Type–safe Linkage for Variables and Functions [*†]

Diomidis Spinellis
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
e-mail: dds@doc.ic.ac.uk

March 1991

Abstract

In a separate compilation environment type checks across modules are difficult to implement, because the natural place to perform them, the linker, is rarely under the control of the compiler developer. A solution to this problem, presented in the C++ Reference Manual, does not cope with global variables and function return types. It is asserted that lifting those limitations would require modifying the linker or providing an environment for separate compilation. We present a solution that lifts the limitations within the existing scheme.

1 Introduction

In many languages declarations and uses of language objects (e.g. variables and functions) must adhere to type rules. Furthermore a number of programming environments provide the facility of *separate compilation.* Languages that lend themselves to separate compilation usualy have a distinction between the *definition* part of a module where exported types and objects are declared and the *implementation* part of it where those and other objects are defined. Changes in the definiction module only require recompilation of that module, whereas changes to the definition module require recompilation of all modules that import it.

Separate compilation can either be implemented within a programming support environment framework where a database acts as a central repository for the source and compiled versions of all modules or it can be based on existing operating system functions with the file system acting as a repository database and the file modification time stamps being used for consistency control.

A problem inherent to separate compilation, using the underlying operating system functionality, is that of keeping the compiled objects consistent with each other. Using the file modification times is not a reliable method for many reasons:

- The programmer might change the modification times on purpose, using normal operating system commands, in order to avoid supposedly unneeded recompilations.
- The project could be developed on a network of different machines with unsynchronised clocks using a common filesystem for storage.
- Certain tools that operate on files can change the modification times.

[*] *SIGPLAN Notices*, 26(8):74–79, August 1991.

- Finally some languages like C and C++ do not define a separation between definition modules and implementation modules. Instead they provide facilities (e.g. the preprocessor) that a disciplined developer can use to implement this separation. Thus there are no rules that define which file depends on which.

For the reasons outlined above it is possible for type errors to remain undetected until the final linking phase of the program. Thus an additional check for type consistency accross modules at the time of the final link would increase the robustness of the development process.

Unfortunately adding type checking in a linker is difficult to implementent as usualy the linker is not under the control of the language implementor, but part of the operating system. Furthermore there are cases where parts of the linking are performed just before program execution by using dynamic link or shared libraries thus complicating the task of type checking even more.

This paper is organised as follows: First we examine some representative examples on how the problem arises and is dealt with, on various languages. Then we present the particularly elegant scheme proposed by Strousrtup [ES90, p. 121] of encoding the type of variables into their names. That scheme can not deal with global variables and function return types. We propose an extension to the scheme using dummy variables that would enable it to cover type checking all language objects at link time.

2 Representative Examples

2.1 Classic C

In traditional (pre ANSI) C, a separate program, *lint* [Joh77] handles the problem of type checking across modules [KR78, p. 103] by examining the set of compilation units as a whole and — among other things — verifying type correctness across them. *Lint* can 'precompile' crucial information from a set of compilation units into a library 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 work through the source text of the library. The solution is not very efficient as typically *lint* needs to examine all modules in order to determine if the functions exported are used in a consistent and type-safe way. It also depends on cooperation from the programmer in order to keep the *lint* libraries up to date.

2.2 Modula-2

The proposal given to handle the problem of type checking across modules in Modula-2 is based on the separation of compilation units into a definition and an implementation part [Wir85, p. 84]. The compiler can check the consistency of different modules by examining, the possibly precompiled, definition parts of every module used by the implementation part of the module being compiled. The solution depends on having the definition modules, the implementation modules and the possibly compiled versions of the definition modules mutually consistent at all times of the program development. A *make* [Fel79] utility can be used to ensure this, but depends on programmer cooperation.

2.3 ANSI C and C++

In ANSI C the type correctness of entities used can be ensured by proper declarations (function prototype declarations for function arguments) [KR88, p. 72] which can be placed in headers [KR88, p. 82]. The compiler reads the header files before reading the program source and thus verifies type correctness. Standard header files are provided for all library functions. A similar solution is presented for C++ in [Str86, p. 104-113]. The problem with this approach is again 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* [MSD90, mkdep(1)] can be used in conjunction with *make* to achieve

the desired effect. Their effectiveness depends on the cooperation, concentration and organizational powers of the programmer.

3 Type Safe Linking

From the examples given above it should be clear that the natural time to check for type consistency across modules should be the link time. At link time type consistency can be effectively policed without relying on extra-lingual tools and approaches, or to user cooperation.

3.1 Requirements for type correct linkage

The process of linking typically associates symbol references to symbol definitions [PW72]. In general, for a set of modules to be linked together successfully, there should be exactly one definition of an object for each set of references to it.

For linkage to be type correct across modules every usage of a symbol should be bound to its definition if and only if the two are type compatible, as defined in the corresponding language. Furthermore some objects should be defined exactly once. These static checks are the ones that are relevant for checking across modules from the checks that are presented in [ASU85, p. 343].

1. Objects which are type compatible should have their references bound to their definition (*if* property). Function overloading falls within the scope of this criterion.
2. Objects with matching names that are not type compatible should not be bound (*only if* property).
3. The linker should ensure that there is exactly one definition for each object in a given namespace (*uniqueness* property).

Traditional systems perform checks 1, 2 and 3 at compile time (depending on programmer cooperation) and enforce check 3 at link time (but for check 3 see also section 5).

3.2 Function Name Encoding

In [ES90, p. 122-127] an approach for type-safe linkage is presented. As a linker is typically not under the control of a language implementor any such scheme must work within the limitations of the linkers available. The scheme presented works within this constraint by encoding type information in function names. It roughly works by appending the types of the arguments passed to a function after the function name. Basic types are encoded as single characters, type modifiers and declarators are encoded by prepending character codes to the basic type and user defined classes are encoded using the name the user provided. 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`.
The advantages of this approach, as given in [ES90, 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;

- 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 done 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 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 type correct program links without a problem:

```
/* File a */
int i;

/* File b */
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 could not be detected at link time as the two variables would end having different names.

It is suggested [ES90, p. 123] that handling all inconsistencies would require either linker support or a mechanism allowing the compiler to access information from separate compilations. In the following paragraphs we will outline a simple addition to the scheme presented which lifts the limitations without any need to change the linker.

4 Extension to Function Name Encoding

In order to be able to fully type check global variables and function return types at compile time we encode the type of global variables and return types of functions into their names and, additionally, create dummy variables with the original names. Thus the following two rules need to be added to the scheme:

> Every function has its return type encoded on its name by appending an uppercase `R` followed by the return type to the function name. In addition for every function a dummy variable definition named after the encoded function name without the return type is inserted into the object file.
>
> Every global variable has its type encoded into its name by appending an uppercase `V` followed by the type encoding to the variable name. In addition for every global variable a dummy variable definition with the same name as the variable, but without the encodings is inserted into the object file.

This scheme is essentially the naive extension presented in section 3.2, with an additional check in the form of dummy variables in order to handle the type check in case 3. Global variables with conflicting definition and declaration/use will appear on the linker error output as unresolved references of the encoded names; global variables with conflicting definitions will appear on the linker output as multiple declarations of the unencoded names.

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

4.0.1 Type Conflicting Declaration and Use

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

```
/* File a (unencoded) */
int a;

/* File b (unencoded) */
extern double a;
main()
{
        a = 3.14;
}
```

These files can be transformed into an equivalent source form with type encoding as follows:[1]

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

/* File b (encoded) */
extern double a_Vd;

main()
{
        a_Vd = 3.14;
}
```

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

```
b.o: Undefined symbol _a_Vd referenced from text segment
b.o: Undefined symbol _a_Vd referenced from text segment
```

4.0.2 Type Conflicting Definition

The following two files contain a type conflicting definition:

```
/* File a (unencoded) */
int a;

/* File b (unencoded) */
double a;

main() {}
```

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

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

/* File b (encoded) */
double a_Vd;
char a = 1;            /* Dummy variable */

main() {}
```

[1] The function `main` is a special function needed by the system library and is not encoded.

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

```
a.o: Definition of symbol _a (multiply defined)
b.o: Definition of symbol _a (multiply defined)
```

The scheme may use up some data space on the final executable, depending on the compiler and linker technology used. This space is used up by the dummy variables. The easy way is to declare them as simple character variables. This is straightforward to implement as it can be implemented as a source to source transformation. A more elegant solution that uses no space is to define the variables as global constants at the linker level. This could be implemented in a source to source transformation if the compiler supports constants via the linker.

5 Implementation Issues

A capability often found in linkers is that of having more than a single definition for a set of references. This can be used to implement the FORTRAN common blocks [Ros84, §5.8]. It is also used by some C implementations which relax the single definition rule by allowing a number of tentative definitions in different modules [KR88, p. 227]. The system proposed in this article depends on the single definition rule. This can be enforced on some systems by supplying an initializer to all definitions used in the proposed scheme thus removing their tentative quality. Some other systems silently ignore multiple definitions of the same variable. No solution has been found for those systems.

Some languages (e.g. Modula-2[Wir85], Ada[ea83]) implement a tree model of entities where an entity is associated with a specific module. This implies that there can be several entities with the same name. As user defined entities are encoded using the user name the scheme needs another extension to handle this case. The obvious extension is to prepend the name of the module to the type name.

6 Conclusions

We have presented a complete scheme to handle type checking across modules at link time. Variables and function return types are handled with the help of additional dummy global variables. The scheme can be used in separate compilation systems without modifying the linker.

Acknowledgements

I would like to thank Sophia Drossopoulou and Susan Eisenbach for their helpful comments on early drafts of this paper.

Support from the British Science and Engineering Research Council is gratefully acknowledged.

References

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[ea83] Jean D. Ichbiah et al. *Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815 A-1983.* Castle House Publication Ltd., 1983.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Fel79] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice & Experience*, 9(4):255–265, 1979.

[Joh77] Stephen C. Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, Murray Hill, NJ, USA, December 1977.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, first edition, 1978.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[MSD90] Mt Xinu Inc., Berkeley, CA. *UNIX User's Reference Manual*, 2.6 MSD edition, January 1990.

[PW72] Leon Presser and John R. White. Linkers and loaders. *ACM Computing Surveys*, 4(3):149–167, September 1972.

[Ros84] L. Rosler. The evolution of C — past and future. *Bell System Technical Journal*, 63(8), October 1984.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, third edition, 1985.