Checking C Declarations at Link Time [1] [2]

Diomidis D. Spinellis
Myrsinis 1
GR-145 62 Kifissia
Greece
dds@leon.nrcps.ariadne-t.gr

Abstract

In a separate compilation environment type checks across source files are difficult to implement, because the natural place to perform them, the linker, is rarely under the control of the compiler developer. This is typically handled either by programs that perform a global declaration and use consistency check, or by relying on the programmer to create and use a consistent set of headers. We present a solution to this problem based on encoding the identifiers with their types. Each function or variable identifier has an encoding of its type appended to its name. In this way type mismatches are caught at link time as undefined references. Multiple definitions of the same identifier with different types are handled by creating dummy variables.

Introduction

The ANSI C standard allows for translation units (source files) to be separately compiled, and linked at a later stage to produce an executable program (§2.1.1[3]). Consequently, a part of the translation process must be performed by the linker. The linker however, is rarely a part of a translator system. Most often, it is a tool provided with the operating system, as it contains knowledge of the formats of object, library and executable files, as well as the conventions used for implementing overlays and shared libraries. In some systems the ability to create program files is restricted to secure programs, i.e. the system linker. C language translator implementors can find themselves in a position of having to provide features that are not supported by the system linker

[1] *The Journal of C Language Translations*, 4(3):238–249, March 1993.

[2] This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work.

[3] All § references in this article refer to paragraphs in the ANSI C standard document [1].

[13, §3.1.2]. In this article we will concern ourselves with implementing cross-module declaration checking at link time. We will first present the problem and outline the various solutions to it, together with their drawbacks. Then, we will demonstrate how the type encoding of function and variable names with external linkage — an extension to a scheme used by some C++ implementations — can be used to provide declaration consistency checking at link time.

The Problem

We start the description of the problem with some definitions. A *declaration* of an object or function is used to specify its interpretation and attributes (§3.5). Furthermore, declarations can be used to define and reserve space for objects and define functions (§3.7). These declarations are called *definitions*. All objects and functions that are used in expressions must have been defined, and no more than one definition is allowed for any object or function with external linkage (§3.7).

As we mentioned in the introduction, the last part of a C program translation can (and often has to) be performed by the system linker. The problem with this approach is that conflicting declarations can not be checked across different modules. Consider the following two source files:

```
/* File 1 */

int f(int i)
{
        return i * 2;
}

/* File 2 */
#include <stdio.h>

int f(void);

void g(void)
{
        printf("%d\n", f());
}
```

In our system, both files compile[4] and link without any warning or error messages. The two files contain declarations[5] for the same function (`f`) that do not have compatible types. Specifically, in file 1 `f` is defined as `int f(int)` whereas in file 2 it is declared as `int f(void)`. According to ANSI C (§3.1.2.6) the behaviour of a program containing declarations that refer to the same object or function that do not have compatible types is undefined (in our system `g` prints 2 when called from

[4] Using the project GNU C compiler [12] version 2.2.2.

[5] The external definition of `f` in file 1 is also a declaration, according to §3.7.

`main`). The compiler is not violating the ANSI standard, since "ignoring the situation with unpredictable results" is permissible undefined behaviour (§1.6). However, a higher quality implementation should issue of a suitable diagnostic statement (such as "conflicting declarations for function f in file 1, file 2").

Possible Solutions

This problem has been addressed in various C development environments in many different ways.

Lint

In the Unix programming environment, a separate program, *lint*[6] [5] handles the problem of type checking across modules [6, p. 103] by examining the set of compilation units as a whole and — among other things — verifying declaration and use consistency across them. *Lint* can 'pre-compile' information from a set of compilation units comprising a library into a special file which can then be used to verify that the library has been used in a correct way by some other code, obviating the need to run through the its source text. Running *lint* on the two example files listed above (modified for the "classic" C supported by our system's *lint*) results in the following error message:

```
f: variable # of args.  f1.c(5)  ::  f2.c(8)
```

The solution is not very efficient as typically *lint* needs to examine all modules in order to determine whether the functions exported are used in a consistent way. It also depends on cooperation from the programmer in order to run *lint* and keep the *lint* libraries up to date.

Function Headers

In ANSI C the type compatibility of objects and functions across compilation units can be ensured by judicious use of header files. If all external declarations are placed in header files, and the header files are included at the beginning of the files where the objects are defined and used, then all conflicting declarations will be in the same *file scope*. Any ANSI conforming implementation should then issue a diagnostic message for declarations to the same object specifying incompatible types as directed by the constraints in §3.5. Standard header files containing declarations are provided for all library functions (§4.1.2). Under this approach our standard example would be coded as follows:

```
/* File 1 */
#include "defs.h"

int f(int i)
```

[6] Named after the "lint" it is supposed to remove from the program source.

```
{
        return i * 2;
}

/* File 2 */
#include <stdio.h>
#include "defs.h"

void g(void)
{
        printf("%d\n", f());
}

/* defs.h */
int f(void);
```

Compiling the two functions in our system generates the following error message:

```
f1.c:5: conflicting types for ‘f’
defs.h:1: previous declaration of ‘f’
```

The problem with this approach is the need to keep the headers consistent with both the source they refer to, and the compiled objects that depend on them. Special tools such as *mkdep* [8, mkdep(1)] can be used in conjunction with *make* [3] to achieve the desired effect. Their effectiveness depends on the cooperation, concentration and organisational powers of the programmer.

Checking C Declarations at Link Time

From the examples given above it should be clear that the natural time to check for declaration consistency across translation units is at program link time. At that point of the translation process, declaration consistency can be effectively policed without relying on user cooperation, or extra-lingual tools and approaches. In the following sections we will examine the requirements for correct linkage of translation units, and the basic idea behind function name encoding.

Requirements for a correct linkage

The process of linking associates symbol references to symbol definitions [9]. In a C translator implementation where the translation units are combined at link time, the following requirements should be met:

1. External object and function references must be resolved, by matching them with their definitions (§2.1.1.2.8).

Type	Character Code
char	c
double	d
float	f
int	i
long	l
long double	r
short	s
void	v
...	e

Table 1: C++ basic type encodings

Modifier	Character Code
array of size *n*	A*n*
volatile	V
const	C
function	F
pointer	P
reference	R
signed	S
unsigned	U

Table 2: C++ type modifier encodings

2. A diagnostic message should be printed for objects and functions that have declarations that are not type compatible (§3.1.2.6) (this requirement is a quality of implementation issue).

3. A diagnostic message should be printed when there is more than one definition for an object or function with external linkage (§3.5).

Function Name Encoding

Function name encoding is a technique where identifiers are encoded with auxiliary information signifying their type. This information is used at link time in order to detect declaration and definition inconsistencies. The scheme was first proposed by [4], and in connection with the C++ language in [2, p. 121–127]. It was also reportedly used by the the DTSS PL/I Linker developed at Dartmouth College in about 1975. The method presented in [2] works by appending a double underscore followed by character encodings of the function argument types after the function name. Basic types are encoded as single characters (listed in table 1), type modifiers and declarators are encoded by prepending character codes (listed in table 2) to the basic type and user defined classes are encoded using their identifier name prefixed by its length.

For example the function:

```
double
a(double b, int c)
{
        ...
}
```

would be encoded as `a__Fdi` where `F` stands for function, `d` for `double` and `i` for `int`. If the function was declared in another translation unit with a different declaration, then the two encoded function identifiers would not resolve at link time generating an "undefined symbol" error message. The description n [2] also contains encodings for the C++ operator functions, ways to minimise the length of the encodings, and a hashing proposal to deal with linkers with short identifier length limits.

The advantages of this approach, as given in [2, p. 122], are:

- the absence of extra-linguistic mechanisms (such as the C preprocessor or the *lint* program checker);
- the ease of implementation as no other programs need to understand the program structure, and
- the avoidance of the need to keep the headers consistent with the program source. (Headers are usually maintained by humans and thus can easily come out of sync with the actual implementation).

The scheme does not encode types of variables and return types of functions. This is necessary in order to ensure that errors arising from declaring a variable or function in two different modules with the same name, but different type or return type correspondingly, are caught by the linker. (Defining the same function with different argument types in separate modules is allowed in order to provide for C++ function overloading).

This scheme handles, in general, checks 1 and 3, and check 2 for function arguments. If the scheme was naively extended to handle global variables and function return types it would perform checks 1 and 2, but not check 3 thus altering the semantics of the language. For example the following which is not a correct program (§3.5) would link without a problem:

```
/* File 1 */
int i;

/* File 2 */
double i;
```

The naively extended scheme would encode the variable in *file a* as `i__i` and the variable in *file b* as `i__d`. The type clash would not be detected at link time as the two variables would end having different names.

The authors suggest [2, p. 123] that handling all inconsistencies would require either linker support or a mechanism allowing the compiler to access information from separate compilations. A solution to this problem without a need for linker modification

is presented in [11]. In the following section we discuss how this solution can be applied for declaration checking of C programs.

Extended Function Name Encoding

In order to be able to fully verify declarations and definitions of external linkage objects and function return types, one must encode the type of global variables and return types of functions into their names and, additionally, create dummy objects with the original names. Thus, the following two rules need to be added to the scheme:

1. Every function with external linkage has its return type encoded on its name by appending to the function name the return type encoding before the parameter encodings. In addition for every function, a dummy object definition named after the encoded function name is inserted into the object file.
2. Every object with external linkage has its type encoded into its name by appending an uppercase `V` followed by the type encoding to the object name. In addition, for every object with external linkage, a dummy object definition with the same name as the original object is inserted into the object file.

This scheme is essentially the naive extension presented in the previous section with an additional check in the form of dummy variables in order to handle the check in case 3. Objects with external linkage and conflicting definition and declaration/use will appear on the linker error list as unresolved references of the encoded names; object with external linkage and conflicting definitions will appear on the linker error list as multiple definitions of the un-encoded names.

In the following two examples we present sample programs with the corresponding encodings for the two cases:

Type Conflicting Declaration and Use

The following source files contain an error of type conflicting declaration and use:

```
/* File 1 (un-encoded) */
int a;

/* File 2 (un-encoded) */
extern double a;
void f(void)
{
        a = 3.14;
}
```

These files can be transformed into an equivalent source form with type encoding as follows:

```
/* File 1 (encoded) */
int a__Vi;
char a = 1;            /* Dummy variable */

/* File 2 (encoded) */
extern double a__Vd;

void f__Fvv(void)
{
        a__Vd = 3.14;
}
```

The linkage of the two files produces the following error on our system:

```
f2.o: undefined reference to '_a__Vd'
f2.o: undefined reference to '_a__Vd'
```

Type Conflicting Definition

The following two files contain a type conflicting definition:

```
/* File 1 (un-encoded) */
int a;

/* File 2 (un-encoded) */
double a;

void f(void) {}
```

These files can be transformed into an equivalent source form with type encoding as follows:

```
/* File 1 (encoded) */
int a__Vi;
char a = 1;            /* Dummy variable */

/* File 2 (encoded) */
double a__Vd;
char a = 1;            /* Dummy variable */

void f__Fvv(void) {}
```

The linkage of the two files produces the following error on our system:

```
f2.o: multiple definition of '_a (.data)'
f1.o: first seen here
```

Type	Character Code
void	v
char	c
signed char	b
unsigned char	a
short, signed short, signed short int, short int	s
unsigned short, unsigned short int,	t
int, signed, signed int	i
unsigned, unsigned int	u
long, signed long, long int, signed long int	l
unsigned long, unsigned long int	m
float	f
double	d
long double	r
...	e

Table 3: C basic type encodings

Type Encodings for ANSI C

The type encodings given in [2] and presented in the previous sections are not suitable for the C language. For example, the `unsigned` and `signed` modifiers used in the C++ approach will always define separate types. This is not the case in ANSI C according to §3.5.2 (`signed short int` and `short int` would be encoded as differently typed objects using the C++ modifiers, although the are the same C type). Therefore, we need to introduce another set of type encodings (listed in table 3) compatible with the distinct types of ANSI C specified in §3.5.2.

Other types and qualifiers are encoded using the following rules:

Arrays are encoded by prefixing the type of the object they consist of, by an uppercase `A` followed by the array length. When encoded as function arguments, they are converted to the appropriate pointer type as specified in §3.5.4.3, §3.7.1.

Structures are encoded with an uppercase `S` followed by the structure member names and type encodings — in the order they were declared — terminated by an underscore. (According to §3.1.2.6 structures must have the same member names to be type compatible). As an example the encoding `s__VS3leni4namePc_` represents the definition:

```
struct symbol {
        int len;
        char *name;
} s;
```

Unions are encoded with an uppercase U followed by the union member names and type encodings in alphabetical order. (According to §3.1.2.6 the order of union

member names is not taken into account for determining type compatibility).

Enumerations are encoded by the member names followed by their values encoded as fixed length hexadecimal constants.

Bit-fields are encoded by their type followed by an uppercase `B` followed by their length. If in an implementation `signed` bit-fields have a different type from plain `int` bit-fields, then a different letter must be used to encode `signed` and `signed int`.

Pointers are encoded by prefixing the type of the object they point to with an uppercase `P`.

Functions are encoded by an uppercase `F` followed by the function's return type, followed by the function's argument types.

Type qualifiers are encoded by prefixing the type encoding of the object with an uppercase `V` for `volatile`, and `C` for `const`. In addition the following two rules must be followed:

- when both are used in an encoding, `const` is encoded before `volatile`, as their order does not affect their type (§3.5.3), and
- `C` and `V` will not be used to encode function parameters, because argument type qualifiers are not taken into account for type compatibility of function types (§3.5.4.3).

As an example `volatile const signed char ch` is encoded as `ch_VCVb`. ■

Typedefs Typedefs are expanded to the type they define.

Implementation Issues

The system described in the previous sections has not yet been implemented. We have tried the various encodings to verify the soundness of the approach. One possible implementation approach we are currently considering would in the form of a source to source transformation filter that would run before the compiler proper. This could be used to retrofit recalcitrant compilers with advanced inter-module type checking. In the following paragraphs we will examine some other implementation details.

The easiest way to handle the dummy variables is to define them as character objects. This is straightforward to implement as a source to source transformation. However, these variables may take up some data space in the final executable image. A more elegant solution that uses no space, is to define the variables as global constants at the linker level. This can also be implemented in a source to source transformation if the compiler supports constants at linker level.

A "feature" often found in linkers is that of allowing more than a single definition for an object. This is often used to implement FORTRAN common blocks [10, §5.8]. It is also used by C implementations to allow a number of tentative definitions in different

modules [7, p. 227] (§3.7.2). The system proposed in this article depends on the linker detecting multiple definitions. This can be enforced on most linkers by supplying an initialiser to all dummy definitions used in the proposed scheme, thus removing their tentative quality. Some linkers however, silently ignore multiple definitions of the same object even when initialiser is supplied. In that case it may be impossible to implement the scheme.

An implementation of this scheme must take into account that supporting software needs to be made aware of the encoding scheme. For example, debuggers, profilers and other utilities that access the compiler generated object and executable code may need to be modified. In addition, the output of the linker could be filtered to convert the error messages to more meaningful ones.

Finally, the implementor of this approach must keep in mind the identifier length restrictions of the target linker. If these are severe (ANSI allows linkers with a 6 character identifier length restriction), then some form of hashing for longer identifiers has to be considered.

Conclusions

We have presented a complete scheme to enforce declaration and definition consistency of objects and functions with external linkage modules at link time. Objects and functions with external linkage are encoded with their type information to avoid conflicting declarations, and it addition, dummy character objects with external linkage are defined to avoid multiple conflicting definitions. The scheme can be used in separate compilation systems without modifying the linker.

References

[1] American National Standard for Information Systems — programming language — C: ANSI X3.159–1989. Published by the American National Standards Institute, 1430 Broadway, New York, New York 10018, December 1989. (Also ISO/IEC 9899:1990).

[2] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[3] S. I. Feldman. Make — a program for maintaining computer programs. *Software: Practice & Experience*, 9(4):255–265, 1979.

[4] Richard G. Hamlet. High-level binding with low-level linkers. *Communications of the ACM*, 19(11):642–644, November 1976.

[5] S. C. Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, Murray Hill, NJ, USA, December 1977.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, first edition, 1978.

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[8] Mt Xinu Inc., Berkeley, CA. *UNIX User's Reference Manual*, 2.6 MSD edition, January 1990.

[9] Leon Presser and John R. White. Linkers and loaders. *ACM Computing Surveys*, 4(3):149–167, September 1972.

[10] L. Rosler. The evolution of C — past and future. *Bell System Technical Journal*, 63(8), October 1984.

[11] Diomidis Spinellis. Type-safe linkage for variables and functions. *ACM SIGPLAN Notices*, 26(8):74–79, August 1991.

[12] Richard M. Stallman. Using and porting GNU CC. Free Software Foundation, 675 Mass Ave, Cambridge, MA, USA, May 1992.

[13] The Accredited Standard Committee X3, Information Processing Systems, Technical Committee for Programming Language C (X3J11). *Rationale for the ANSI C Programming Language*. Silicon Press, 25 Beverly Road, Summit, NJ 07901, USA, 1990.

Diomidis D. Spinellis is currently completing his Ph.D. on multiparadigm programming at Imperial College of Science, Technology and Medicine (University of London). Mr. Spinellis has worked as a contract programmer in the areas of CAD, Unix administration, multimedia, and product localisation for the Greek market. In relation to the C language he is best not known for his three winning entries in the International Obfuscated C Code Contest.

∞