signaled by the return value of functions. However, besides having more control when dealing with errors, there are also some drawbacks, especially for embedded systems programming. It is difficult to determine the runtime and memory consumption when using exception handling. Also the program size increases when exception handling is used. Therefore, exception handling is excluded from in the EC++ specification.

- **Runtime type information (RTTI)** allows to determine the exact type of an instance at runtime by using the `typeid` keyword. Casting using `dynamic_cast` also makes use of RTTI. Using these facilities is useful in programs that make heavy use of polymorphism. However, RTTI unavoidably introduces some overhead. The overhead and embedded systems programs typically making little use of RTTI (because they are usually small) are the reasons, why RTTI is excluded from the EC++ specification.

- **Namespaces** create contexts for identifiers and thus allow using the same identifier in different namespaces. This is especially useful in large–scale programs to structure the code and to avoid name clashes. Due to the target processors of EC++ having little memory, programs are usually not very large and name conflicts rarely occur. Hence, namespaces are excluded from the EC++ specification.

- **Templates** enable generic programming. They are a very powerful language tool, especially when used along with inheritance and operator overloading. Templates do not necessarily introduce overhead. However, when used carelessly, program size can increase dramatically. Using templates efficiently requires a high level of experience. Learning to use templates well will require a large of amount of time for the average C programmer. For these reasons, templates are excluded from the EC++ specification.

- **Multiple inheritance and virtual inheritance** appropriately can lead to expressive and elegant class hierarchies. However, designing such hierarchies is difficult even for expert programmers in that field. Programs that use only single inheritance tend to be more understandable, readable, and maintainable. Virtual inheritance only makes sense in association with multiple inheritance. Therefore, multiple inheritance and virtual inheritance are excluded from the EC++ specification.

2.2. Specification of EC++ based on C

One objective of this thesis is to propose a specification of EC++ based on C. Since EC++ removes large portions from C++, it shifts further towards the C language specification. At first glance, it seems possible to define an alternative specification for EC++ based on C rather than C++. However, creating such a differential specification on the same level— i. e., a list of all textual changes needed for the specification—as the original specification has shown to become unmanageable.

In Section 2.2.1 we present the differences between C and EC++ from a functional point of view. In Chapter 2.2.2 we introduce an alternative grammar for EC++ based on the C grammar and in Section 2.2.3 we investigate the complexity of a formal specification of EC++ based on C.

2.2.1. Functional Differences between C and EC++

In this section, we will examine the different functionalities and the incompatibilities between C and EC++. With additional functionality we denote that one language allows certain constructs, which result in an error in the other language. Incompatibilities are constructs that are legal in both languages, but result in different behavior. In this sense, EC++ and C++ are compatible and the incompatibilities between C and C++ are the same as between EC++ and C++. The additional functionality of C is also the same when compared to either C++ or EC++. Stroustrup describes some incompatibilities between C and C++, mainly meant as an aid for programmers to write more portable code [29]. A more comprehensive list is given by Tribble, who presents a list of 48 differences between C and C++ [37]. The differences shown in both sources do not cover the new features of C++ compared to C. The differences between C and EC++ can be organized into three categories. Exemplarily, we present some issues from each category. For a summary of all differences, see Section 2.3. These are the three categories:

- Functionality of EC++ not present in C (Section 2.2.1.1)
- Functionality of C not present in EC++ (Section 2.2.1.2)
- Incompatibilities between C and EC++ (Section 2.2.1.3)

2.2.1.1. Functionality of EC++ not present in C

We will give a brief overview of some of the additional features of EC++ when compared to C. An extensive discussion of these features is beyond the scope of this thesis, see [29] for more detailed descriptions.

Classes together with their associated features form the foundation of object–oriented programming in EC++. Additionally to data members, classes (or structures) can contain methods, inherit from other classes, control component access and have friends. In C, structures lack that additional functionality.

```
class Cursor {
  int x,y;
public:
  int getX(); int getY();
};

class MouseCursor : public Cursor {
  bool buttonState;
public:
  bool getButtonState();
};
```

User–defined operators allow further customization of classes in EC++. Operators have a behavior defined by the language for its built–in types. User–defined operators allow to define the semantics of operators for user–defined types. In C, operators are only allowed to be used on the built–in types.

```
struct Address {
  int number;
  char name[32];

  bool operator>(Address& a) {
    return number > a.number;
  }
};
```

Overloading allows the definition of multiple functions with the same name but different parameters. When an overloaded function is called, disambiguation rules are used to determine which instance of the function to call. The selection is done at compile time. C does not allow multiple definitions using the same name.

```
int square(int a) {
  return a*a;
}

float square(float a) {
  return a*a;
}
```

Empty structures are structure declarations with an empty body and are not allowed in C. In EC++ however, such structures can legally be declared.

```
struct A {
};
```

2.2.1.2. Functionality of C not present in EC++

In this section, we will present some of the additional features of C when compared to EC++. These are features, which C supports and that will result in errors when used in EC++.

Variable–length arrays (VLAs) were introduced in C99. VLAs are arrays of variable size that are allocated on the stack, so the expression defining the size does not have to be a compile–time constant. VLAs can also be used as function parameters, allowing a variable expression as array size or `[*]` for unspecified array size. Due to VLAs, the way a `sizeof` expression is evaluated changes, since it does not always result in a constant value. EC++ does not support VLAs: array sizes are required to be specified by constant expressions.

```
void f(int x) {
  int arr[x];
  // ...
}

void f(int n, float a[n]);

void g(int a[*]);
```

Designated initializers are also a new feature of C99. They allow initialization of specific components in structures and unions using the components' name. Initialization of arrays can be specified using the subscript. EC++ does not support designated initializers. Thus, structure initializers must follow the order, in which its components are declared and array initializers begin at the first element.

```
struct Address {
  int number;
  char name[32];
};

struct Address a = {
  .number = 496,
  .name = ”test”
};

struct Address b[] = {
  [6] = { .number = 8128 },
  [28] = { .name = “Peter”,
           .number = 496 }
};
```

Compound literals were also introduced in C99. Additionally to the specification of literals of primitive types, compound literals allow the specification of structures and array types in constant expressions. This is not supported in EC++. However, comparable functionality can be achieved using constructors in EC++.

```
struct Address {
  int number;
  char name[32];
};
void f(struct Address);
void g(int[2]);

void h(void) {

  f((struct Address){496, “test”});

  g((int[2]){6,28});
}
```

2.2.1.3. Incompatibilities between C and EC++

In this section, we will show some of the incompatibilities between C and EC++. These are issues, where constructs are allowed in both C and EC++, but result in different behavior.

Function name mangling is the process that generates a unique symbolic name for a function or variable. In C, identifiers are used directly as symbolic names. However, in EC++ this is insufficient because member functions of different classes can have the same name and because of function and operator overloading. Therefore, C and EC++ will in general produce different symbolic names. How the mangling is to be done is not defined in the EC++ specification but is left to the compiler implementation. To be able to call a function compiled by a C compiler from EC++, that function must be declared with an appropriate linkage specification using `extern "C"`. This tells the compiler, not to mangle the name of that function, but to use the C naming conventions. Linkage specification is an additional feature of EC++, which is not allowed in C.

Character literals have different types in C and EC++. In C `'a'` is of type `int`, whereas it is of type `char` in EC++. This can lead to incompatible code when using `sizeof` expressions. In the improbable case, that an implementation uses the same sizes for `char` and `int`, these problems do not show up.

```
void f(void) {

  if(sizeof('a')==sizeof(int))
    printf("C");

  if(sizeof('a')==sizeof(char))
    printf("EC++");
}
```

Empty parameter lists have different meanings in C and EC++. A function declaration with no parameters in EC++ declares a function that expects no arguments. In C, it declares a function with unspecified parameters. A declaration with `void` as the only parameter declares a function with no parameters in both languages.

```
void f();

void g(void) {

  f(6,28); // ok in C
           // not ok in EC++
}
```

Nested structure tags show different behavior in C and EC++. In C, a structure, union, or enumeration declared within a structure declaration has global scope. In EC++, these have class scope. Therefore, in C, those declarations are visible at global scope, but are not visible in EC++.

```
struct A {
  struct B { int x; } b;
  enum E { ENTRY1 } e;
};

struct B c; // visible in C
            // not visible in EC++

enum E f; // visible in C
          // not visible in EC++
```

2.2.2. Grammar of EC++ based on C

Each of the three languages (C, EC++, and C++) provides, along with their specifications, a grammar that accepts the associated language. These are context–free grammars and are given in Backus–Naur form [3]. The grammars actually accept supersets of the associated languages. This means, that all legal programs are accepted, but also programs that are incorrect (i. e., ones that violate semantic rules) may be accepted. Accepted programs are said to be syntactically correct.

The Embedded C++ Technical Committee presents a grammar for the EC++ language [12]. This grammar is defined differential to the C++ grammar as given in the C++ specification. No new rules are introduced, only existing rules are completely or partially removed.

We created a grammar in an analogous way, which is based on the C grammar. This requires not only deletion, but also alteration and insertion of rules and symbols. We paid attention to keep the differences small. As much as possible, we inserted the rules for the additional EC++ features from the official EC++ specification without change. In some cases, replacing some rules completely with the ones from the official EC++ grammar was sensible. For example, the rules that describe structure declarations were removed and replaced with the rules of the official EC++ grammar that cover classes, inheritance, members, access control, etc. This approach—compared to adjusting and extending the existing rules for structures—led to a much clearer grammar. Appendix B shows the resulting grammar. Note that this grammar and the one defined by the EC++ committee do not accept exactly the same languages, but both accept supersets of the EC++ language. Syntactically correct programs that do not respect the EC++ semantics are rejected by the semantic analysis.

2.2.3. Formal Specification of EC++ based on C

One intention of this thesis is to produce an alternative formal specification of EC++, based on C instead of C++. The way it is written should follow the one used in the official specification: a complete list of all textual differences, that need to be applied on the underlying specification.

However, the task of creating such a specification has shown to be hardly manageable. The way in which the official specification is written, is unsuitable for the new one. In the official EC++ specification, all differences are short statements, almost always to omit a certain phrase, paragraph or chapter. However, when the basis is C, many alterations and insertions are required.

To get a better picture of the extent of the alternative specification, we took one section from the C specification (section "6.5.2.2 Function calls"), for which there is a closely corresponding section in the C++/EC++ specification (section "5.2.2 Function call"). For that section, we wrote a list of differences in the same way as the official EC++ specification is written (see Appendix A).

The result of our effort is a differential specification that is just as long as the source section and that quotes large portions from the EC++ specification. Note that the additional information quoted from the EC++ specification is not from a chapter respective to the new features of EC++. An issue that we did not address is fitting the references of the quoted EC++ text, since we have no corresponding information in the example. This section is just about 2 pages long, so this should only be used to get an impression. There are sections for which less adjustment will be necessary, but there are also sections that will be much harder to adjust. For example, sections that do not even have such a clear correspondence in the EC++ specification will make adjustment more complicated.

In fact, about one third of all paragraphs in the C specification are not correct for EC++ and must be adjusted, replaced, or removed. The remaining paragraphs also need editing because of either incompatibilities or the additional EC++ functionality. The new EC++ features are defined in the C++ specification. Issues concerning those new features are not only found in their respective chapters, but are also spread throughout the whole document. Taking this information and extending the C specification with it, is not sensibly manageable. Only very little of the C specification could remain as is. Essentially, it is comparable to replacing the C specification with the C++ specification or to writing an entirely new specification.

We estimate the extent of the alternative EC++ specification based on C to be comparable to that of the whole C specification (150–200 pages). In comparison, the official specification fits on 18 pages. So the alternative specification would also be inferior to the original concerning readability and understandability. For specifications in general, completeness, and correctness are essential, which are at least difficult to assert.

2.3. Summary of Differences between C, EC++ and C++

In this section, we present two tables, which summarize the differences between C, EC++, and C++. Table 2.1 shows the differences in supported features between the three languages. Some of these are described in Sections 2.1 and 2.2.1. Table 2.2 summarizes the incompatibilities between C and EC++ and between EC++ and C++. With C++ being a proper superset of EC++, there are no incompatibilities between these languages. Some of these issues were described in Section 2.2.1.3. Also see [29, 37] for more extensive descriptions of the features and incompatibilities mentioned.

Feature	C	EC++	C++
Exception handling	✗	✗	✓
Multiple inheritance	✗	✗	✓
Namespaces	✗	✗	✓
Runtime type information (RTTI)	✗	✗	✓
Templates	✗	✗	✓
Virtual inheritance	✗	✗	✓
Aggregate initializers	✗	✓	✓
Alternate punctuation spellings	✗	✓	✓
Anonymous unions	✗	✓	✓
Boolean type	✗	✓	✓
Classes	✗	✓	✓
Conditional expression declarations	✗	✓	✓
Default arguments	✗	✓	✓
Empty structures	✗	✓	✓
Function and operator overloading	✗	✓	✓
Functions returning `void`	✗	✓	✓
`new` and `delete`	✗	✓	✓
References	✗	✓	✓
User–defined operators	✗	✓	✓
Compound literals	✓	✗	✗
Designated initializers	✓	✗	✗
Dynamic `sizeof` evaluation	✓	✗	✗
Enumeration constants	✓	✗	✗
Flexible array members	✓	✗	✗
Hexadecimal floating–point literals	✓	✗	✗
IEC 60559 arithmetic support	✓	✗	✗
`long long` integer type	✓	✗	✗
Non–prototype function declarations	✓	✗	✗
`pragma` keyword	✓	✗	✗
Predefined identifiers (`__func__`)	✓	✗	✗
`restrict` keyword	✓	✗	✗
Variable–argument preprocessor function macros	✓	✗	✗
Variable–length arrays	✓	✗	✗

Table 2.1.: Summary of differences between C, EC++ and C++.

Feature	C ↔ EC++	EC++ ↔ C++
Character literals	↯	✓
Comma operator results	↯	✓
`const` linkage	↯	✓
Duplicate type definitions	↯	✓
Empty parameter lists	↯	✓
Empty preprocessor function macro arguments	↯	✓
Enumeration declarations with trailing comma	↯	✓
Enumeration types	↯	✓
Function name mangling	↯	✓
Function pointers	↯	✓
Inline functions	↯	✓
Keywords	↯	✓
Nested structure tags	↯	✓
One definition rule	↯	✓
`static` linkage	↯	✓
String initializers	↯	✓
String literals are `const`	↯	✓
Structures declared in function prototypes	↯	✓
Type definitions versus type tags	↯	✓
Variable–argument function declarators	↯	✓
`void` pointer assignments	↯	✓

Table 2.2.: Summary of incompatibilities between C, EC++ and C++.

3. Extending a C Compiler to Support EC++

In this chapter we will discuss what is necessary to upgrade an existing C compiler to compile EC++ code in addition to C code. The C compiler we chose for this task is vbcc [4]. It is a highly optimizing, portable, and retargetable ISO C compiler. It supports ISO C according to C89 and a subset of the new features of C99. However, we pretend that we are upgrading a fully C99 compliant compiler. Actually, vbcc includes all features from C99 that are also allowed in EC++. Only new features of C99 that are not allowed in EC++ need to additionally be removed from a full C99 compliant compiler; for vbcc no change is needed, since those features are not supported in the first place. We focus solely on issues concerning the EC++ language, not the EC++ library.

3.1. Compiler Phases of vbcc

The processing that a source file undergoes in the compiler until the assembler code is produced can be partitioned into different compiler phases (see Figure 3.1). Such an organization is used by most compilers for any language and is documented in many books and papers about compiler construction, e. g., [2, 48, 6]. In the actual implementation the phases are not as cleanly separated as depicted in the figure, e.g., in the case of vbcc the preprocessing and lexical analysis phase are performed in the same step. The front end and back end however, can clearly be distinguished. The front end eventually generates intermediate code, which is optimized and then fed to the back end. The vbcc back ends are exchangeable and they are available for different platforms.

The preprocessor and lexer module reads from the source file and splits it into a stream of tokens. A token is an atomic unit of the language, e.g., keyword, identifier, constant, operator, parenthesis, etc. The input is scanned just in time, which means that only when the parser requires the next token, the lexical analysis of the source file progresses and returns the next token to the parser. The vbcc compiler uses a recursive descent parser [1] to recognize the structure of the stream. Thus, the parser essentially is a set of functions which closely reflect the rules in the grammar. The parser triggers semantic analysis and intermediate code generation when appropriate. For example, when the input is an expression, the parser creates an Abstract Syntax Tree (AST)

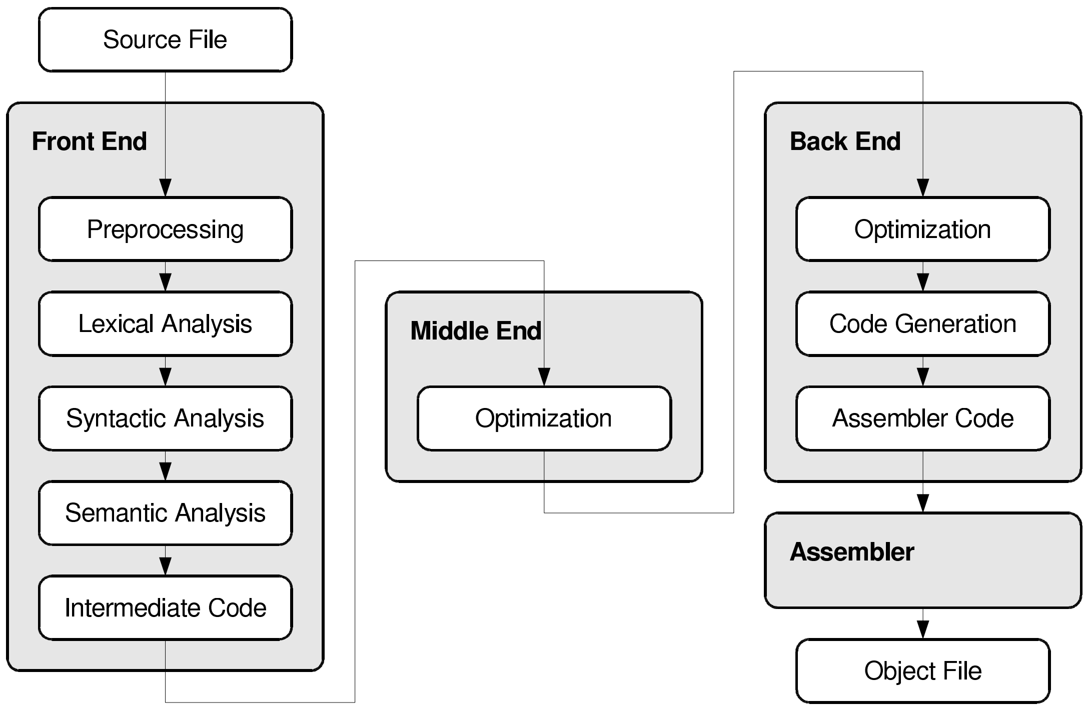


Figure 3.1.: Compiler phases.

for that expression and afterwards triggers semantic analysis to attribute the AST of the current expression. Then, intermediate code generation for the attributed AST is called and thereafter parsing continues with the next statement of the input.

What follows is an overview of the modifications that have to be applied in each specific phase.

- **Preprocessing phase**
  Minor modifications have to be applied concerning the preprocessor, mostly to remove additional functionality from C. For example, C supports function macros with a variable number of arguments, EC++ does not allow this.

- **Lexical analysis phase**
  Aside from some minor modifications of removing some C functionality (e.g., universal character names), the set of accepted keywords also has to be adjusted. Note that the keywords that are used for C++ functionality that is not part of EC++ are still reserved. New tokens also are: “`->*`”, “`.*`”, and “`::`”.

- **Syntactic analysis phase**
  The parser has to be modified to accept all of the EC++ constructs. This includes method and user–defined operator declarations and definitions, base class specifiers, nested name specifiers, etc.

- **Semantic analysis phase**
  Besides additionally having to process EC++ constructs, semantic analysis has to be modified to address numerous other EC++ specific issues, e. g., method calls, pointers to methods, casting of pointers, calls of user–defined operators and references. Some of these cases are discussed in depth in Section 3.2.

- **Intermediate code and subsequent phases**
  No additional functionality is required for the intermediate code, hence no modifications on the intermediate code generation phase and subsequent phases are necessary. All of the optimizations that may be applied for C may also be applied to EC++. However, there are additional optimizations that are applicable for EC++. Some optimizations that are planned for future versions of vbcc are described in Chapter 6.

3.2. Implementation Details

In this section we will discuss some of the implementation issues of EC++ in depth. It is not a complete list of all changes required to upgrade a C to an EC++ compiler. The basic concept, we use is to reduce the new EC++ constructs to equivalent C constructs. This is essentially the same approach that Cfront [39] uses. Cfront was the original compiler for C++, which converted C++ to C. In contrast to Cfront, we do not generate C code, but perform the transformations from EC++ to C constructs in vbcc's internal data structures. Note that not all EC++ features have been implemented and thus are not discussed in this section. See Section 3.3 for a discussion of the missing features.

3.2.1. Name Mangling

In EC++, different functions and methods are allowed to have the same identifier if they are declared in different scopes or if their parameters differ as required for overloading. So, a unique, symbolic name has to be created, that is used to refer to the specific function instance. Symbolic names must also be created for user–defined operator functions. This is not required for variables, since different variables (of different type) are not allowed to use the same identifier. The process of creating the symbolic name is called name mangling. In C, this is not an issue, since neither methods nor functions that have the same identifiers are allowed. How to create the mangled names is not defined in the EC++ specification but is implementation dependent.

For vbcc we chose the mangling scheme defined by Intel in the Itanium–64 ABI [9]. The GNU Compiler Collection [14] also uses this method in version 3.x and 4.x. In this scheme, mangled variable names are the same as their plain name. Mangled

EC++ Code	Mangled Name
`int x;`	`x`
`void f();`	`_Z1fv`
`int g(int,float,double);`	`_Z1gifd`
`void hh(int***);`	`_Z2hhPPPi`
`struct A {`	
`  int f(double);`	`_ZN1A1fEd`
`  static int abc(float);`	`_ZN1A3abcEf`
`};`	
`void abc(A);`	`_Z3abcN1AE`
`typedef int (func)(double);`	
`void f(func);`	`_Z1fPFidE`

Table 3.1.: Name mangling scheme used for vbcc.

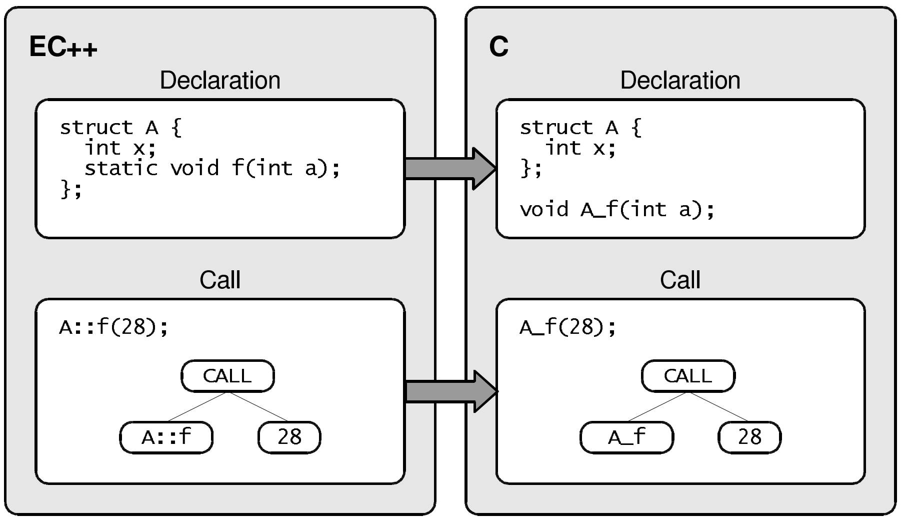
names of functions and methods have a prefix of `_Z` followed by their encoded scope, identifier, and parameters. Names of scopes and identifiers are mangled as their plain names prefixed with their length. Scoped names additionally have a `N` prepended and an `E` appended, e. g., `1f`, `3abc`, `N5these3are7encoded6scopes3varE`, etc. Parameters are encoded according to their type, e. g., `int` is encoded as `i`, pointer to double as `Pd`. For a full specification, see [9]. Note that neither specifiers such as `static` or `virtual`, nor the implicit `this` pointer are encoded in the mangled name. Table 3.1 shows some examples of this method. The EC++ specification forbids user–declared variables to have a `_Z` prefix, thus name clashes are avoided.

For better readability, we will use a simpler scheme in the examples shown below. For example, the method `f` of class `A` receives the mangled name `A_f`.

3.2.2. Method Calls

A function declared inside a class or struct declaration is called a method. In C, function declarations inside structures are not allowed. The behavior of methods is defined in the EC++ specification, but how to realize it is left to compiler implementation. In the implementation for vbcc we used what is probably the method used most commonly by compilers. This method is described in many books about compiler construction for object–oriented languages, e. g., [2, 48].

All methods are compiled as global functions. When a method is called, the syntactic analysis transforms that call expression into the appropriate AST nodes in the same way as an ordinary function call would be transformed. That the expression is a method call is detected during the semantic analysis of the method identifier node. Then the specific method to be called is determined following the scoping and overloading rules. Afterwards, it must be checked if the access rules allow calling that



```
struct A {
  int x;
  static void f(int a);
};
```

```
A::f(28);
```

```
struct A {
  int x;
};

void A_f(int a);
```

```
A_f(28);
```

Figure 3.2.: Declaration and call of a static method.

method from the current context. All method calls can be transformed into C compliant function calls. This principle is used to transform the AST representing a method call into an equivalent AST containing only C compliant constructs. Two different kinds of calls can be distinguished. If the function to be called can be determined at compile–time, it is a *static* call. If the actual function to call must be determined at runtime, it is called a *dynamic* call. Dynamic calls are used for virtual functions.

The names of these function are mangled as specified in [9], but in the figures we will use the simpler scheme, as said in Section 3.2.1. The following figures containing examples will show the original EC++ source code as written by the programmer, the equivalent C code, the AST as it is before the transformation and the AST after transformation.

3.2.2.1. Static method calls

The declaration of the static method is internally transformed into a declaration of a global function using the mangled name and having the same parameters and return type. A call to a static method is transformed into a static call of the according function, the arguments to the function/method remain unchanged. Thus, static methods may be mapped to global functions in a straightforward way. See Figure 3.2.

3.2.2.2. Non–virtual method calls

When a non–virtual, non–static method is defined, internally, a global function with the mangled name is defined. In addition, that function receives the implicit `this` pointer as its first parameter. The user–defined parameters are appended after it. The `this` pointer parameter is of type pointer to class, to which the method belongs.

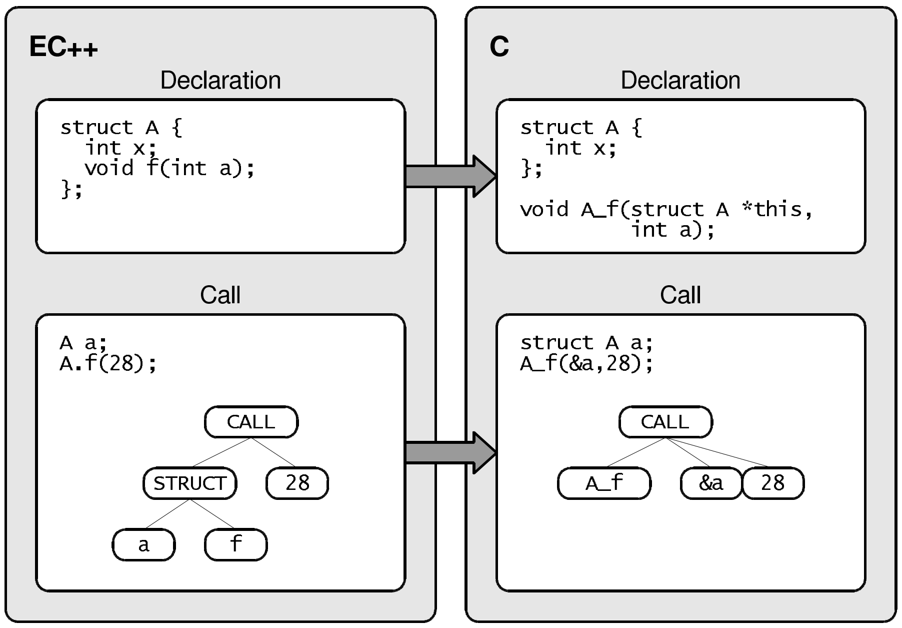


Figure 3.3.: Declaration and call of a non–virtual method.

When a non–virtual method is about to be called, the AST for the call is transformed to a static call to the according global function and the implicit `this` pointer argument is passed to the function as the first argument (see Figure 3.3). The `this` pointer argument can either be explicitly given (e.g., `a.f()` or `ap->f()`) or, if the context is a non–static method, the `this` pointer of that surrounding method is used.

3.2.2.3. Virtual method calls

Firstly, let us give a brief example of when and how a virtual method call behaves differently from its non–virtual counterpart (see Figure 3.4). Suppose a class `B` declares a method and a derived class `D` declares a method with the same name and the same parameters (this method *overrides* the base class' method). In EC++, it is allowed to cast a pointer to a derived class to a pointer to the base class. Further suppose a method is invoked for a pointer to the base class `B`, but which actually points to an instance of the derived class `D`. If the method was not declared as virtual (`f`), the pointer type would determine which function to actually call. So, this would be a static call of the base class' method (`B::f`). On the other hand, if the method was declared as virtual (`g`), the method of the concrete object's type would be called, which is the derived class' method (`D::g`). This is a dynamic call, since the type of object pointed to must be evaluated at runtime. Virtual methods are always non–static and also carry an implicit `this` pointer.

How to achieve this behavior is not defined in the specification but is left for the compiler implementation. We use the most common approach: using virtual tables.

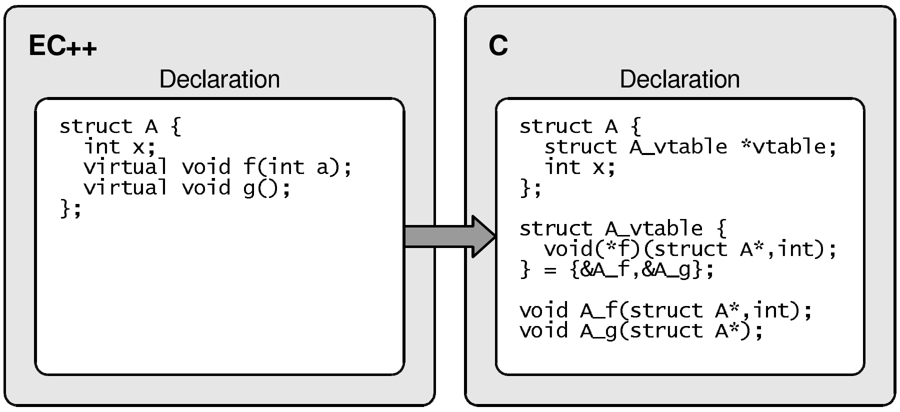
```
struct B {
  void f();
  virtual void g();
};

struct D : B {
  void f();
  virtual void g();
};

void test(){
  D d;
  B *p=&d;

  p->f(); // calls B::f()
  p->g(); // calls D::g()
}
```

Figure 3.4.: Different behavior of virtual and non–virtual methods.



```
struct A {
  int x;
  virtual void f(int a);
  virtual void g();
};
```

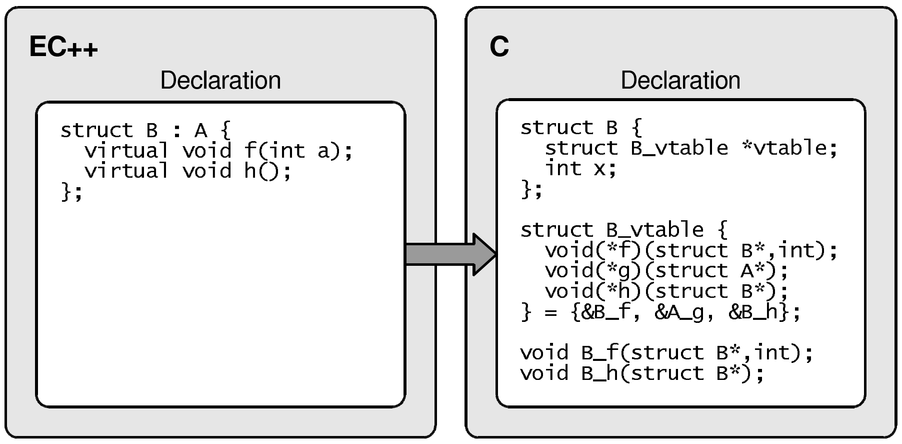
```
struct A {
  struct A_vtable *vtable;
  int x;
};

struct A_vtable {
  void(*f)(struct A*,int);
} = {&A_f,&A_g};

void A_f(struct A*,int);
void A_g(struct A*);
```

Figure 3.5.: Virtual table creation and initialization.

Descriptions of this method can be found in many compiler books for object–oriented languages, e. g. [2, 48]. For each class that has virtual methods (either declared in itself or inherited from a base class), a virtual table is generated. A virtual table is a constant structure that contains the addresses of all virtual methods for the associated class. It suffices to create just one instance of a virtual table per class and to have a pointer in each class instance point to this table. For an example see Figure 3.5. A derived class inherits all virtual methods of the base class, virtual methods that are added to the derived class are appended to the table. This way, a method has the same offset in the virtual table of a derived class as it has in the base class. When a derived class overrides a virtual method, the appropriate entry in the virtual table is replaced with the address of the overriding method. The virtual table pointer is not inherited by derived classes, but replaced with pointers to the virtual table of the derived class. For an example, see Figure 3.6.



```
struct B : A {
  virtual void f(int a);
  virtual void h();
};
```

```
struct B {
  struct B_vtable *vtable;
  int x;
};

struct B_vtable {
  void(*f)(struct B*,int);
  void(*g)(struct A*);
  void(*h)(struct B*);
} = {&B_f, &A_g, &B_h};

void B_f(struct B*,int);
void B_h(struct B*);
```

Figure 3.6.: Virtual table creation and initialization in a derived class.

When a virtual method is invoked, the virtual table pointer of the associated instance is dereferenced. Then the appropriate method address is looked up in that virtual table, and the function at that address is called (see Figure 3.7).

Performing a dynamic function call compared to a static function call is costly. It requires two additional pointer dereferencings and makes inlining of that function difficult. Also, modern processors can benefit from pre–fetching instructions when jumping to a constant address. Thus, replacing dynamic calls with static calls wherever possible can improve runtime performance significantly. This is possible, when a method is always called for the same class type. The most common case is, when the method is called for a concrete instance—rather than for a pointer to instance—, for an example, see figure 3.8.

Care must be taken when a virtual method is called for an expression that may have side effects. For example, such side effects may occur, when a function that returns a pointer to a class type is called in the expression (see Figure 3.9). Transforming such a construct the same way as shown in Figure 3.7 would invoke the side effects of `h` twice—once when the virtual table pointer is looked up and once when the `this` pointer is passed. Therefore, a temporary variable must be created and the value of the expression must be assigned to it. Then that temporary is used for both the lookup and the `this` pointer. This way, the expression and its side effects are evaluated only once.

3.2.3. Pointer Conversion

When casting a pointer to a derived class to a pointer to the base class another issue must be addressed. Assume, a base class `B` has no virtual methods, and a class derived from it (`D`) has virtual methods (see Figure 3.10). Thus, `D` must receive a virtual table pointer as its first component, whereas `B` must not have such a pointer. When a cast

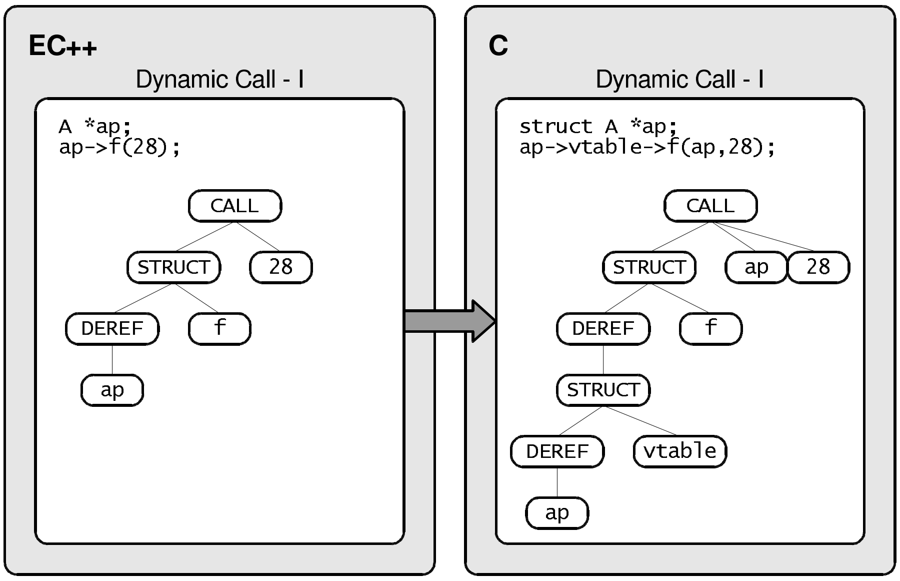


Figure 3.7.: Dynamic call of a virtual method.

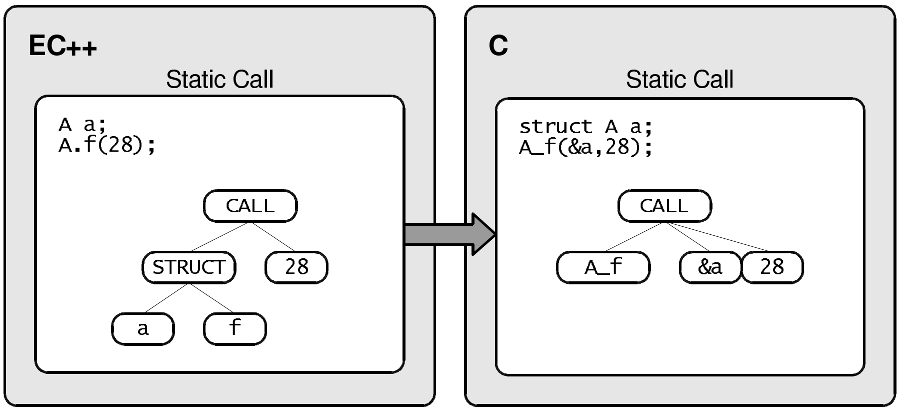
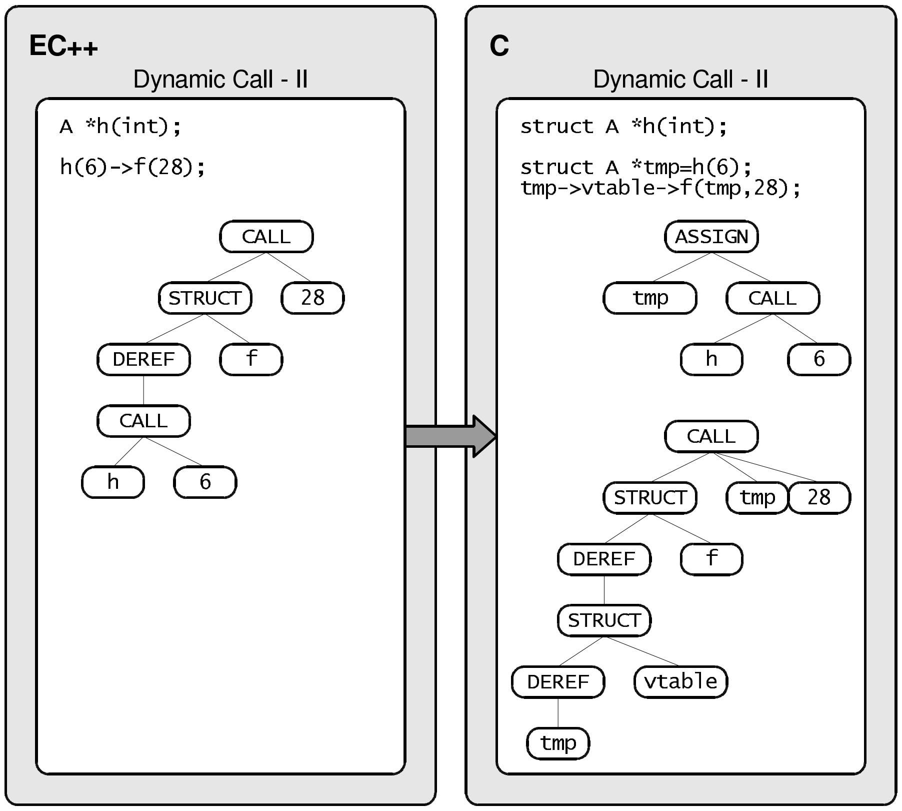


Figure 3.8.: Static call of a virtual method.



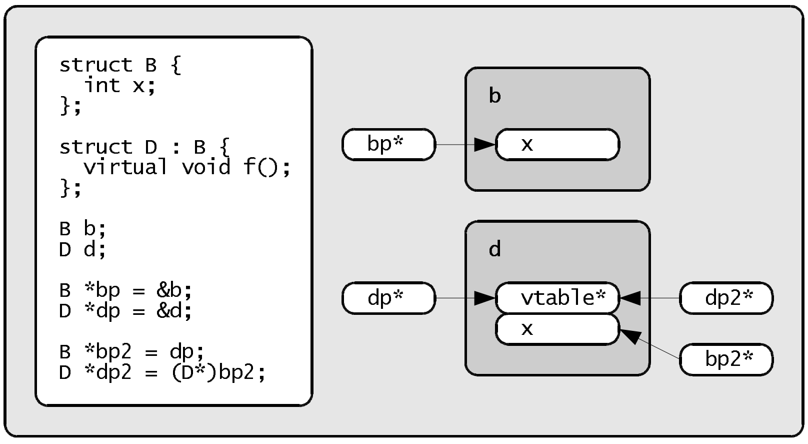
```
A *h(int);

h(6)->f(28);
```

```
struct A *h(int);

struct A *tmp=h(6);
tmp->vtable->f(tmp,28);
```

Figure 3.9.: Dynamic call of a virtual method on an expression with side effects.



```
struct B {
  int x;
};

struct D : B {
  virtual void f();
};

B b;
D d;

B *bp = &b;
D *dp = &d;

B *bp2 = dp;
D *dp2 = (D*)bp2;
```

Figure 3.10.: Pointer conversion.

is performed, the resulting pointer must not point to the virtual table pointer, like it did before, but to the following component. Therefore, such a cast involves increasing the address pointed to. Analogously, when casting a pointer to base class without virtual table down to a pointer to derived class with virtual table, the address must be decreased.

The special case of casting a null pointer must obey the following rule: a null pointer value is always converted to a null pointer value of the destination type. In that case, the address must not be incremented or decremented.

3.3. Features not implemented

We were not able to implement all features of EC++ into vbcc. This was due to the limited time available and the implemented features being more involved than initially expected. We implemented the features in order of importance to enable object–oriented programming. The implemented features are classes with static, virtual and usual methods, constructors, destructors, single–inheritance, access control, friends and overloading. Features that are still missing are references, user–defined operators, and some incompatibility issues. We estimate the completeness to be about 70%. In Sections 3.3.1 and 3.3.2 we present how we intend to implement references and user–defined operators, respectively.

3.3.1. References

Wikipedia defines a reference in the C++ language—whose behavior is not changed for EC++—as "a simple reference datatype that is less powerful but safer than the pointer type inherited from C" [46]. The functionality of reference types can be reduced to C equivalent functionality using pointer types. Hence, the intermediate language of vbcc does not need to be changed to handle references. However, additional type checking is required by the semantic analysis. For example, the semantics of references in assignments, as function arguments, as return types, etc. must be verified.

3.3.2. User–defined Operators

As stated in Section 2.2.1.1, EC++ allows to define the semantics of operators for user–defined types. How user–defined operators are compiled is similar to compiling methods. When a user–defined operator is defined, internally a global function is defined. The name of that function is mangled as specified in [9].

Operators can either be called explicitly (e. g., `a.operator=(b);`) or implicitly (e. g., `a=b;`). The explicit calls are handled analogous to method calls. Implicit operator calls must be detected during semantic analysis. Analogously to method calls, overloading and access rules must be verified.

4. Metrics

In this chapter, we will discuss different metrics to quantify the size and complexity of the specifications and the implementation of our EC++ compiler. Our goal for the specifications is to compare the two alternative EC++ specifications and to determine, how sensible and realizable a EC++ specification based on C is. For the implementation our objective is to estimate how expensive upgrading a C compiler to EC++ is and to approximate how much additional work is required for an upgrade to C++. As in the rest of this thesis, we will focus solely on language issues, and are not concerned with the runtime libraries.

4.1. Specification Analysis

In this section, we will use metrics to quantify and compare the physical extent of the two alternative EC++ specifications. Additionally, we will apply a metric to quantify the functional complexity of the specifications. We compare the sizes of the respective grammars in Section 4.1.1. In Section 4.1.2, we compare the textual extent of the official EC++ specification with the estimated extent of the alternative specification. Quantifying the functional complexity of a specification is a difficult task. We regarded alpha metrics as the most suitable metric for this, which we apply in Section 4.1.3.

4.1.1. Complexity of the Grammars

Recall from Section 2.2.2, that along with their specifications, each of the three languages (C, EC++, and C++) provide a grammar that accepts the associated language. For the alternative EC++ specification, we constructed a grammar in the same manner—but based on C rather than C++—(see Appendix B). The official EC++ grammar is specified based on C++, omitting certain rules or parts of rules. The alternative grammar is specified in the same way, but in difference to the C grammar. In addition to removing rules, it is also required to introduce new rules and to extend some.

To compare the official EC++ grammar with the alternative EC++ grammar, we analyze the number of rules and the number of changed rules. Also, we examine the extent of the grammars they are based on: C++ for the official grammar and C for the alternative grammar.

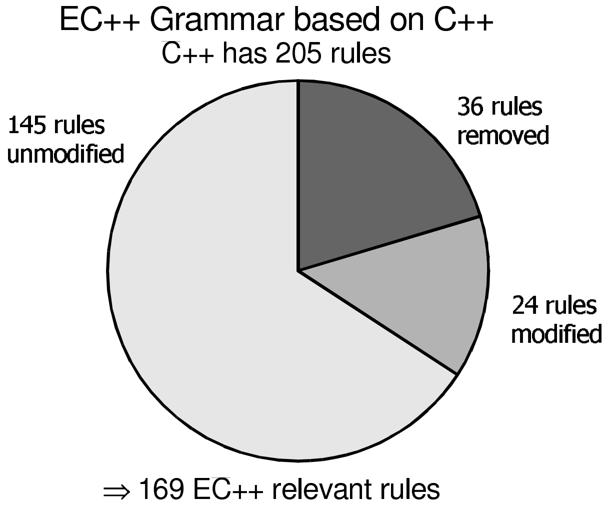


Figure 4.1.: Categorization of the rules of the official EC++ grammar.

4.1.1.1. Official EC++ grammar

The C++ grammar consists of 205 rules. The EC++ specification removes 36 of these completely, because these cover features that not present in EC++. Also because of the functionality removed from C++, 24 of the remaining rules need to be modified. What remains are 145 rules that are applicable for EC++ unaltered. Thus, defining the EC++ grammar in this way requires the 145 unmodified rules and the 24 adjusted rules, which is a total of 169 rules that are relevant for EC++, as depicted in Figure 4.1.

4.1.1.2. Alternative EC++ grammar

The C grammar is presented in the C specification as 136 rules. For the definition of the EC++ grammar based on C, new rules for the additional EC++ features must be added. 45 new rules are added to the C grammar. Most of these are exactly the same as in the official EC++ grammar or with small adjustments. To fit the new rules into the base grammar, and to address the incompatibilities between C and EC++, 27 rules need to be modified and 14 have to be removed. Relevant for this alternative EC++ grammar are the 96 unmodified C grammar rules, the 27 modified rules, and the 45 newly introduced rules. In sum, 167 rules are relevant for the alternative EC++ grammar. This composition is depicted in Figure 4.2.

4.1.1.3. Conclusion

We observe that both grammars have approximately the same extent when comparing the number of rules relevant for EC++. To measure the amount that must be changed compared to the base grammars, we analyze how many unchanged rules of the basis grammar the EC++ grammars contain. In the alternative grammar, due to many new

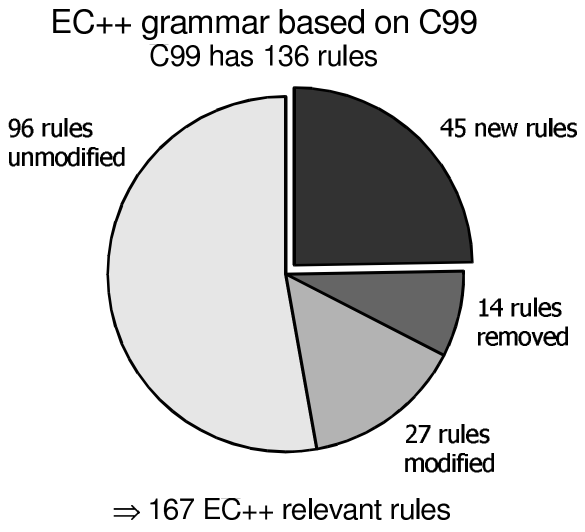


Figure 4.2.: Categorization of the rules of the C–based EC++ grammar.

rules, the unchanged C part makes about 57% of all EC++ relevant rules. For the official grammar, about 86% of the EC++ relevant rules are unchanged rules of the C++ grammar. Hence, the extent of both grammars is very much the same, but for the official version a bigger portion of its basis can be retained, whereas the alternative grammar needs to introduce a significant amount of changes and new rules. Note that a grammar for itself has only limited expressiveness and a definition of its semantics is necessary. The semantics are verified in the semantic analysis phase of the compiler.

4.1.2. Textual Extent

In this section, we will measure the textual extent of the official EC++ specification and estimate the extent of the alternative specification. The metric we use primarily is the number of paragraphs of the specifications. The average length of paragraphs, as well as the amount of information per paragraph are comparable in the C and the C++ specification. So, measuring the number of paragraphs is suitable. Also, the official EC++ specification often states to remove specific paragraphs, which can be measured directly. There are two things we will compare. First, how much of the base specification is still used in the EC++ specification based on it. Second, the sizes of the differential EC++ specifications.

4.1.2.1. Official EC++ Specification

The official EC++ specification is based on the C++ specification, which spans 1516 paragraphs. For EC++, 411 paragraphs are removed from C++, resulting in 1105 paragraphs relevant for EC++. About 20% of these need modifications, so about

80% of the EC++ relevant paragraphs are unchanged paragraphs from the C++ specification. The official EC++ specification itself consists of 234 short statements, e. g. “Because EC++ omits exceptions, a function never has a function try–block. <2nd par.>”.

4.1.2.2. Alternative EC++ Specification

The C specification on which the alternative EC++ specification is based contains 668 paragraphs. To add the additional functionality of EC++, the respective sections from the C++ specification must be added, which span 368 paragraphs, resulting in 1036 paragraphs relevant for EC++ so far. However, the information of the new functionality not in the respective C++ chapters is still missing. As said in Section 2.2.3, still many new paragraphs must be inserted.

4.1.2.3. Conclusion

It is difficult to compare the two EC++ specifications, since—due to creation of the alternative specification based on C being impractical—, we can only estimate the extent of the alternative one. However, as mentioned in Section 2.2.3, we estimate the extent of the alternative specification to be about 150–200 pages, magnitudes bigger than the official specification, which fits on 18 pages.

4.1.3. Alpha Metrics

Alpha metrics are used on a specific input text to yield an alpha value, which relates to the complexity of the text. Alpha metrics are described in [23, 24, 7]. We will describe how to calculate the alpha value and how to use it to compute the values for the C, the EC++, and the C++ specifications. To calculate it, we first have to convert the text—i. e. the specification—into a string of bits. To do that, each character is converted into a series of six bits using a code table. The bit string of the whole text is interpreted as a Brownian walk, where 1 means a step up and 0 means a step down. Figure 4.3 shows portions of the walks of the C, EC++, and C++ specifications.

The next step requires to determine the long–range correlations of the text. They are computed using Equations 4.1 and 4.2.

$$F^2(l) \equiv \overline{[\Delta y(l)]^2} - \overline{\Delta y(l)}^2 \tag{4.1}$$

$$\Delta y(l) \equiv y(l_0 + l) - y(l_0) \tag{4.2}$$

Where

F is the root of mean square fluctuation about the average of displacement,

l is the window size—the distance between two points on the walk,

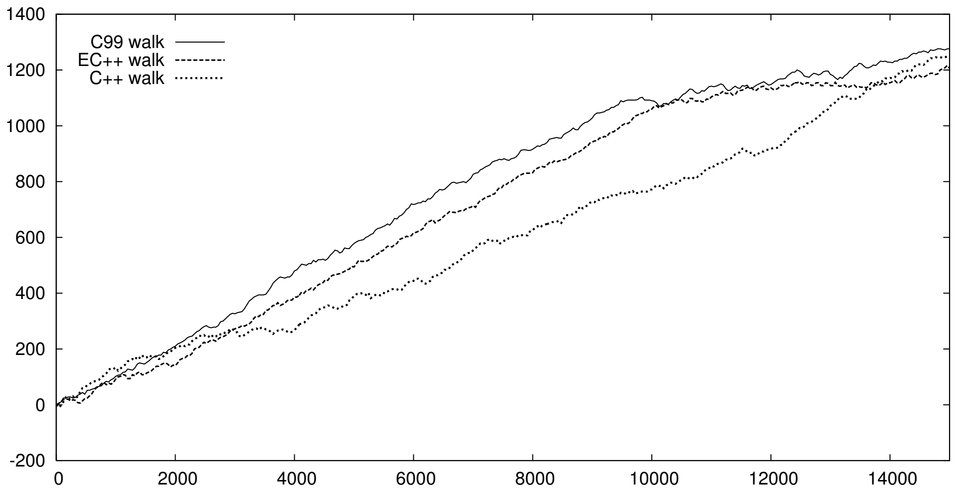


Figure 4.3.: Portions of the Brownian walks of the C, EC++, and C++ specification.

Specification	Alpha value
C++	0.660
C	0.648
EC++	0.609

Table 4.1.: Alpha values of the C++, C and EC++ specifications.

l_0 is the starting point of the walk,

$y(x)$ is the value of the walk at position x.

We calculate the values of $F(l)$ for all possible values of l. The highest l values must be discarded, because the window size is close to the sample size. The function $F(l)$ approximately describes a power law, thus

$$F(l) \approx l^{\alpha} \tag{4.3}$$

The power law can be observed most easily, when plotting $F(l)$ using double logarithmic axes. Then, the alpha value is the slope of the curve. If the string sequence is uncorrelated, then $\alpha \approx 0.5$. As an example, we applied the alpha metric on a text of random characters. Figure 4.4 shows $F(l)$ for that input. The resulting alpha value is very close to 0.5. This result is expected, because a random data sequence has small or no correlations.

We then calculated the alpha values for the C++ specification, the C specification, and the official EC++ specification (see Figures 4.5, 4.6, and 4.7). The resulting alpha values can be seen in the following Table 4.1.

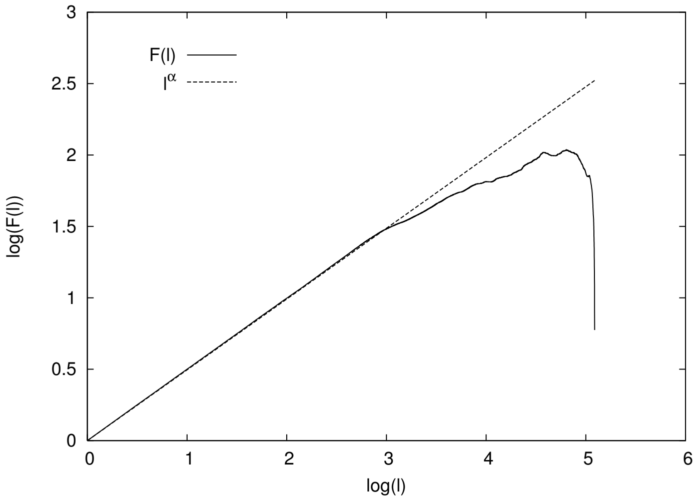


Figure 4.4.: Random input; $\alpha = 0.496$.

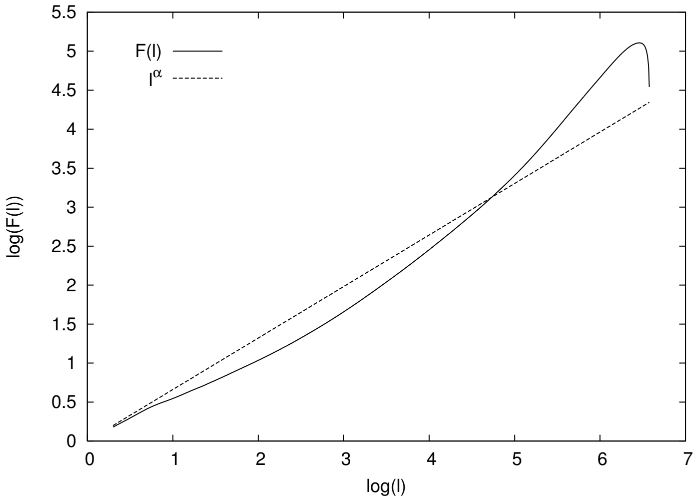


Figure 4.5.: C++ specification; $\alpha = 0.660$.

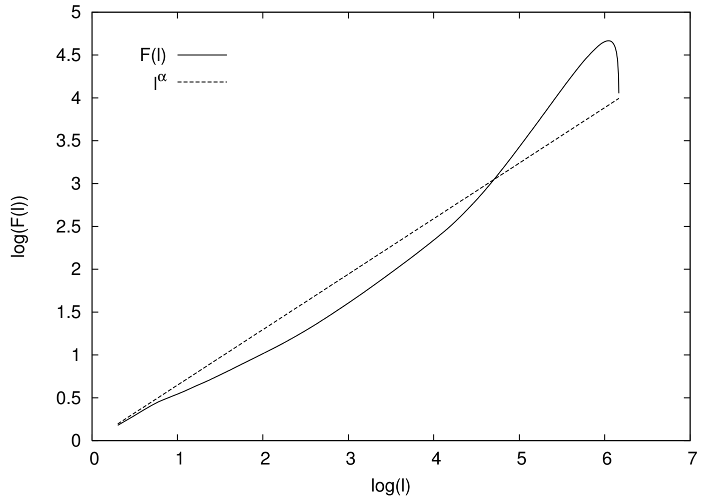


Figure 4.6.: C specification; $\alpha = 0.648$.

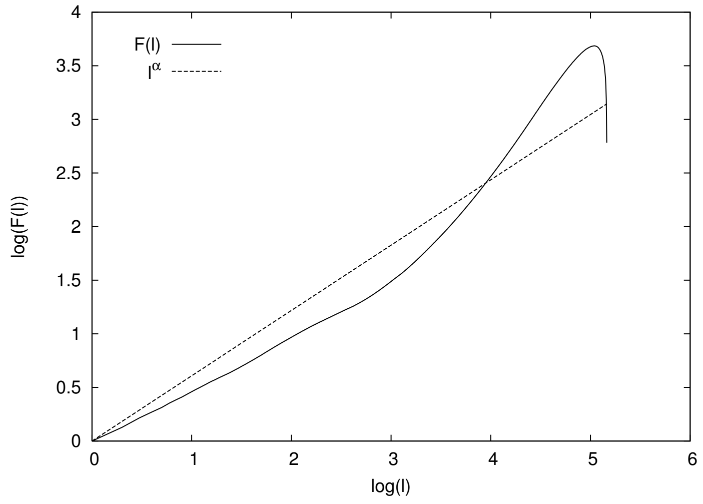


Figure 4.7.: Official EC++ specification; $\alpha = 0.609$.

Subject	Locations	Lines of code
Extensions	67	791
Modifications	21	31
New functions	33	1161

Table 4.2.: Lines of code added to vbcc.

The results are approximately the same for all three specifications. However, in [23], Kokol applies alpha metrics to different but functionally equivalent specifications and the resulting alpha values are quite different. He concludes "[...] to find out the reasons much more research will be needed". Hence, we cannot interpret from the results, that the three specifications are of comparable functional complexity.

4.2. Implementation Analysis

To estimate the extent and complexity of upgrading a C compiler to support EC++, we analyze our implementation regarding how many lines of code we needed to write (Section 4.2.1), and how the binary size of the compiler changed (Section 4.2.2). In Section 4.2.3, we further examine the complexity by calculating McCabe's cyclomatic complexity and observe, how it is affected by the upgrade. In Section 4.2.4, we compare the sizes of the GNU C and C++ compilers in an attempt to quantify the work that would be necessary to further upgrade an EC++ compiler to a full C++ compiler.

4.2.1. Extent of the Source Code

In this section, we will use the most traditional metric for measuring the size of source code. This metric is lines of code (LOC). LOCs are non-blank, non-comment lines in the text of the source code. Thus, before counting we stripped the code of comments and empty lines. We will evaluate the size of vbcc with and without EC++ support. Also, we will categorize the changes applied to the vbcc source code. The three categories we used are extensions, modifications, and new functions.

- An extension is code that extends the original functionality.
- A modification is code where the original functionality of vbcc had to be altered. These are mostly one–liners that create a branch.
- New functions that have been implemented for EC++ support.

Table 4.2 shows the number of lines of code added for each category. The locations column contains how many different locations needed editing for the extensions and modifications, and number of new functions. So, in total 1983 LOCs were introduced for the EC++ functionality. Since we estimate the functionality implemented to be

GCC call	used `strip`	without EC++	with EC++	Relation
`gcc`	no	544 kB	581 kB	+7%
`gcc -O3`	no	510 kB	540 kB	+6%
`gcc -Os`	no	379 kB	401 kB	+6%
`gcc`	yes	508 kB	544 kB	+7%
`gcc -O3`	yes	476 kB	504 kB	+6%
`gcc -Os`	yes	344 kB	364 kB	+6%

Table 4.3.: Sizes of the vbcc program with and without EC++ support and using different compiler options.

about 70%, the modifications for full EC++ will be about 2800 LOCs. The source code of vbcc with EC++ support and one back end has about 28000 LOCs. Thus, the EC++ extension makes up about 7% of the vbcc source code and about 10% with all features implemented.

4.2.2. Program Size

Another metric we applied is the evaluation of the compiled program sizes. For the compilation of vbcc we used the GCC compiler version 3.3.5 with different optimization settings. We optionally used the UNIX command `strip` on these executable to remove the symbol information not required for the execution. Figure 4.3 shows the results. The increment in size of vbcc with EC++ support is very much the same, for any compiler options used. The usage of `strip` does significantly affect the program size, however the relation of vbcc with and without EC++ support remains constant. Considering the EC++ support to be 70% complete, the estimated increase of program size with full EC++ support is about 9%–10%.

4.2.3. Cyclomatic Complexity

The evaluation of the cyclomatic complexity of a software module was introduced my McCabe in 1976 [25]. McCabe's cyclomatic complexity number (M) measures the number of paths through a software unit. It "is the most widely used member of a class of static software metrics" [8]. The complexity number correlates to the understandability, maintainability, and testability of the source code. Calculating M is a quick way to find functions that are potentially error–prone and that should be reorganized. McCabe suggests in [26] to avoid functions with M higher than 10 or to have a good reason for it being that high. In [8], Table 4.4 is presented, which gives further interpretation of the cyclomatic number M. The cyclomatic complexity is not a measure for the extent of software, but rather for its quality. We will use it

Cyclomatic Complexity Number (M)	Risk Evaluation
1–10	a simple program, without much risk
11–20	more complex, moderate risk
21–50	complex, high risk program
greater than 50	untestable program (very high risk)

Table 4.4.: Cyclomatic complexity [8].

function name	M without EC++	M with EC++
`type_expression2`	616	682
`var_declaration`	266	285
`declaration_specifiers`	153	262
`direct_declarator`	77	90

Table 4.5.: Cyclomatic numbers (M) of the vbcc functions most edited.

to see how complex the vbcc is without the EC++ extensions and compare it to its complexity with the EC++ functionality included.

Many of the vbcc functions have incredible high values of cyclomatic complexity. For example, 10 functions have a value higher than 150 and 49 functions have values higher than 50. Still, the vbcc is a very stable program, but extending functions with such a high complexity actually has shown to be error–prone and time–consuming. For comparison, we applied the complexity metric also on the GNU C compiler. The cyclomatic number here is even higher, 50 functions having an M value higher than 100. Thus, it deems not untypical to have some functions with very high cyclomatic numbers.

We will analyze the vbcc functions that were edited the most for EC++ support, and see how the modifications affect the value of M. Table 4.5 shows the functions that were editted the most for the EC++ functionality and their complexity with and without the EC++ functionality. Especially the function `declaration_specifiers` has increased much in complexity, and more modularization of the additional functionality has to be considered. Actually, all four of the functions in Table 4.5 should be reviewed to be restructured into less complex functions. Note that segmenting a function does not in general reduce the total complexity, but distributes it.

Secondly, we will apply the cyclomatic complexity metric to the functions newly introduced for the EC++ functionality. Table 4.6 shows the 33 new functions and their associated cyclomatic numbers. Most of these functions are of moderate or less complexity. However, at least the ones with a complexity higher than 20 should be reviewed and be segmented into less complex parts.

function name	M	function name	M
`ecpp_declarator`	53	`ecpp_find_ext_var`	7
`ecpp_ctor_init_list`	34	`ecpp_call_ctor`	6
`ecpp_mangle_arg`	30	`ecpp_clone_tree`	6
`ecpp_check_access`	26	`ecpp_access_specifier`	6
`ecpp_find_overloaded_func`	21	`ecpp_is_friend`	5
`ecpp_find_scope`	20	`ecpp_is_member_struct`	5
`ecpp_mangle_name`	19	`ecpp_gen_set_vtable`	5
`ecpp_transform_call`	17	`ecpp_call_dtor`	5
`ecpp_struct_offset`	16	`ecpp_dtor_epilog`	5
`ecpp_rank_arg_type`	14	`ecpp_free_init_list`	3
`ecpp_find_best_overloaded_func`	13	`ecpp_mangle_nested_identifier`	3
`ecpp_find_member`	13	`ecpp_add_friend`	2
`ecpp_find_struct`	13	`ecpp_dtor_prolog`	2
`ecpp_gen_default_dtor`	9	`ecpp_add_this_pointer`	2
`ecpp_gen_default_ctor`	9	`ecpp_auto_call_dtors`	2
`ecpp_linkage_specification`	9	`ecpp_auto_dtor`	1
`ecpp_find_var`	8		

Table 4.6.: Cyclomatic numbers (M) of the functions new for EC++.

4.2.4. Comparison of the GNU C and C++ Compilers

In this section, we compare the sizes of the GNU C and C++ compilers [14]. The GNU compiler collection (GCC) originally was only a C compiler, but by now it supports many additional language front ends, e. g. Ada, C++, Fortran, Java, Objective–C, and Objective–C++. The language front ends share a common internal structure and produce an abstract syntax tree, which is then fed to the intermediate code generator and optimizer to finally produce assembler code.

Table 4.7 shows the sizes of the C and C++ compiler programs of different versions (the name of C compiler program is `cc1`, that of the C++ program is `cc1plus`). The relation between the program sizes varies significantly depending on the version. One main reason of the ratio being smaller in newer versions is that modules common to both compilers (e. .g.: optimizer, back end) have grown in size. Thus, no clear conclusion about the relation between the C and C++ extent can be made. The absolute size increase does not change between the different versions. This implies, that the C++ front end has not been extended much in newer versions.

We also compared the sizes of the source code of the C and C++ compilers. We did not count comments or empty lines, just as we did for the source code in Section 4.2.1. Table 4.8 shows the results, which show the same tendencies as for the program sizes. The ratio between C and C++ code also decreases with new versions. The absolute

GCC version	`cc1` size	`cc1plus` size	Difference	Relation
4.0.0	4288 kB	4716 kB	+428 kB	+10%
3.3.5	3036 kB	3550 kB	+514 kB	+17%
2.95.4	1648 kB	2097 kB	+449 kB	+27%

Table 4.7.: Sizes of the GNU C and C++ compilers.

GCC version	C lines of code	C++ lines of code	Difference	Relation
4.0.0	311 k	374 k	+63 k	+20%
3.3.5	247 k	312 k	+65 k	+26%
2.95.4	166 k	229 k	+63 k	+38%

Table 4.8.: Sizes of the sources of the GNU C and C++ compilers.

differences—which are clearly dedicated to the C++ front end—do not change much across the different versions. This confirms the implication made about C's and C++'s program sizes: that the C++ front end has not been extended much in more recent versions.

4.2.5. Conclusion

By measuring the extent of the source code and the size of the program with and without EC++ support, we evaluated an increase by about 10% due to the EC++ upgrade. Our analysis of the sizes of the GCC compilers did not yield an estimate of the relative increase of the C++ compiler compared to the C compiler. However, we could conclude that the size of the C++ front end has remained very constant across different versions. The cyclomatic complexity analysis reveals functions which have a high complexity and thus should be reviewed for restructuring. Some functions of vbcc have a very high complexity which in some cases even increased through the upgrade to EC++. Those functions are candidates for future restructuring. Overall, we estimate the effort of upgrading a C compiler to support EC++ to be moderate.

5. Related Work

The basic concept we use to implement the EC++ extensions into a C compiler is not novel, but has been used in other compilers. To create a EC++ compiler by upgrading a C compiler however, is a new approach since all other EC++ compilers in the market are restricted C++ compilers. Designing languages and defining specifications based on existing languages and specifications—like C++ was designed based on C, and EC++ is specified based on C++—is a common practice. In this chapter we examine specific representatives of these categories.

The approach we use to implement the additional EC++ functionality, is to reduce the new EC++ constructs to equivalent C constructs. This is basically the same concept that Cfront [39] uses. Cfront was the original compiler for C++, which converted C++ to C. In contrast to Cfront, we do not generate C code, but perform the transformations from EC++ to C constructs in vbcc's internal data structures.

There are a limited number of EC++ compiler vendors in the market, including Green Hills Software [15], IAR Systems [16], and TASKING [35]. All offered compilers are C, C++, and EC++ compliant compilers. Thus, it is almost certain, that the EC++ compilers are restricted C++ compilers. To our knowledge, there is no EC++ compiler available, that is based on a C compiler. The only EC++ library implementation that is widely available is developed by Dinkumware [10]. This library is compatible with most popular C++ compilers. Any of the compatible C++ compilers can be used together with the EC++ library to compile EC++ applications. An interesting suggestion by Dinkumware is to mix the EC++ library with the standard template library (STL) in C++ applications.

The Java Micro Edition (Java ME) [34] by Sun Microsystems, is a set of specifications of Application Programming Interfaces (APIs) for embedded devices. These APIs are much smaller and have reduced functionality compared to the ones used for PC applications. Which of the specifications apply is defined by the application environment. One of the most prominent applications of Java ME technology are Java MIDlets [44]. MIDlets are frequently used in mobile phones to employ third party programs such as games. There are two API specifications that apply to MIDlets: one is device–independent (CLDC [31]), and the other is device–dependent (MIDP [30]). In contrast to EC++, the Java language is not restricted for MIDlets. EC++ applications are eventually deployed as programs consisting of machine code that runs directly on the target processor. On the other hand, MIDlets are compiled into byte–code which is run in a virtual machine on the mobile device. Both EC++ and MIDlets use smaller libraries designed for embedded systems. However, the MIDlet libraries are designed especially for mobile devices, whereas the EC++ library targets no specific category of embedded systems.

The Java Card technology [33] is also developed by Sun Microsystems, which enables small Java programs to be run on smart cards. Smart cards are "defined as any pocket–sized card with embedded integrated circuits" [47]. The Java Card specification [32] contains specifications for the virtual machine and the runtime environment for Java Card applications. Similar to EC++ being a restricted C++, a Java Card virtual machine supports only a subset of the Java language. Features not supported by the Java Card virtual machine include threads, cloning, `char`, `double`, `float`, and `long` types. A garbage collector is not required for a virtual machine, but can optionally be supported. EC++ and Java Card have in common, that they both use smaller libraries designed for embedded systems. However, in contrast to the EC++ library targeting no specific class of embedded systems, the Java Card libraries are designed especially for smart cards.

The development of programming languages by extension is quite common. EC++ and C++ extend C by object–oriented features, a trend which occurred to many languages. For example, Common Lisp [40] introduces the object–oriented paradigm to Lisp [42]. "Lisp is a family of computer programming languages" and "a number of dialects have existed over its history" [42]. Common Lisp is an ANSI standard, which was developed as an effort to standardize divergent Lisp dialects. By including the Common Lisp Object System (CLOS), object–oriented programming is supported in Common Lisp. The development history of Lisp and Common Lisp compared to the development of C, C++, and EC++ differ greatly. However, they have in common, that an older language—or family of languages—has been extended with object–oriented features.

Another example, where a language has been extended with object–oriented features is Object Pascal [45]. Object Pascal inserts object–oriented functionality into Pascal. The Delphi language [45] also evolved from Pascal and is closely related to Object Pascal. In 1993, the Technical Committee X3J9 created a draft for object–oriented extensions of Pascal [36]. This draft is based on the ISO specifications of Pascal [17] and Extended Pascal [19]. The draft never advanced to a standard, but inspired other languages, such as Delphi. To some extent, this approach is comparable to the one we intended for the specification of EC++ based on C. However, the biggest difference lies in the level of specifity on which the two definitions are done. The way in which the draft specifies the extensions is more abstract than we aimed for in the EC++ specification. Our goal was a specification in the same manner as the official EC++ specification, thus a complete list of all textual adjustments required on the basis. The draft for the Pascal extensions does not define how to textually adjust the bases, but rather states the modifications on a more functional level.

6. Future Work

In this chapter, we present the tasks for the future development for the topics of this thesis.

- **Implementation of missing features**
  The EC++ extension of the vbcc compiler is lacking the features specified in Section 3.3 (mainly references and user–defined operators). Obviously, the task of completing the EC++ features in order to reach EC++ compliance has high priority.

- **Additional optimizations**
  As stated in Section 3.2.2.3, dynamic function calls are costly compared to static function calls. This is primarily due to two additional pointer dereferencings being required. For further optimization, a global program analysis can be performed to find locations in which a dynamic function call actually always calls the same function. In these locations, the dynamic calls can safely be replaced with static calls [2].

- **Include a standard EC++ library**
  The official EC++ specification [13] defines the EC++ language, as well as the standard EC++ library. In this thesis and in the vbcc implementation we did not concern ourselves with the library. However, to fully support EC++, the compiler must be accompanied by a standard EC++ library. There are basically two options: the library can be custom–built, or a third party library is included.

- **Additional grammar metrics**
  In Section 4.1.1, we compared the extent of the official EC++ grammar [12], and the alternative EC++ grammar based on C (stated in Appendix B) by counting the number of rules. In [27], additional metrics for the comparison of grammars are specified. One of the main ideas in this paper is to interpret the grammars as programs. This is done by mapping the grammar elements onto elements of programs: "The procedures correspond to non–terminals, and procedure bodies are the right–hand sides of the production rules. The control primitives are the union and concatenation operations of context free grammars, which correspond to alternation and sequencing respectively" [27]. This way, metrics typically used for programs can be applied to grammars. For example,

the cyclomatic complexity metric which we applied on the implementation of vbcc in Section 4.2.3, can be applied on the alternative EC++ grammars.

7. Conclusion

EC++ is officially defined in terms of differences to C++. In this thesis, we took a different approach on EC++ and examined it in terms of differences to C. Defining EC++ based on C from a functional point of view resulted in a moderately sized list of incompatibilities and different functionality between the two languages. This set of issues proved to be a quite concise and descriptive definition of EC++ based C. Our attempt to specify EC++ as textual differences to the C specification turned out to be unreasonable both in terms of usability and producability. Thus, this low level of abstraction was inappropriate for an alternative definition. However, the differences viewed on a higher abstraction level—from a functional point of view—yielded a usable definition.

We also used this functional definition as a guideline for our implementation of a EC++ compiler by extending a C compiler. We applied metrics to measure the form of the specifications, by evaluating the textual extent. This provided an impression of the size and complexity of the specifications. Using metrics to measure the content of specifications in terms of complexity turned out to be of little use. Basically, the only statement that could be concluded from the results of the alpha metric is that the specifications are not just random texts. However, the ideas behind the alpha metric are interesting, but more research is required to improve their usability.

Implementing a EC++ compiler by extending a C compiler proved to be reasonable. The incompatibilities and different functionality required significant adjustment and extension of the front end. But the required changes and extensions were limited only to the front end of the compiler. The intermediate code, optimizations, and back end of the C compiler can be used for EC++ without change. All additional EC++ functionality could be reduced to equivalent C constructs. Implementing the EC++ features in this way proved to be realizable quite straightforward. Applying metrics to evaluate the source code and program size gave an impression of the extent of the effort. The evaluation of the cyclomatic complexity revealed hot–spots of the implementation where restructuring is appropriate to decrease complexity and improve maintainability and understandability.

A. Specification of EC++ based on ISO C99

6.5.2.2 Function calls

- Adjust[1] <1st par.>:
  The expression that denotes the called function shall have type lvalue that refers to a function (in which case the function–to–pointer standard conversion (4.3) is suppressed on the postfix expression), or pointer to function ~~returning `void` or returning an object type other than an array type.~~ or a member function call. For a member function call, the postfix expression shall be an implicit (9.3.1, 9.4) or explicit class member access (5.2.5) whose *id–expression* is a function member name, or a pointer–to–member expression (5.5) selecting a function member. The first expression in the postfix expression is then called the *object expression*, and the call is as a member of the object pointed to or referred to. In the case of an implicit class member access, the implied object is the one pointed to by `this`. [Note: a member function call of the form `f()` is interpreted as `(*this).f()` (see 9.3.1).] If a function or member function name is used, the name can be overloaded (clause 13), in which case the appropriate function shall be selected according to the rules in 13.3. The function called in a member function call is normally selected according to the static type of the object expression (clause 10), but if that function is `virtual` and is not specified using a *qualified–id* then the function actually called will be the final overrider (10.3) of the selected function in the dynamic type of the object expression [Note: the dynamic type is the type of the object pointed or referred to by the current value of the object expression. 12.7 describes the behavior of virtual function calls when the object expression refers to an object under construction or destruction.]

- Replace <2nd par.>with:
  If no declaration of the called function is visible from the scope of the call the program is ill–formed.

- Omit <6th par.>

[1]Text underlined is inserted into and text crossed out is removed from the C99 specification.

- Replace <7th par.>with:
  When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument. When a function is called, the parameters that have object type shall have completely–defined object type. [Note: this still allows a parameter to be a pointer or reference to an incomplete class type. However, it prevents a passed–by–value parameter to have an incomplete class type.] During the initialization of a parameter, an implementation may avoid the construction of extra temporaries by combining the conversions on the associated argument and/or the construction of temporaries with the initialization of the parameter (see 12.2). The lifetime of a parameter ends when the function in which it is defined returns. The initialization and destruction of each parameter occurs within the context of the calling function. [Example: the access of the constructor, conversion functions or destructor is checked at the point of call in the calling function.] The value of a function call is the value returned by the called function except in a virtual function call if the return type of the final overrider is different from the return type of the statically chosen function, the value returned from the final overrider is converted to the return type of the statically chosen function.

- Omit <9th par.>

- New paragraph:
  Calling a function through an expression whose function type has a language linkage that is different from the language linkage of the function type of the called function's definition is undefined (7.5).

- New paragraph:
  The type of the function call expression is the return type of the statically chosen function (i. e., ignoring the `virtual` keyword), even if the type of the function actually called is different. This type shall be a complete object type, a reference type or the type `void`.

- New paragraph:
  A function can be declared to accept fewer arguments (by declaring default arguments (8.3.6)) or more arguments (by using the ellipsis, `...` 8.3.5) than the number of parameters in the function definition (8.4). [Note: this implies that, except where the ellipsis (`...`) is used, a parameter is available for each argument.]

- New paragraph:
  Recursive calls are permitted, except to the function named `main` (3.6.1).

- New paragraph:
  A function call is an lvalue if and only if the result type is a reference.

B. Lexical Grammar

B.1. Lexical Grammar

B.1.1. Lexical Elements

token:
- *keyword*
- *identifier*
- *constant*
- *string–literal*
- *punctuator*

preprocessing–token:
- *header–name*
- *identifier*
- *pp–number*
- *character–constant*
- *string–literal*
- *punctuator*
- *boolean–literal*
- *each non–white–space character that cannot be one of the above*

B.1.2. Keywords

keyword: one of

auto	**break**	**case**	**char**
const	**continue**	**default**	**do**
double	**else**	**enum**	**extern**
float	**for**	**goto**	**if**
inline	**int**	**long**	**register**
~~**restrict**~~	**return**	**short**	**signed**
sizeof	**static**	**struct**	**switch**
typedef	**union**	**unsigned**	**void**
volatile	**while**	~~**_Bool**~~	~~**_Complex**~~
~~**_Imaginary**~~			
asm	**bool**	**catch**	**class**
const_cast	**delete**	**dynamic_cast**	**explicit**
export	**false**	**friend**	**mutable**
namespace	**new**	**operator**	**private**
protected	**public**	**reinterpret_cast**	**static_cast**
template	**this**	**throw**	**true**
try	**typeid**	**typename**	**using**
virtual	**wchar_t**		

B.1.3. Identifiers

identifier:
identifier–nondigit
identifier identifier–nondigit
identifier digit

identifier–nondigit:
nondigit
~~*universal–character–name*~~
~~*other implementation–defined characters*~~

nondigit: one of
_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

digit: one of
0 1 2 3 4 5 6 7 8 9

B.1.4. Universal Character Names

~~*universal–character–name*~~*:*
    ~~**\u** *hex–quad*~~
    ~~**\U** *hex–quad hex–quad*~~

~~*hex–quad*~~*:*
    ~~*hexadecimal–digit hexadecimal–digit*~~
    ~~*hexadecimal–digit hexadecimal–digit*~~

B.1.5. Constants

constant:
    integer–constant
    floating–constant
    enumeration–constant
    character–constant
    boolean–literal

integer–constant:
    decimal–constant integer–suffix$_{opt}$
    octal–constant integer–suffix$_{opt}$
    hexadecimal–constant integer–suffix$_{opt}$

decimal–constant:
    nonzero–digit
    decimal–constant digit

octal–constant:
    0
    octal–constant octal–digit

hexadecimal–constant:
    hexadecimal–prefix hexadecimal–digit
    hexadecimal–constant hexadecimal–digit

hexadecimal–prefix: one of:
    0x 0X

nonzero–digit: one of:
    1 2 3 4 5 6 7 8 9

octal–digit: one of:
    0 1 2 3 4 5 6 7

hexadecimal–digit: one of:

 `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$_{opt}$
 ~~*unsigned–suffix long–long–suffix*~~
 long–suffix unsigned–suffix$_{opt}$
 ~~*long–long–suffix unsigned–suffix*$_{opt}$~~

unsigned–suffix: one of:

 `u U`

long–suffix: one of:

 `l L`

~~*long–long–suffix: one of:*~~

 ~~**`ll LL`**~~

floating–constant:

 decimal–floating–constant
 ~~*hexadecimal–floating–constant*~~

decimal–floating–constant:

 fractional–constant exponent–part$_{opt}$ *floating–suffix*$_{opt}$
 digit–sequence exponent–part floating–suffix$_{opt}$

~~*hexadecimal–floating–constant:*~~

 ~~*hexadecimal–prefix hexadecimal–fractional–constant*~~
   ~~*binary–exponent–part floating–suffix*$_{opt}$~~
 ~~*hexadecimal–prefix hexadecimal–digit–sequence*~~
   ~~*binary–exponent–part floating–suffix*$_{opt}$~~

fractional–constant:

 digit–sequence$_{opt}$ **`.`** *digit–sequence*
 digit–sequence **`.`**

exponent–part:

 `e` *sign*$_{opt}$ *digit–sequence*
 `E` *sign*$_{opt}$ *digit–sequence*

sign: one of

 `+ -`

digit–sequence:
 digit
 digit–sequence digit

~~*hexadecimal–fractional–constant*~~*:*
 ~~*hexadecimal–digit–sequence*$_{opt}$ **.** *hexadecimal–digit–sequence*~~
 ~~*hexadecimal–digit–sequence* **.**~~

~~*binary–exponent–part*~~*:*
 ~~**p** *sign*$_{opt}$ *digit–sequence*~~
 ~~**P** *sign*$_{opt}$ *digit–sequence*~~

~~*hexadecimal–digit–sequence*~~*:*
 ~~*hexadecimal–digit*~~
 ~~*hexadecimal–digit–sequence hexadecimal–digit*~~

floating–suffix: one of:
 f l F L

enumeration–constant:
 identifier

character–constant:
 ' *c–char–sequence* **'**
 L' *c–char–sequence* **'**

c–char–sequence:
 c–char
 c–char–sequence c–char

c–char:
 any member of the source character set except the single–quote **'** *, back-slash* \ *, or new–line character escape–sequence*

escape–sequence:
 simple–escape–sequence
 octal–escape–sequence
 hexadecimal–escape–sequence
 ~~*universal–character–name*~~

simple–escape–sequence: one of
 **\\' \\" \\? **
 \\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

boolean–literal:
  true
  false

B.1.6. String Literals

string–literal:
  " *s–char–sequence*$_{opt}$ **"**
  L" *s–char–sequence*$_{opt}$ **"**

s–char–sequence:
  s–char
  s–char–sequence s–char

s–char:
  any member of the source character set except the double–quote **"** *, back-slash* \ *, or new–line character*
  escape–sequence

B.1.7. Punctuators

punctuator: one of

[	**]**	**(**	**)**	**{**	**}**	**.**	**->**
++	**--**	**&**	*****	**+**	**-**	**˜**	**!**
/	**%**	**«**	**»**	**<**	**>**	**<=**	**>=**
==	**!=**	**ˆ**	**|**	**&**	**&**	**||**	**?**
:	**;**	**...**	**=**	***=**	**/=**	**%=**	**+=**
-=	**«=**	**»=**	**&=**	**ˆ=**	**|=**	**,**	**#**
##	**<:**	**:>**	**<%**	**%>**	**%:**	**%:%:**	

::	**.***	**->***	**new**	**delete**	**and**
and_eq	**bitand**	**bitor**	**compl**	**not**	**not_eq**
or	**or_eq**	**xor**	**xor_eq**		

B.1.8. Header Names

header–name:
 < *h–char–sequence* **>**
 " *q–char–sequence* **"**

h–char–sequence:
 h–char
 h–char–sequence h–char

h–char:
 any member of the source character set except the new–line character and **>**

q–char–sequence:
 q–char
 q–char–sequence q–char

q–char:
 any member of the source character set except the new–line character and **"** “

B.1.9. Preprocessing Numbers

pp–number:
 digit
 . *digit*
 pp–number digit
 pp–number identifier–nondigit
 pp–number **e** *sign*
 pp–number **E** *sign*
 pp–number **p** *sign*
 pp–number **P** *sign*
 pp–number **.**

B.2. Phrase Structure Grammar

B.2.1. Expressions

primary–expression:
 ~~*identifier*~~
 constant
 string–literal
 (*expression* **)**
 this
 id–expression

id–expression:
 unqualified–id
 qualified–id

unqualified–id:
 identifier
 operator–function–id
 conversion–function–id
 ~ *class–name*

qualified–id:
 :: $_{opt}$ *nested–name–specifier unqualified–id*
 :: *identifier*

postfix–expression:
 primary–expression
 postfix–expression **[** *expression* **]**
 postfix–expression **(** *argument–expression–list*$_{opt}$ **)**
 postfix–expression **.** *identifier*
 postfix–expression **->** *identifier*
 postfix–expression **++**
 postfix–expression **--**
 ~~**(** *type–name* **)** **{** *initializer–list* **}**~~
 ~~**(** *type–name* **)** **{** *initializer–list* **,**~~ **}**
 type–specifier **(** *expression*$_{opt}$ **)**

argument–expression–list:
 assignment–expression
 argument–expression–list **,** *assignment–expression*

unary–expression:
- *postfix–expression*
- **++** *unary–expression*
- **--** *unary–expression*
- *unary–operator cast–expression*
- **sizeof** *unary–expression*
- **sizeof (** *type–name* **)**
- *new–expression*
- *delete–expression*

unary–operator: one of
- **& * + - ~ !**

new–expression:
- **::** $_{opt}$ **new** *new–placement*$_{opt}$ *new–type–id new–initializer*$_{opt}$
- **::** $_{opt}$ **new** *new–placement*$_{opt}$ **(** *type–id* **)** *new–initializer*$_{opt}$

new–placement:
- **(** *expression–list* **)**

new–type–id:
- *type–specifier–seq new–declarator*$_{opt}$

new–declarator:
- *ptr–operator new–declarator*$_{opt}$
- *direct–new–declarator*

direct–new–declarator:
- **[** *expression* **]**
- *direct–new–declarator* **[** *constant–expression* **]**

new–initializer:
- **(** *expression–list*$_{opt}$ **)**

delete–expression:
- **::** $_{opt}$ **delete** *cast–expression*
- **::** $_{opt}$ **delete [ ]** *cast–expression*

cast–expression:
- *unary–expression*
- **(** *type–name* **)** *cast–expression*

pm–expression:
cast–expression
pm–expression **.*** *cast–expression*
pm–expression **->*** *cast–expression*

multiplicative–expression:
cast–expression
multiplicative–expression ***** ~~*cast–expression*~~ *pm–expression*
multiplicative–expression **/** ~~*cast–expression*~~ *pm–expression*
multiplicative–expression **%** ~~*cast–expression*~~ *pm–expression*

additive–expression:
multiplicative–expression
additive–expression **+** *multiplicative–expression*
additive–expression **-** *multiplicative–expression*

shift–expression:
additive–expression
shift–expression **«** *additive–expression*
shift–expression **»** *additive–expression*

relational–expression:
shift–expression
relational–expression **<** *shift–expression*
relational–expression **>** *shift–expression*
relational–expression **<=** *shift–expression*
relational–expression **>=** *shift–expression*

equality–expression:
relational–expression
equality–expression **==** *relational–expression*
equality–expression **!=** *relational–expression*

AND–expression:
equality–expression
AND–expression **&** *equality–expression*

exclusive–OR–expression:
AND–expression
exclusive–OR–expression **ˆ** *AND–expression*

inclusive–OR–expression:
exclusive–OR–expression
inclusive–OR–expression **|** *exclusive–OR–expression*

logical–AND–expression:
  inclusive–OR–expression
  logical–AND–expression **&&** *inclusive–OR–expression*

logical–OR–expression:
  logical–AND–expression
  logical–OR–expression **||** *logical–AND–expression*

conditional–expression:
  logical–OR–expression
  logical–OR–expression **?** *expression* **:** *conditional–expression*

assignment–expression:
  conditional–expression
  unary–expression assignment–operator assignment–expression

assignment–operator: one of
  = *= /= %= += -= «= »= &= ˆ= |=

expression:
  assignment–expression
  expression **,** *assignment–expression*

constant–expression:
  conditional–expression

B.2.2. Declarations

declaration:
  declaration–specifiers init–declarator–list$_{opt}$ **;**
  <u>*linkage–specification*</u>
  <u>*asm–definition*</u>

declaration–specifiers:
  storage–class–specifier declaration–specifiers$_{opt}$
  type–specifier declaration–specifiers$_{opt}$
  type–qualifier declaration–specifiers$_{opt}$
  function–specifier declaration–specifiers$_{opt}$

init–declarator–list:
  init–declarator
  init–declarator–list **,** *init–declarator*

init–declarator:
  declarator
  declarator ~~**=**~~ *initializer*

storage–class–specifier:
  typedef
  extern
  static
  auto
  register
  friend

type–specifier:
  void
  char
  short
  int
  long
  float
  double
  signed
  unsigned
  ~~**_Bool**~~
  ~~**_Complex**~~
  ~~**_Imaginary**~~
  ~~*struct–or–union–specifier*~~
  enum–specifier
  ~~*typedef–name*~~
  bool
  :: $_{opt}$ *nested–name–specifier*$_{opt}$ *class–enum–typedef–name*
  class–specifier
  elaborated–type–specifier

type–specifier–seq:
  type–specifier type–specifier–seq$_{opt}$

~~*struct–or–union–specifier:*~~
  ~~*struct–or–union identifier*$_{opt}$ **{** *struct–declaration–list* **}**~~
  ~~*struct–or–union identifier*~~

~~*struct–or–union*~~*:*
  ~~**struct**~~
  ~~**union**~~

~~*struct–declaration–list*~~*:*
  ~~*struct–declaration*~~
  ~~*struct–declaration–list struct–declaration*~~

~~*struct–declaration*~~*:*
  ~~*specifier–qualifier–list struct–declarator–list* **;**~~

class–enum–typedef–name:
  class–name
  enum–name
  typedef–name

elaborated–type–specifier:
  class–key **::** $_{opt}$ *nested–name–specifier*$_{opt}$ *identifier*
  enum **::** $_{opt}$ *nested–name–specifier*$_{opt}$ *identifier*

specifier–qualifier–list:
  type–specifier specifier–qualifier–list$_{opt}$
  type–qualifier specifier–qualifier–list$_{opt}$

struct–declarator–list:
  struct–declarator
  struct–declarator–list **,** *struct–declarator*

struct–declarator:
  declarator
  declarator$_{opt}$ **:** *constant–expression*

enum–specifier:
  enum *identifier*$_{opt}$ **{** *enumerator–list* **}**
  ~~**enum** *identifier*$_{opt}$ **{** *enumerator–list* **,** **}**~~
  ~~**enum** *identifier*~~
  enum *identifier*$_{opt}$ **{ }**

enumerator–list:
  enumerator
  enumerator–list **,** *enumerator*

enumerator:
  enumeration–constant
  enumeration–constant **=** *constant–expression*

type–qualifier:
  const
  ~~**restrict**~~
  volatile

function–specifier:
  `inline`
  `virtual`
  `explicit`

asm–definition:
  `asm {` *string–literal* **`}`**

linkage–specification:
  `extern` *string–literal* **`{`** *declaration–list*$_{opt}$ **`}`** **`;`**
  `extern` *string–literal declaration*

declarator:
  pointer$_{opt}$ *direct–declarator*

direct–declarator:
  ~~*identifier*~~
  `(` *declarator* **`)`**
  ~~*direct–declarator* **`[`** *type–qualifier–list*$_{opt}$ *assignment–expression*$_{opt}$ **`]`**~~
  ~~*direct–declarator* **`[`** **`static`** *type–qualifier–list*$_{opt}$ *assignment–expression*~~
    ~~**`]`**~~
  ~~*direct–declarator* **`[`** *type–qualifier–list* **`static`** *assignment–expression* **`]`**~~
  ~~*direct–declarator* **`[`** *type–qualifier–list*$_{opt}$ **`*`** **`]`**~~
  ~~*direct–declarator* **`(`** *parameter–type–list* **`)`**~~
  ~~*direct–declarator* **`(`** *identifier–list*$_{opt}$ **`)`**~~
  declarator–id
  direct–declarator **`(`** *parameter–type–list*$_{opt}$ **`)`** *cv–qualifier–seq*$_{opt}$
  direct–declarator **`[`** *constant–expression*$_{opt}$ **`]`**

declarator–id:
  id–expression
  `::` $_{opt}$ *nested–name–specifier*$_{opt}$ *type–enum–typedef–name*

pointer:
  `*` *type–qualifier–list*$_{opt}$ *cv–qualifier–seq*$_{opt}$
  `*` *type–qualifier–list*$_{opt}$ *cv–qualifier–seq*$_{opt}$ *pointer*
  `&` *type–qualifier–list*$_{opt}$
  `&` *type–qualifier–list*$_{opt}$ *pointer*

type–qualifier–list:
  type–qualifier
  type–qualifier–list type–qualifier

parameter–type–list:
  parameter–list
  parameter–list **,** **...**
  parameter–list$_{opt}$ **...**

parameter–list:
  parameter–declaration
  parameter–list **,** *parameter–declaration*

parameter–declaration:
  declaration–specifiers declarator
  declaration–specifiers abstract–declarator$_{opt}$
  declaration–specifiers declarator **=** *assignment–expression*
  declaration–specifiers abstract–declarator$_{opt}$ **=** *assignment–expression*

identifier–list:
  identifier
  identifier–list **,** *identifier*

type–name:
  specifier–qualifier–list abstract–declarator$_{opt}$

abstract–declarator:
  pointer
  pointer$_{opt}$ *direct–abstract–declarator*

direct–abstract–declarator:
  (*abstract–declarator* **)**
  ~~*direct–abstract–declarator*$_{opt}$ **[** *assignment–expression*$_{opt}$ **]**~~
  ~~*direct–abstract–declarator*$_{opt}$ **[** ***** **]**~~
  ~~*direct–abstract–declarator*$_{opt}$ **(** *parameter–type–list*$_{opt}$ **)**~~
  direct–abstract–declarator$_{opt}$ **(** *parameter–type–list* **)** *cv–qualifier–seq*$_{opt}$
  direct–abstract–declarator$_{opt}$ **[** *constant–expression*$_{opt}$ **]**

cv–qualifier–seq:
  cv–qualifier cv–qualifier–seq$_{opt}$

cv–qualifier: Same as type–qualifier
  const
  volatile

typedef–name:
  identifier

initializer:
- **=** *assignment–expression*
- **=** **{** *initializer–list* **}**
- **=** **{** *initializer–list* **,** **}**
- **(** *expression–list* **)**

initializer–list:
- ~~*designation*~~$_{opt}$ *initializer*
- *initializer–list* **,** ~~*designation*~~$_{opt}$ *initializer*

~~*designation*~~*:*
- ~~*designator–list* **=**~~

~~*designator–list*~~*:*
- ~~*designator*~~
- ~~*designator–list designator*~~

~~*designator*~~*:*
- ~~**[** *constant–expression* **]**~~
- ~~**.** *identifier*~~

B.2.3. Statements

statement:
- *labeled–statement*
- *compound–statement*
- *expression–statement*
- *selection–statement*
- *iteration–statement*
- *jump–statement*

labeled–statement:
- *identifier* **:** *statement*
- **case** *constant–expression* **:** *statement*
- **default** **:** *statement*

compound–statement:
- **{** *block–item–list*$_{opt}$ **}**

block–item–list:
- *block–item*
- *block–item–list block–item*

block–item:
    declaration
    statement

expression–statement:
    expression$_{opt}$ **;**

condition:
    expression
    type–specifier–seq declarator **=** *assignment–expression*

selection–statement:
    if (~~*expression*~~ *condition* **)** *statement*
    if (~~*expression*~~ *condition* **)** *statement* **else** *statement*
    switch (~~*expression*~~ *condition* **)** *statement*

iteration–statement:
    while (~~*expression*~~ *condition* **)** *statement*
    do *statement* **while (** ~~*expression*~~ *condition* **) ;**
    for (~~*expression*$_{opt}$~~ *condition*$_{opt}$ **;** ~~*expression*$_{opt}$~~ *condition*$_{opt}$ **;**
        expression$_{opt}$ **)** *statement*
    for (*declaration* ~~*expression*$_{opt}$~~ *condition*$_{opt}$ **;** *expression*$_{opt}$ **)** *statement*

jump–statement:
    goto *identifier* **;**
    continue ;
    break ;
    return *expression*$_{opt}$ **;**

B.2.4. External Definitions

translation–unit:
    external–declaration
    translation–unit external–declaration

external–declaration:
    function–definition
    declaration

function–definition:
    ~~*declaration–specifiers declarator declaration–list*$_{opt}$ *compound–statement*~~
    declaration–specifiers$_{opt}$ *declarator ctor–initializer*$_{opt}$ *compound–statement*

declaration–list:
  declaration
  declaration–list declaration

B.2.5. Preprocessing Directives

preprocessing–file:
  group$_{opt}$

group:
  group–part
  group group–part

group–part:
  if–section
  control–line
  text–line
  `#` *non–directive*

if–section:
  if–group elif–groups$_{opt}$ *else–group*$_{opt}$ *endif–line*

if–group:
  `# if` *constant–expression new–line group*$_{opt}$
  `# ifdef` *identifier new–line group*$_{opt}$
  `# ifndef` *identifier new–line group*$_{opt}$

elif–groups:
  elif–group
  elif–groups elif–group

elif–group:
  `# elif` *constant–expression new–line group*$_{opt}$

else–group:
  `# else` *new–line group*$_{opt}$

endif–line:
  `# endif` *new–line*

control–line:
  `#` **`include`** *pp–tokens new–line*
  `#` **`define`** *identifier replacement–list new–line*
  `#` **`define`** *identifier lparen identifier–list*$_{opt}$ **`)`** *replacement–list new–line*
  `#` **`define`** *identifier lparen* **`...`** **`)`** *replacement–list new–line*
  `#` **`define`** *identifier lparen identifier–list* **`,`** **`...`** **`)`** *replacement–list*
    new–line
  `#` **`undef`** *identifier new–line*
  `#` **`line`** *pp–tokens new–line*
  `#` **`error`** *pp–tokens*$_{opt}$ *new–line*
  `#` **`pragma`** *pp–tokens*$_{opt}$ *new–line*
  `#` *new–line*

text–line:
  pp–tokens$_{opt}$ *new–line*

non–directive:
  pp–tokens new–line

lparen:
  a **`(`** *character not immediately preceded by white–space*

replacement–list:
  pp–tokens$_{opt}$

pp–tokens:
  preprocessing–token
  pp–tokens preprocessing–token

new–line:
  the new–line character

B.2.6. Classes

class–name:
  identifier

class–specifier:
  class–head **`{`** *member–specification*$_{opt}$ **`}`**

class–head:
  class–key identifier$_{opt}$ *base–clause*$_{opt}$
  class–key nested–name–specifier identifier base–clause$_{opt}$

class–key:
  `class`
  `struct`
  `union`

member–specification:
  member–declaration member–specification$_{opt}$
  access–specifier `:` *member–specification*$_{opt}$

member–declaration:
  decl–specifier–seq$_{opt}$ *member–declarator–list*$_{opt}$ `;`
  function–definition `;` $_{opt}$
  `::` $_{opt}$ *nested–name–specifier unqualified–id* `;`

member–declarator–list:
  member–declarator
  member–declarator–list `,` *member–declarator*

member–declarator:
  declarator
  declarator pure–specifier
  declarator constant–initializer
  identifier$_{opt}$ `:` *constant–expression*

pure–specifier:
  `= 0`

constant–initializer:
  `=` *constant–expression*

B.2.7. Derived Classes

base–clause:
  `:` *base–specifier–list*

base–specifier–list:
  base–specifier

base–specifier:
  `::` $_{opt}$ *nested–name–specifier*$_{opt}$ *class–name*
  access–specifier `::` $_{opt}$ *nested–name–specifier*$_{opt}$ *class–name*

access–specifier:
    `private`
    `protected`
    `public`

B.2.8. Special Member Functions

conversion–function–id:
    `operator` *conversion–type–id*

conversion–type–id:
    declaration–specifiers conversion–declarator$_{opt}$

conversion–declarator:
    pointer conversion–declarator$_{opt}$

ctor–initializer:
    `:` *mem–initializer–list*

mem–initializer–list:
    mem–initializer
    mem–initializer **`,`** *mem–initializer–list*

mem–initializer:
    mem–initializer–id **`(`** *argument–expression–list*$_{opt}$ **`)`**

mem–initializer–id:
    `::` $_{opt}$ *nested–name–specifier*$_{opt}$ *class–name*
    identifier

B.2.9. Overloading

operator–function–id:
    `operator` *operator*

operator: one of

```
new   delete   new[]   delete[]
+     -     *     /     %     ^     &     |     ~
!     =     <     >     +=    -=    *=    /=    %=
^=    &=    |=    «     »     »=    «=    ==    !=
<=    >=    &&    ||    ++    --    ,     ->*   ->
()    []
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison–Wesley, Reading, Massachusetts, USA, 1986.

[2] Andrew W. Appel. *Modern Compiler Implentation in Java.* Cambridge University Press, Cambridge, UK, 2002.

[3] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, 1960.

[4] Volker Barthelmann. vbcc – portable ISO C compiler, 2002. `http://www.compilers.de/vbcc.html`.

[5] Grady Booch. *Object Solutions: Managing the Object–Oriented Project.* Addison–Wesley, Reading, Massachusetts, USA, 1996.

[6] Benjamin M. Brosgol. TCOLAda and the "middle end"of the PQCC Ada compiler. In *SIGPLAN '80: Proceeding of the ACM-SIGPLAN symposium on Ada programming language*, pages 101–112, New York, NY, USA, 1980. ACM Press.

[7] Ana Isabel Cardoso, Rui Gustavo Crespo, and Peter Kokol. Assessing software structure by entropy and information density. *SIGSOFT Softw. Eng. Notes*, 29(2):2–2, 2004.

[8] Carnegie Mellon Software Engineering Institute. Cyclomatic Complexity, 2000. `http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html`.

[9] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ ABI (Revision: 1.86), 2005. `http://www.codesourcery.com/cxx-abi/abi.html`.

[10] Dinkumware, Ltd. Dinkum Compleat Libraries, 2006. `http://www.dinkumware.com/manuals/`.

[11] Embedded C++ Technical Committee. Rationale for the Embedded C++ specification, 1998. `http://www.caravan.net/ec2plus/rationale.html`.

[12] Embedded C++ Technical Committee. Annex A (informative): Grammar summary, Version WP-AM-003, 1999. `http://www.caravan.net/ec2plus/gram_efd.html`.

[13] Embedded C++ Technical Committee. The Embedded C++ specification, Version WP-AM-003, 1999. `http://www.caravan.net/ec2plus/spec.html`.

[14] Free Software Foundation. The GNU Compiler Collection, 2006. `http://gcc.gnu.org/`.

[15] Green Hills Software, Inc. Green Hills Optimizing C/C++/EC++ Compilers, 2004. `http://www.ghs.com/download/datasheets/c++_ec++compiler.pdf`.

[16] IAR Systems. Extended Embedded C++, 2006. `http://www.iar.com/`.

[17] International Organization for Standardization. ISO/IEC 7185:1990: Programming language Pascal, 1990.

[18] International Organization for Standardization. ISO/IEC 9899:1990: Programming languages — C, 1990.

[19] International Organization for Standardization. ISO/IEC 10206:1991: Programming language Extended Pascal, 1991.

[20] International Organization for Standardization. ISO/IEC 14882:1998: Programming languages — C++, 1998.

[21] International Organization for Standardization. ISO/IEC 9899:1999: Programming languages — C, 1999.

[22] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1978.

[23] Peter Kokol. Measuring formal specification with α–metric. *SIGSOFT Softw. Eng. Notes*, 24(1):80–81, 1999.

[24] Peter Kokol, Vili Podgorelec, Henri Habrias, and Nassim Hadj Rabia. The complexity of formal specifications–assessments by α–metric. *SIGPLAN Not.*, 34(6):84–88, 1999.

[25] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[26] Thomas J. McCabe and Arthur H. Watson. Software Complexity. *Crosstalk, Journal of Defense Software Engineering*, 7(12):5–9, 1994.

[27] J. Power and B. Malloy. Metric-based analysis of context-free grammars, 2000.

[28] A. Romanovsky, J. Xu, and B. Randell. Exception Handling in Object-Oriented Real–Time Distributed Systems. *In Proceedings of the 1st IEEE International Symposium on Object–Oriented Real-time Distributed Computing (ISORC '98)*, pages 32–42, 1998.

[29] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, Massachusetts, 1997.

[30] Sun Microsystems, Inc. Mobile Information Device Profile Specification, Version 2.0, 2002.

[31] Sun Microsystems, Inc. Connected Limited Device Configuration (CLDC) Specification, Version 1.1, 2003.

[32] Sun Microsystems, Inc. Java CardTM Specification, Version 2.2.1, 2003.

[33] Sun Microsystems, Inc. Java Card Technology Overview, 2006. `http://java.sun.com/products/javacard/overview.html`.

[34] Sun Microsystems, Inc. Java ME Technologies, 2006. `http://java.sun.com/javame/technologies/index.jsp`.

[35] TASKING, Altium Limited. Embedded C++ compiler technology, 2006. `http://www.altium.com/tasking/resources/technologies/compilers/ecpp/`.

[36] Technical Committee X3J9. Object–Oriented Extensions to Pascal, 1993. `http://www.pascal-central.com/OOE-stds.html`.

[37] David R. Tribble. Incompatibilities Between ISO C and ISO C++, 2001. `http://david.tribble.com/text/cdiffs.htm`.

[38] Jim Turley. The Two Percent Solution, 2002. `http://www.embedded.com/showArticle.jhtml?articleID=9900861`.

[39] Wikipedia. Cfront, 2006. `http://en.wikipedia.org/wiki/Cfront`.

[40] Wikipedia. Common Lisp, 2006. `http://en.wikipedia.org/wiki/Common_Lisp`.

[41] Wikipedia. Embedded system, 2006. `http://en.wikipedia.org/wiki/Embedded_systems`.

[42] Wikipedia. Lisp programming language, 2006. `http://en.wikipedia.org/wiki/Lisp_programming_language`.

[43] Wikipedia. Metaprogramming, 2006. `http://en.wikipedia.org/wiki/Meta_programming`.

[44] Wikipedia. MIDlet, 2006. `http://en.wikipedia.org/wiki/MIDlet`.

[45] Wikipedia. Object Pascal, 2006. `http://en.wikipedia.org/wiki/Object_Pascal`.

[46] Wikipedia. Reference (C++), 2006. `http://en.wikipedia.org/wiki/Reference\_\%28C\%2B\%2B\%29`.

[47] Wikipedia. Smart card, 2006. `http://en.wikipedia.org/wiki/Smart_cards`.

[48] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau*. Springer–Verlag, Berlin, Germany, 1997.

[49] World Semiconductor Trade Statistics. WSTS Semiconductor Market Forecast Autumn 2005, 2005. `http://www.wsts.org`.