ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

C+- : Γλώσσα για εκμάθηση προγραμματισμού, βασισμένη στις C και C++

Κωνσταντίνος Δ. Φερλές

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2012

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

C+- : Γλώσσα για εκμάθηση προγραμματισμού, βασισμένη στις C και C++

Κωνσταντίνος Δ. Φερλές
Α.Μ.: 1115200800151

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Αναπληρωτής Καθηγητής

ΠΕΡΙΛΗΨΗ

Αντικείμενο της παρούσας εργασίας είναι ο ορισμός και η υλοποίηση μιας γλώσσας προγραμματισμού, με εκπαιδευτικό χαρακτήρα, η οποία θα είναι καθαρό υποσύνολο της C++. Σκοπός είναι να δημιουργηθεί μια γλώσσα η οποία θα είναι κατάλληλη για εκπαίδευση αρχαρίων στον προγραμματισμό σε μαθήματα προπτυχιακού επιπέδου. Η C+- θα πρέπει να διατηρεί τα περισσότερα πλεονεκτήματα που έχει η C++ σε σχέση με την C, θα πρέπει να επιτρέπει τις πιο συνηθισμένες χρήσεις της standard template library (STL), αλλά θα πρέπει να απαγορεύει τις περισσότερες συντακτικές ασάφειες της C++ όπως επίσης δυσνόητα, προχωρημένου επιπέδου χαρακτηριστικά της που είναι πιθανόν να χρησιμοποιηθούν εσφαλμένα από αρχάριους χρήστες ή είναι αχρείαστα γι αυτούς.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Γλώσσες Προγραμματισμού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: εκμάθηση προγραμματισμού, C, C++, αρχάριοι χρήστες, χρήση πρότυπης βιβλιοθήκης

ABSTRACT

The purpose of this project is to define and implement a programming language for education. The language will be a pure subset of C++. The intent is to create a language suitable for low-level undergraduate education. The C+- language should maintain most of the advantages of C++ over C, it should enable most common uses of the standard template library (STL), yet it should disallow most syntactic ambiguities of C++, as well as obscure, advanced features that are likely to be misused by novice users or are unnecessary to them.

SUBJECT AREA: Programming Languages

KEYWORDS: learning programming, C, C++, novice users, standard library

To my parents, Eleni & Dimitris

Acknowledgements

First, I would like to thank my supervisor, Mr. Yannis Smaragdakis for giving such a challenging project as my undergraduate thesis. I am also grateful for the constant support and encouragement throughout the last and a half year that I am working on this project.

Second, I would like to thank my family and friends for all the support and advices throughout my undergraduate studies.

Last but not least, I would like to thank my aunt Katerina for saving me ten years ago by being in the right place the right moment.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

PROLOGUE

This project has been developed since July of 2011 in the University of Athens at the department of Informatics and Telecommunications as my undergraduate thesis. However we will continue the design and implementation of the project after graduation as a research project.

1. Introduction

Choosing a language to teach programming in an introductory course has always been a controversial issue. The design of an introductory language has several dimensions, such as choosing the paradigm of the language (e.g. Imperative or Object Oriented) or even if the language is going to be a pseudo-language or a programming language that is being used in the the real world. In the following, we propose a new programming language for introductory Computer Science (CS) education. The guide for our design decisions has been the consideration of what a freshman undergraduate in Computer Science (CS) finds difficult to cope with.

High and low level languages both have disadvantages, so choosing one kind over the other always sacrifices important CS education elements. Teaching a high level language, probably object oriented, such as Java, has the benefit of teaching abstraction in a structured way (via modules or classes). It also allows writing quick and easy programs as the language provides a variety of built-in data structures. At the same time it moves the programmer away from the machine, so that there is no awareness of memory management, computer architecture and other crucial issues. On the other hand teaching an imperative low-level language, like C [1], develops a programmer's intuition for all the aforementioned subjects, but has the disadvantage of a lack of built-in data abstraction that can help novice programmers build more complex structures and programs in general. Moreover it is noticeable that students with one of the above programming backgrounds have trouble adjusting to the other. For example, students with a Java-like background often experience problems in understanding concepts such as pointers and memory management (allocation, deallocation), whereas students with a C-like background often have difficulty using interfaces of generic abstractions (e.g., container data structures in a library).

Trying to find the right balance between these two directions, we observe that C++ [2][3] offers an interesting combination of low-level programming model and high-level abstraction, but it also has a lot of drawbacks stemming from it's complexity as a language. C++, as a superset of C, has all the advantages of a low-level language yet offers object-oriented abstraction in the form of classes. Furthermore, because of the standard template library (STL), C++ can be used by a novice user to write straightforward and simple programs. Yet, because of the language's advanced features (i.e. operator overloading, user defined templates), C++ programs are very difficult to parse, both visually, by the programmer, and for language processing, by the compiler.

Based on the above considerations, we decided that we need something in the middle, namely, a language that is a superset of C (syntactically) and provides features of C++ that are useful for novice programmers to write easily more complex programs.

Our new language, C+- (pronounced “C-plus-minus”) is a pure subset of C++. Every program in C+- is a syntactically correct C++ program. This means that the students have the chance to learn a real world programming language. At the same time, the language is more disciplined than full C++: several advanced mechanisms are removed and ambiguities in either static or dynamic semantics are settled so that error prone programs are avoided [4].

Since C+- is a subset of C++, it can be implemented as merely a front-end compiler. Its responsibilities will be the parsing and the semantic check of the program, with back-end compilation being delegated to a regular C++ compiler (selectable by the user). The

main complexity of the C+- implementation consists of its type system and its special-purpose provisions for dealing with regular C++ libraries. Compatibility is paramount and our goal is for our implementation to work with a variety of operating systems, different utility setups, different back-end C++ compilers and possibly even different IDEs.

In the next chapters we describe the C+- syntax and type system, how our implementation deals with complications such as C, C++ header files, the C preprocessor, the use of STL, etc.

2. Overall Language Design

In this chapter we describe the main design of our language. Since C+- is predominantly a superset of C , our discussion mostly focuses on features that originate from C++, either adopted or omitted in C+-. C+- actually extends C with classes as well as other C++ conveniences, such as namespaces, generic data structures, etc. We first present the main design axis of our language (section 2.1). Subsequently, we discuss which C standard our implementation follows, pointing out where our syntax slightly differs (section 2.2). Then we introduce the subset of C++ our language supports, and also explain the reasons behind rejecting some advanced features (section 2.3). Finally we describe the C+- type system (section 2.4).

2.1 Overview of Language

C+- is largely a superset of ANSI C [1] and subset of C++ [2][3]. It aims to maintain the low-level nature of C but enhance it with central features of C++. Specifically, C+- adds to C:

1. Classes, much like in C++. C+- is a multiple-inheritance object-oriented language, with only subtype-inheritance (“public” inheritance in the C++ parlance).
2. Namespaces, for better management of identifiers.
3. The ability to use large portions of the C++ standard library, such as container classes, file and stream management, etc. In C++, such library functionality is implemented using templates, yet C+- does not support templates. Therefore support for these features (to the extent that we expect will be helpful to a novice programmer) is built into the language.

More generally, however, C+- makes scores of smaller design changes over both C++ and, occasionally, ANSI C, in order to produce a language appropriate for novice programmers. Such changes include several C++-like extensions to C, but also restrictions of ANSI C for better parsing, the abolishment of C++ “references”, implicit conversions, operator overloading, and much more. We discuss these topics in the next sections.

2.2 ANSI C & Syntactic Differences

The requirement for compatibility with a variety of C++ compilers (as back-end compilers), led us to follow the syntax and rules of the ANSI C standard. In general, we handle declarations, statements and expressions in fashion identical to ANSI C. For example, we allow complex types such as pointers to functions, functions returning pointers to functions, arrays of pointers to functions, etc. But we disallow, by not supporting the syntax, common C extensions that are outside the ANSI C standard (e.g., nested functions, gnu's __attribute__ [10], __stdcall from visual C++ [11], etc.). This is because compilers do not support the same set of extensions or they implement differently a certain extension.

C+- does restrict the ANSI C standard in minor ways. These exceptions are due either to technical considerations for the implementation or to the desire to prevent the user from employing certain obscure patterns. So we continue in this section by

demonstrating the syntactic differences of our language, also providing the reasons for rejecting certain parts of the ANSI C syntax.

First of all, we simplify the declarations when there is a struct or union definition. After a struct (or union) definition, ANSI C allows declarations of objects, that are records/instances of that struct (fig 2.1). It also allows to declare storage class specifiers (e.g., `static`, `auto`) and cv-qualifiers (i.e., `const` and `volatile`) for the objects (fig 2.2).

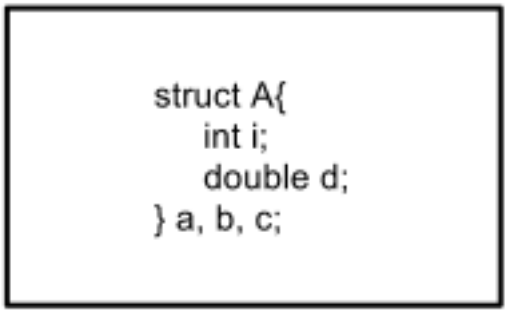


Figure 2.1 : Struct definition and object declarations

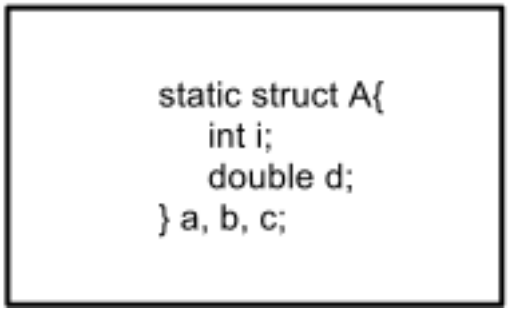


Figure 2.2 : Declaring static specifier for the objects

We disallow the syntax in figure 2.2 for two reasons. First, the definition is hard to read: it is easy to misread the `static` specifier as an attribute of the class and not the objects. To make matters worse, there are languages, such as Java, where `static` for a class and `static` for an object have different meanings. So a user familiar with these meanings can easily get confused while parsing the above example. Since one of our goals is for C+- programs to be parsed easily visually, disallowing this syntax makes sense. The second reason is that the ANSI C syntax complicates not only visual parsing but also machine parsing. Supporting the syntax requires more complex lookahead and parsing rules, and even produces an appreciably larger parser, for the parser generator we use (antlr [7][8]).

The second ANSI C feature we remove is the typedef of an anonymous struct. In C, typedefs can contain any declaration, for example pointers, functions, pointers to functions, etc. This is also allowed when defining a struct or a union, as the next example demonstrates:

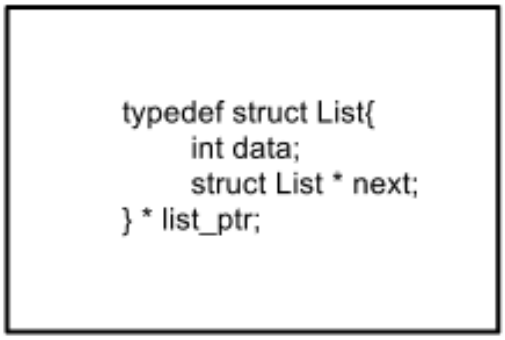


Figure 2.3 : typedef a pointer to a struct (while defining the struct)

Applying the same pattern with an anonymous struct, makes the creation of an object via structured means impossible. That is, object creation requires the name of the struct, so if the definition of the anonymous struct is used in, e.g., a typedef to a pointer

(as in fig. 2.4), this name is unavailable. However, the creation of an object can be achieved by manually allocating the space and then accessing it through a pointer, i.e., by mere casting of raw memory. But we want to prevent the user from writing such code, because it requires deep knowledge of both the compiler and the machine to handle various issues such as memory padding.

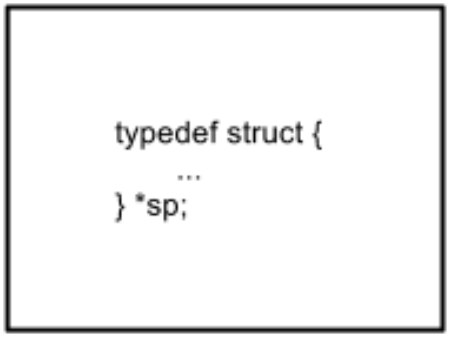


Figure 2.4 : typedef a pointer to an anonymous struct

The last point over which we deviate from the ANSI C standard is the handling of the return type in function declarations. In C, if the return type of a function is omitted, then it is implicitly declared to return integer. Although C+- allows the syntax, our implementation yields an error in this case. C++ also forbids this feature with few exceptions (e.g., declaration of main function, functions declared in an extern “C” block, etc.).

2.3 C++ Features

In this section we describe the C++ subset that our language supports. We keep the minimum subset of C++ that we consider to be easy and useful for a novice programmer. In general, we add to C support for namespaces and classes. So we also extend the definition of C structs and unions with useful elements such as methods, access specifiers for encapsulation, etc. We continue in this section by listing all these features, as well as all the C++ elements that we do not support.

Entirely removed C++ features

The most major element that we remove from C++ is the concept of a reference. Having two ways for passing parameters (i.e., pass by value and pass by reference) is confusing even for post-graduate students. To simplify our language we allow passing parameters only by value, but the parameters could be themselves pointers, thus allowing modification of the data they point-to.

The next element that we entirely remove from the language is user defined templates. It is a feature that even advanced programmers often cannot use properly. It can also lead to error prone programs, where the bugs depend on the use of the template. That is, certain bugs and compile time errors may be triggered only by certain instantiations of the template.

We also remove default arguments for both methods and functions. Supporting this feature alongside function/method overloading complicates the rules of finding the candidate function/method at call-site. To avoid the interference between these two

features, we keep only method overloading, as we demonstrate later in this section, because it is generally a more useful feature than default arguments. Furthermore, the absence of default arguments does not limit expressiveness, since all default values can be manually supplied at the call-site. On the other hand there is no workaround for method overloading and it is quite helpful while building a class' interface.

Classes

The C++ features for classes (also for structs and unions), that C+- supports are:

1. Access specifiers can be declared for fields and methods.
2. Constructors and use of the "`explicit`" keyword, to avoid implicit conversions.
3. Destructor, which can also be declared `virtual`.
4. Method overloading, almost as described in the C++ standard [3]. There is a subtle difference (see section 2.4)caused by a limitation of the C+- type system.
5. Methods can be declared `virtual` and with the implicit parameter (`this`) to be `const` and/or `volatile`.
6. Overriding virtual methods, much like the C++ standard specifies[3]. The same subtle difference asfor method overloading applies (see section 2.4).
7. Inner types are supported. That is, a class declaration can be a nested class definition or forward declaration, an enumeration definition, or a typedef.
8. Static methods and fields.
9. Public multiple inheritance. We want to encourage the user to write separate classes, which a new class can later inherit and combine their interfaces.
10. Initializer list in constructor.
11. Friend classes.
12. External class definition. That is, when there is a class forward declaration within a class the definition of the forward declared class can be outside the outer class (fig. 2.5).

We also support all the appropriate syntax for the aforementioned features, such as new and delete operators, the syntax of public inheritance, etc.

```
class Outer{

    class Inner;

};

class Outer::Inner : public Outer{
  //...
};
```

Figure 2.5 : External class definition.

Conversely, the features of C++ classes that C+- does ***not*** support are:

1. Copy constructors are not supported because there are no references. (Copy constructors, by definition, take a reference as a parameter).
2. Even though C+- supports method overloading, it does not support operator overloading. This is because overloading operators for a class makes both visual and mechanical parsing need a lot of context information.
3. Using declarations are not supported (fig. 2.6), because they introduce names only from base classes and the name lookup becomes complex.
4. Friend functions.
5. Pointers to fields, for example the next example is invalid:

   int A::* pi = NULL;

   where A is a class.
6. Private and protected inheritance are not supported, because both are syntactic variants for composition. That is, every private or protected “is-a” relation can be replaced by a “has-relation”. Furthermore, the rules for the derived class members' accessibility become more complex.
7. Virtual inheritance.

```
class A{
public:
    int x;
};

class B : public A{
    using A::x; //using declaration
};
```

Figure 2.6 : Using declaration

Namespaces

The features that we support are:

1. Method overloading, similar to class definitions.
2. Namespaces are open, as in C++: new members can be added at any point in the program and the full contents of the namespace can never be assumed known. This feature can be useful for programmers, but it is also necessary to support the std namespace, which is defined across several files.
3. Using directives (e.g., `using namespace std;`).

All other features described in the C++ standard [3], such as static namespaces or namespace aliasing are not permitted by C+-.

2.4 The C+- Type System

In this section we describe our type system and how it differs from the C++ type system. The C+- type system maintains the core of the C++ one in order to achieve compatibility with C++ compilers (as back-end compilers). However, we introduce some new rules to make our type system more disciplined. These new rules are mostly about type conversions (implicit and explicit) and type casts. Moreover, as our implementation is a front-end compiler, these rules use only static information. If our compiler were able to perform code transformations, we would be able to introduce more rules, but in this case our language would not have been compatible with C++.

Before we discuss the new rules, we briefly describe the C+- primitive types. C+- supports all the primitive types that a modern C++ compiler supports.

The list of all C+- primitive types is below:

- `void`
- `bool`
- `char`, `unsigned char`
- `short`, `unsigned short`
- `int`, `unsigned int`
- `long`, `unsigned long`
- `long long`, `unsigned long long`
- `float`
- `double`
- `long double`

As mentioned earlier, C+- imposes several restrictions (compared to ANSI C or C++) on type conversions. First, we limit implicit type conversions for integer-like primitive types. The only implicit type conversion that is allowed is when the target type is wider than the source type, because it is guaranteed that there will be no information loss. On the other hand when the target type is narrower than the source type, the conversion is valid only if there is an explicit type cast (fig. 2.7), because in this case execution of the same program may vary between implementations/compilers (due to information loss) [9]. We generally want to force the programmer to use type casts in a proper way, i.e., when there is a possibility that the source type is not compatible with the destination.

```
int i;
double d;
d = i;          //OK widening
i = d;          //error narrowing without type cast
i = (int) d;  //OK narrowing with type cast
```

Figure 2.7 : Widening vs Narrowing between primitives types

Figure 2.8 sums up all the explicit and implicit conversions for integer-like primitive types. The arrows in the graph are transitive and if there are two paths between two nodes (this is caused by the `long long` type only), the path with the implicit

conversions dominates, i.e., no explicit conversion is necessary. For the two primitive types (i.e., `void` and `bool`) that are not included in the graph, C+- applies the rules below:

1. C+- disallows declaration of fields and variables with void type, following both the ANSI C and the C++ standard. That is, if a field or variable is declared to be `void`, it is considered undeclared for the whole program.
2. For the bool type, C+- allows neither implicit nor explicit conversion from/to an integer-like type.

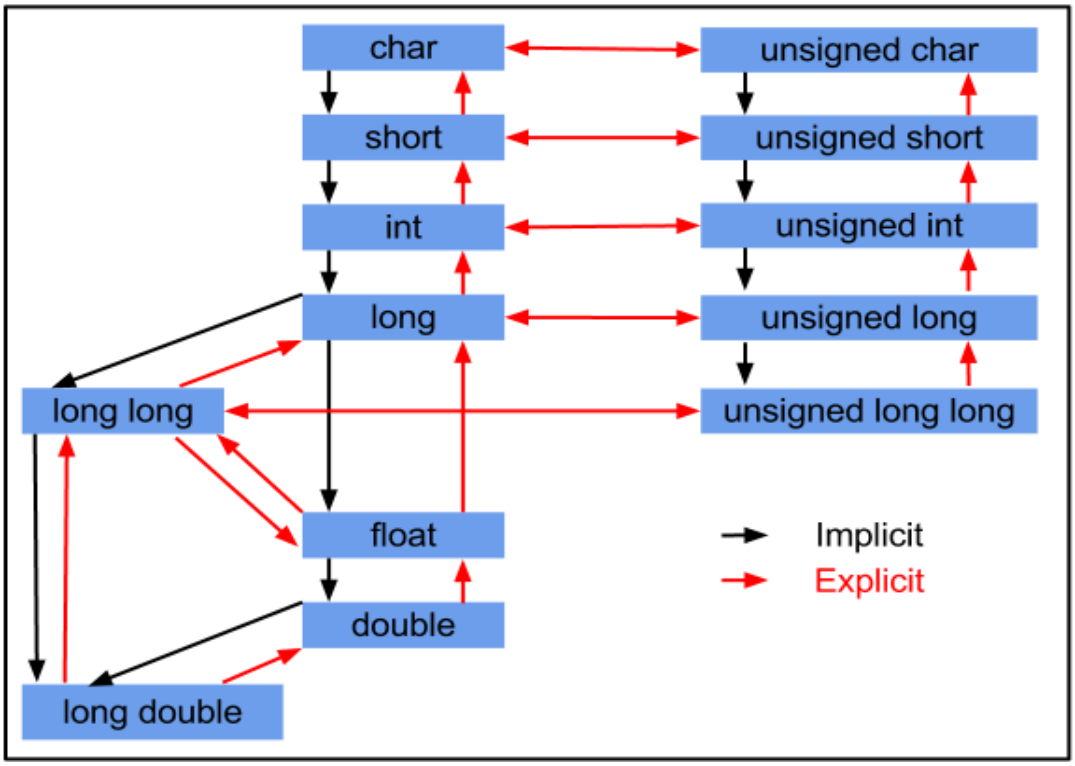


Figure 2.8 : Implicit and Explicit conversion for integer-like primitive types

Furthermore, by disallowing conversions between integers and booleans we can statically catch errors caused by misunderstandings of the C++ syntax [4]. For example, C++ allows the syntax below:

```
if( 1 <= x < 3){
  ...
}
```

Figure 2.9 : Implicit conversion between integer and boolean

At first glance, the condition in the above `if` seems to mean that “if x is greater or equal than 1 and smaller than 1”. But the actual meaning for this condition is: “evaluate `1<= x` and then check if the result is smaller than 3“. Since, in both C and C++, a boolean expression is equivalent to an integer (i.e., false is zero and true is everything non zero)

the type system allows the above program (however, some implementations may yield a warning).

Moreover, C+- differentiates the static semantics of type cast. Although we keep only the primitive cast syntax (i.e., ANSI C style and not `static_cast`, `reinterpret_cast`, etc. C++ operators), we do not allow type casts between arbitrary types. In more detail, C+- adds the following rules to the primitive cast syntax:

1. Conversions of pointers to a `const` type that omit the type's `const` specifier are not allowed even with a cast. This particular case is listed as undefined behavior in ANSI C and C++, and there are cases where even gcc and g++ execute differently a certain program (fig. 2.10). On the contrary, both implicit and explicit conversions from a non `const` to a `const` type is allowed.
2. No conversions between integers and pointers (or any non-pointer type and pointers) even with a cast. Nevertheless, we maintain `void*` as a universal pointer type. This is sufficient for dealing with functions (i.e., passing parameters and return types) from both the standard library and the user (e.g., already in K&R C, 2nd edition `malloc` returns `void*`)
3. For classes, even though we are using the primitive cast syntax, the semantics is `static_cast`. That is, one cannot cast from A* to B* if A is not a supertype of B (or a subtype, which is a trivial cast).

```
#include <stdio.h>

int main(){

        const int i = 3;

        int *pi = (int *) &i;
        (*pi)++;

        /*
         * compiling with gcc 4.6.3, prints i = 4;
         * compiling with g++ 4.6.3, prints i = 3;
         * in both cases no optimization flag is used;
         */
        printf("i = %d\n", i);
        printf("*pi = %d\n", *pi);

}
```

Figure 2.10 : Undefined behavior whit pointer to `const` type.

Finally, C+- treats arrays almost as pointers. Both ANSI C and C++ allow arbitrary constant expressions as dimension sizes in arrays declarations. A C/C++ compiler evaluates the constant expressions by performing constant propagation. Since our implementation does not perform any constant propagation, we only check the array's number of dimensions (i.e., we convert the array declaration to a pointer equivalent). This limitation has the following consequences:

1. For semantic errors for which array's dimensions size matter, we delegate the error reporting to the back-end compiler.

2. For function/method overloading and method overriding we are more limited than C++, because we convert the array declaration to a pointer (fig. 2.11).

```
class A {

        void foo(char [][3]);
        void foo(char [][4]);

};
```

Figure 2.11 : Overloadable for C++, but not for C+-

2.5 Declarations and Name Lookup

In this section we describe how we simplify the C++ rules for declarations inside namespaces and classes and how our rules formulate our general strategy for name lookup. C++ in general needs a lot of context information during parsing. Since our requirement is to keep syntactic ambiguities to a minimum level, we simplify the rules for declarations so both parsing and name lookup would be easier than C++. Before we discuss these rules and our name lookup strategy, we present some examples that demonstrate why parsing C++ is a complicated process.

The example that follows shows that the same piece of code has different meanings in different contexts. Consider the code in figure 2.12 inside a block, where expressions are allowed. In the case where "`A`" is a visible type name (e.g., a `class` name, a `typedef`, etc.) the code below is a declaration ("`a`" is a pointer to "`A`"). On the other hand, if both "`A`" and "`a`", are integers for example (or any type that supports infix operator *), the code is a multiplication.

```
{
    //...
    A*a;
    //...
}
```

Figure 2.12 : According to context either a multiplication or a pointer declaration.

If there is no name conflict between a field or variable and a visible type, parsing for both previous cases is still straightforward. But in the case where there are name conflicts between a visible type and another element (e.g., field, local variable, etc.), parsing becomes more complicated both visually and mechanically. This can be shown by adding a couple of code lines to the previous example:

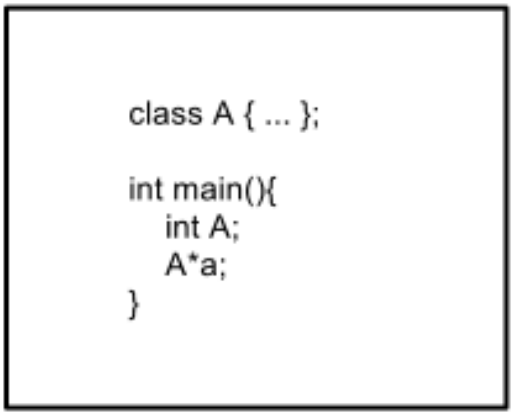


Figure 2.13 : Ambiguity: either pointer declaration or multiplication

Compiling the above code snippet, yields an error that "`a`" is undeclared. That is, C++ tries to parse "`A*a`" as an expression because "`A`" is a local variable. On the other hand, because "`a`" is not a visible variable and multiplications as statements are rare, "`A*a`" in figure 2.13 visually may be interpreted as a declaration. However, the above example can easily be disambiguated by explicitly referring to the global "`A`" (i.e., "`::A`").

It is obvious that even with very simple code, parsing can be very complicated. To make matters worse, there are cases, such as classes, where double declarations are allowed (i.e., two or more elements inside the scope can have the same name) and rules for shadowing must also be taken into consideration for parsing. To simplify declarations a bit we introduce the rules below:

1. Declarations in general cannot shadow an unambiguous visible type name (i.e., a `class`, `typedef`, `enum` or `namespace` name). So programs such as that in figure 2.13 are considered ill-formed in C+-. However, a type can shadow a visible type name, if it is in a base class or another namespace (fig. 2.15).
2. Within a namespace or a class, names for classes, namespaces, typedefs and enums (i.e., what names a type in general) are unique.

Rule 1 only applies to unambiguous visible type names. This restriction is in place in order to support independent development and later merging of namespaces/classes that contain conflicting definitions of identifiers. If a type name is already ambiguous within a scope (e.g., it is a nested type in two different base classes), a declaration can shadow this name (fig. 2.14).

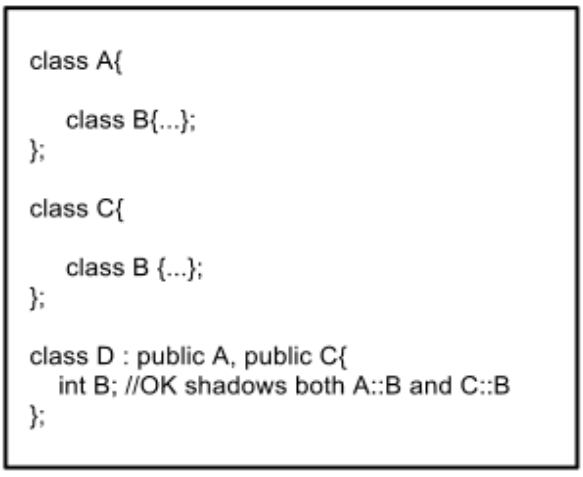


Figure 2.14 : Shadowing an ambiguous type name with a field in the derived class.

```
namespace N{
   class A { ... };
}

using namespace N;

class A { //OK shadows N::A

   class B{ ... };
};

class C : public A{

   class B { ... }; // OK shadows A::B
};
```

Figure 2.15 : Shadowing a visible type name with another type name

The only case in which C+- needs to check if there is conflict between a type name and a field is that below:

```
class A{
public:
   class B {
   public:
      class C{ ... };
   };
};

class C{
public:
   int B;
};

class D : public A, public C{

   B*b; //error: ambiguous reference
   class B* b1; //OK
   B::C c; //also OK
};
```

Figure 2.16 : Ambiguous reference for C+-

In figure 2.16, there is nothing in `class D` to shadow the identifier "`B`", therefore the first declaration is an ambiguous reference. For the other two declarations there is no ambiguity, because the syntax ensures that "`B`" refers only to a type name. It is worth mentioning that in classes C+- allows the use of fields and methods before their declarations. So, in the previous example if "B*b" was inside a block and there was a field "b" declared afterwards, the expression would be valid.

Taking into consideration the rules and the exception above, our general strategy for parsing ambiguous code that can be either a declaration or an expression is the below:

We first try to parse everything as a declaration, i.e. we try to interpret identifiers as types. So, in case we find a valid type name, we consider the rest as a list of declarators

(i.e., pointer declarators, function declarators, etc.). On the other hand, if there is no valid type name (or there is and it is either private or ambiguous) and an expression can be formed we consider the whole statement to be an expression. Then when we perform semantic checks for expressions we try to find unambiguous fields or local variables for all the identifiers. We need two phases here (i.e., first consider all expressions valid and then check if they are semantically correct), because in classes the use of a member may occur before its declaration.

3. The C+- Preprocessor

In this chapter we describe the preprocessor features that C+- supports. We general want to limit the C/C++ preprocessor because of its unhygienic macro system [6], which can lead to error prone programs. Since we want programmers to avoid common pitfalls, such a design decision makes sense. So, in this chapter we first list all the C preprocessor features we allow for users (section 3.1) and then we explain the reasons for rejecting macro expansion, providing simultaneously workarounds for most common uses (section 3.2).

3.1 Allowed C Preprocessor Features.

The C/C++ preprocessor features that C+- supports are the ones below:

1. `#define identifier`
2. `#ifdef identifier`
3. `#ifndef identifier`
4. `#endif identifier`
5. `#include <C/C++ System Header File>`
6. `#include "User's Header File"`

We support the features 1 – 4, because they are useful for conditional inclusion. So a user can create multiple configurations for compiling a program and/or avoid problems when there are cycles in file inclusion.

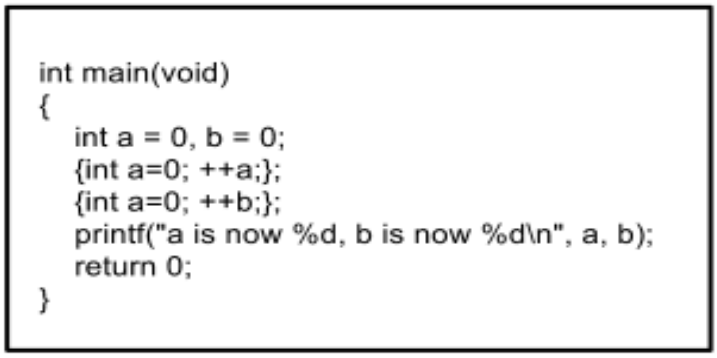
3.2 C/C++ Macros

In this section we describe the reasons for not having full macro support in C+-. The most serious reason is the unhygienic macro system of the C/C++ preprocessor [6]. It is possible for existing variable bindings to be hidden by variable bindings that are created during a macro's expansion. The next example illustrates the aforementioned problem (fig. 3.1 and 3.2).

```
#define INCI(i) {int a=0; ++i;}
int main(void)
{
    int a = 0, b = 0;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

Figure 3.1 : Unhygienic macro expansion pt. 1

Passing the code in figure 3.1 through the C/C++ preprocessor produces the code below:



```
int main(void)
{
   int a = 0, b = 0;
   {int a=0; ++a;};
   {int a=0; ++b;};
   printf("a is now %d, b is now %d\n", a, b);
   return 0;
}
```

Figure 3.2 : Unhygienic macro expansion pt. 2

It is obvious that this code will print “a is now 0, b is now 1”. Another pitfall is that a macros' parameters are not evaluated before expansion. For example, let's assume the next macro:

```
#define print_twice(i) printf(“%d %d\n”, i, i);
```

Then passing from the preprocessor this code: `print_twice(i++);`, the output is the one below:

```
printf(“%d %d\n”, i++, i++); //i is increased twice
```

Although C+- does not have full macro support there are workarounds for their common uses. For example, macros such as the one in figure 3.1 can be replaced by actual methods/functions. Moreover, when macros are being used to define constants (e.g., “`#define BUFF_SIZE 1024`”), they can be replaced by global constant integers or enumerations (for a group of constants).

4. Dealing With Standard Libraries

We continue in this chapter by discussing the subset of C and C++ standard library that C+- supports. Supporting any subset of these libraries and still maintaining compatibility with a variety of systems and C++ implementations is a challenge. In section 4.1 we discuss, from a technical point of view, how we achieve this compatibility, while in section 4.2 we describe the subset of the C standard library that we support, as well as the STL containers that C+- maintains.

4.1 C/C++ Preprocessor And Library Support.

It is a big challenge for a front-end compiler to be compatible with a variety of systems and C++ implementations mostly for technical reasons. The core of the problem is that supporting these libraries depends on a big amount of system header files. Moreover, these header files differ even for same implementations, because they are system and architecture specific. In addition, different C++ implementations have different syntaxes in order to support their own language extensions. For example, gnu compiler has the __attribute__ [10] syntax for calling conventions, string formats, etc., while visual C++ uses __stdcall [11] for calling conventions.

Since we do not want our parser to deal with all these extensions, we have to provide our own versions of these header files. System header files are split into two categories, those that we handle (semantically), and those we do not handle at all. Generally having an include file that we don't handle will result a parse error.

Among the rest, for include files that we handle (C, C++ and STL include files), we provide our, very cut-down versions of the same include files, with clean, simple headers and a small subset of the functionality. We include these using the standard preprocessor and parse the output normally. This implies that our implementation needs to supports preprocessor 's line markers [12][13], for correct reporting of line numbers in error messages over the preprocessed output. However, some C++ features must be handled syntactically and semantically through C+-. For example:

1. Instantiation of a template container, e.g., `vector<int> v;`
2. Applying operator “[ ]” to a vector, `v[1]`.
3. Input and output operators for streams (i.e., operators “<<” and “>>”).
4. C++ iterators (e.g., they support operator “→”, unary operator “*”, etc.).

To implement the logic above we use preprocessor' s -nostdinc flag, to prevent it from searching in system's path for include files, and -I flag to provide the path with our own versions of these files. Note that none of the above processing of include files affects target compilation, that is, the files given as input to the back-end compiler are the original files the user supplied.

4.2 C and C++ Libraries.

In this section we give an overview of the subset of the standard library that C+- supports. For ANSI C library we have no limitations for any header file, since we support in general all the appropriate syntax. On the other hand, we do not provide full support

of the C++ standard library, because our grammar cannot parse all the system header files. We continue in this section by first describing in more details our strategy about ANSI C standard library and second list the core of the C++ system header we support.

C+- by default supports only a subset of the ANSI C library that contains everything that we consider to be useful for a novice user. Nevertheless, header files can be either added or removed from this subset according to course's goals. So, C+- standard library is adjustable and for example can be extended to support very specific system calls that may be needed in a project.

For the C++ standard library we support only a few header files that they have reduced functionality and the syntax can be handled by our parser (as we described in section 4.1). These files are mostly about strings, files and streams manipulation, as well as STL containers. For the first category (strings, files and streams) we provide simplified header files (see appendix 1), since they only contain classes and all the supported methods and our parser can handle these cases (except from input and output operators, i.e., “<<” and “>>”, that need to be handled semantically).

In order to add STL containers we treat specially all the header files. Since we do not support syntax for templates our header files are simple C++ class (see appendix 1) and the template parameters are being handled by our implementation. In addition, we support only certain parameters for every container, since we want only the basic functionalities for the users and not to deal with parameters such as allocators, etc.

Next table summarizes all the containers and their parameters that C+- consider to be valid.

Table 1: Valid STL containers and supported template parameters.

STL container	Supported Template Parameters
vector	**T**: Type of the elements.
list	**T**: Type of the elements.
set	**Key:** Key Type. **Compare**: Comparison Class. (Optional)
multiset	**Key:** Key type. **Compare**: Comparison Class. (Optional)
map	**Key:** Key type. **T:** Type of the mapped value. **Compare:** Comparison Class. (Optional)
multimap	**Key:** Key type. **T:** Type of the mapped value. **Compare:** Comparison Class. (Optional)
bitset	**N:** Number of bits to contain

Moreover, our implementation does not perform template instantiation as C++ does, so our error messages are much smaller and more readable. That is, C+- keeps for every container its instantiation parameters and for every usage of the container checks only the types of the parameters. For example, consider the next example:

```
vector<int> v1;
vector<int> v2;
v1.push_back(1);
v2.push_back(2);
v2.insert(v2.begin(), v1.begin(), v2.end());
```

Figure 4.1 : Instantiation and use of a vector.

C+- does not only check the parameter of the `push_back` is compatible with `int`, which is the template instantiation parameter, but it also checks that the two last parameters of `insert`, which is a template function, are actually iterators (they are not template parameters, we manually put these constraints). Our implementation also yields an error when a container that needs a comparison operator and the instantiation parameter does not support it.

Finally, a lot of methods in the the C++ standard library have as parameters and/or return values references, while C+- does not support syntax for references. Since we treat those libraries much like keywords, we consider that these parameters and return values are plain values and not references, so they can be handled semantically.

5. Implementation Overview

In this chapter we briefly describe the outline of our implementation. To generate our parser we use “Antlr” parser generator [7][8]. Since C+- is a front-end compiler we only user antlr's features that do not perform code transformations. Error reporting is being handled either by antlr 's error recovery technique or manually by us (as embedded actions in antlr grammars).

The steps of our implementation are:

1. Check if the preprocessor features inside the user's input files are valid for C+-.
2. If the first step succeed, pass all the code files through the standard preprocessor (with C+- system header files though).
3. Perform semantic checks on preprocessor' s output files.
4. If there was no error in the above process, provide all the original user files, as long as the command line options to the back end compiler.

Finally there is an under development project, which can be found at github.

Github address: git://github.com/kferles/Cpm.git

Conclusion

To conclude in this project we presented C+-, a new programming language for education. C+- is a superset of ANSI C and a subset of C++, it is actually adds classes, namespaces and the C++ standard library to C. It has all the appropriate features that a novice user needs to write straightforward and easy programs. Our future goal is C+- to be used and evaluated by users. Such a feedback will give us all the appropriate information to alter features that still novice users find difficult to cope with, make error messages even more user friendly, etc. Eventually we would like C+- to be used as main language for both introductory and more advanced courses.

Abbreviations

ANSI	American National Standards Institute
CS	Computer Science
STL	Standard Template Library
Antlr	Another Tool for Language Recognition

Appendix 1

Here we present sample header files that are being used to add functionality from both ANSI C and C++ to C+-.

First we present the header file for C input/output operations, `stdio.h`

"stdio.h":

```
#ifndef _STDIO_H
#define _STDIO_H

struct FILE;
typedef unsigned int size_t;

/* The possibilities for the third argument to `fseek'.
   These values should not be changed.  */
#define SEEK_SET 0   /* Seek from beginning of file.  */
#define SEEK_CUR 1   /* Seek from current position.  */
#define SEEK_END 2   /* Seek from end of file.  */

extern struct FILE *stdin;         /* Standard input stream.  */
extern struct FILE *stdout;        /* Standard output stream.  */
extern struct FILE *stderr;        /* Standard error output stream.  */

int printf (const char * format, ...);

int sprintf (char * s, const char * format, ...);

int fprintf (FILE * stream, const char * format, ...);

int fclose (FILE * stream);

int fflush (FILE * stream);

FILE *fopen (const char * filename, const char * modes);

FILE *fdopen (int fd, const char * modes);

int scanf (const char * format, ...);
```

```
int sscanf (const char * s, const char * format, ...);

int fscanf (FILE * stream, const char * format, ...);

int fgetc (FILE * stream);

int getc (FILE * stream);

int getchar (void);

int fputc (int c, FILE * stream);

int putc (int c, FILE * stream);

int putchar (int c);

char *fgets (char * s, int n, FILE * stream);

char *gets (char * s);

int fputs (const char * s, FILE * stream);

int puts (const char * s);

int ungetc (int c, FILE * stream);

size_t fread (void *ptr, size_t size, size_t n, FILE * stream);

size_t fwrite (const void * ptr, size_t size, size_t n, FILE * s);

int fseek (FILE * stream, long int off, int whence);

long int ftell (FILE * stream);

void rewind (FILE * stream);

/* Clear the error and EOF indicators for STREAM.  */
extern void clearerr (FILE * stream);
```

```
/* Return the EOF indicator for STREAM.  */
extern int feof (FILE * stream);

/* Return the error indicator for STREAM.  */
extern int ferror (FILE * stream);

/* Print a message describing the meaning of the value of errno.
 */
extern void perror (const char * s);

#endif
```

Since our grammar recognizes all the above syntax no further support from our implementation is needed.

Next we present the file for supporting vector stl container.

```
"vector"
#ifndef __VECTOR__
#define __VECTOR__

#pragma GCC system_header

namespace std{

    typedef unsigned int size_t;

    class vector{

        public:

            typedef size_t  size_type;;

            vector(size_type n, const T value);

            vector(InputIterator first, InputIterator last);

            vector(const vector x);
```

```
~vector();

/*
 * Iterators
 */
iterator begin();

const_iterator begin() const;

iterator end();

const_iterator end() const;

reverse_iterator rbegin();

const_reverse_iterator rbegin() const;

reverse_iterator rend();

const_reverse_iterator rend() const;

/*
 * Capacity
 */
size_type size() const;

size_type max_size() const;

void resize(size_type sz, T c);

size_type capacity() const;

bool empty() const;

void reserve(size_type n);

/*
```

```
            * Element access
            */
           //operator [] will be handled from the syntax

           const T at(size_type n) const;

           T at(size_type n);

           T front();

           const T front() const;

           T back();

           const T back() const;

           /*
            * Modifiers
            */

           void assign(InputIterator first, InputIterator last);

           void assign(size_type n, const T u);

           void push_back(const T x);

           void pop_back();

           iterator insert(iterator position, const T x);

           void insert(iterator position, size_type n, const T x);

           void insert(itarator position, InputIterator first,
InputIterator last);

    };
```

```
}
#endif
```

Classes T, InputIterator (for the template functions), vector::iterator, vector::const_iterator are being handled by our implementation.

For the object-oriented input/output operations we provide simplified class that keep the basic functionality and we handle through our implementation the support of operators such as “<<” and “>>”. Finally the standard streams are just members of the istream and ostream classes:

```
istream cin;
ostream cout,cerr;
```

References

[1] Brian W. Kernighan and Dennis M. Ritchie, The C programming Language.

[2] Bjarne Stroustrup, The C++ Programming Language.

[3] International Standard ISO/IEC 14882, Programming Languages – C++, First edition

[4] http://www.horstmann.com/cpp/pitfalls.html [Accessed 14/10/2012]

[5] http://www.cplusplus.com [Accessed 14/10/2012]

[6] http://en.wikipedia.org/wiki/Hygienic_macro [Accessed 14/10/2012]

[7] Terence Parr, The Definitive ANTLR Reference Building Domain-Specific Languages.

[8] http://www.antlr.org [Accessed 14/10/2012]

[9] http://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html [Accessed 14/10/2012]

[10] http://www.unixwiz.net/techtips/gnu-c-attributes.html [Accessed 14/10/2012]

[11] http://msdn.microsoft.com/en-us/library/zxk0tw93(v=vs.80).aspx [Accessed 14/10/2012]

[12] http://msdn.microsoft.com/en-us/library/3sxhs2ty(v=vs.80).aspx [Accessed 14/10/2012]

[13] http://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html [Accessed 14/10/2012]