Symbol management

$Revision: 1.3 $
$Date: 1999/01/05 06:36:33 $

Symbol management is a linker's key function. Without some way to refer from one module to another, there wouldn't be much use for a linker's other facilities.

Binding and name resolution

Linkers handle a variety of kinds of symbols. All linkers handle symbolic references from one module to another. Each input module includes a symbol table. The symbols include:

- Global symbols defined and perhaps referenced in the module
- Global symbols referenced but not defined in this module (generally called externals)
- Segment names, which are usually also considered to be global symbols defined to be at the beginning of the segment.
- Non-global symbols, usually for debuggers and crash dump analysis. (Sometimes mixed in with global symbols, somtimes in a separate table in the file.)
- Line number information, to tell source language debuggers the correspondence between source lines and object code. The linker reads all of the symbol tables in the input module, and extracts the useful information, which is sometimes all of the incoming info, frequently just what's needed to link. Then it builds the link-time symbol tables and uses that to guide the linking process. Depending on the output file format, the linker may place some or all of the symbol information in the output file.

Some formats have multiple symbol tables per file. For example, ELF shared libraries can have one symbol table with just the information needed for the dynamic linker and a separate, larger table useful for debugging and relinking. This isn't necessarily a bad design; the dynamic linker table is usually much smaller than the full table and making it separate can speed up the dynamic linking process, which happens far more often than a library is debugged or relinked.

Symbol table formats

Linker symbol tables are similar to those in compilers, although usually simpler, since the kinds of symbols a linker needs to keep are usually less complex than those in a compiler. There's one symbol table for input files and library modules, keeping the per-file information. A second symbol table handles global symbols, the ones that the linker has to resolve among input files. A third table may handle intra-module debugging symbols, although more often than not the linker need not create a full-fledged symbol table for debug symbols, needing only pass the debugging symbols through from the input to the output file.

Within the linker itself, a symbol table is often kept as an array of table entries, using a hash function to locate entries, or as an array of pointers, indexed by a hash function, with all of the entries that hash together chained from each header, Figure 1. To locate a symbol in the table, the linker computes a hash of the symbol name, uses that hash value modulo the number of buckets to select one of the hack buckets (symhash[h%NBUCKET] in the figure where h is the hash), runs down the chain of symbols looking for the symbol.

Traditionally, linkers only supported short names, ranging from eight charaters on IBM mainframes and early UNIX systems to six on most DEC systems to as few as two on some justly obscure minicomputers. Modern linkers support much longer names, both because programmers use longer names than they used to (or, in the case of Cobol, are no longer willing to twist the names around to make them unique in the first eight characters), and because compilers "mangle" names by adding extra characters to encode type information.

Older linkers did a string comparison of each symbol name in the lookup hash chain until they found a match or ran out of symbols, but this is a performance problem when a program contains many long symbols that can be identical up the last few characters, as is often the case with C++ mangled names. An easy fix is to store the full hash value in the symbol table and to do the string comparison only when the hashes match. Depending on the context, if a symbol is not found, the linker may either add it to the chain or report an error.

Figure 1: Symbol table

Typical symbol table with hashes or hash headers with chains of symbols

```
struct sym *symhash[NBUCKET];

struct sym {
  struct sym *next;
  int fullhash;/* full hash value */
  char *symname;
  ...
};
```

Module tables

The linker needs to track every input module seen during a linking run, both modules linked explicitly and those extracted from libraries. Figure 2 shows the structure of a simplified version of the module table for a GNU linker that produces a.out object files. Since most of the key information for each a.out file is in the file header, the table just stores a copy of the header,

Figure 2: Module table

```
/* Name of this file.  */
char *filename;
/* Name to use for the symbol giving address of text start */
/* Usually the same as filename, but for a file spec'd with -l
   this is the -l switch itself rather than the filename.  */
char *local_sym_name;
```

```
  /* Describe the layout of the contents of the file */

  /* The file's a.out header.  */
  struct exec header;
  /* Offset in file of GDB symbol segment, or 0 if there is none.  */
  int symseg_offset;

  /* Describe data from the file loaded into core */

  /* Symbol table of the file.  */
  struct nlist *symbols;
  /* Size in bytes of string table.  */
  int string_size;
  /* Pointer to the string table.
     The string table is not kept in core all the time,
     but when it is in core, its address is here.  */
  char *strings;

  /* Next two used only if `relocatable_output' or if needed for */
  /* output of undefined reference line numbers. */

  /* Text reloc info saved by `write_text' for `coptxtrel'.  */
  struct relocation_info *textrel;
  /* Data reloc info saved by `write_data' for `copdatrel'.  */
  struct relocation_info *datarel;

  /* Relation of this file's segments to the output file */

  /* Start of this file's text seg in the output file core image.  */
  int text_start_address;
  /* Start of this file's data seg in the output file core image.  */
  int data_start_address;
  /* Start of this file's bss seg in the output file core image.  */
  int bss_start_address;
  /* Offset in bytes in the output file symbol table
     of the first local symbol for this file.  Set by `write_file_symbols'.  *
  int local_syms_offset;
};
```

The table also contains pointers to in-memory copies of the symbol table string table (recall that a.out files store the symbol name strings in a separate table from the symbol table itself), and relocation tables, along with the computed offsets of the text, data, and bss segments in the output. If the file is a library, each library member that is being linked has its own module table entry. (Details not shown here.)

During the first pass, the linker reads in the symbol table from each file, generally just copying it verbatim into an in-memory buffer. In symbol formats that put the symbol names in a separate string table, the linker also reads in the symbol names and, for ease of subsequent processing, runs down the symbol table and turns each name string offset into a pointer to the in-memory version of the string.

Global symbol table

The linker keeps a global symbol table with an entry for every symbol referenced or defined in any input file, Figure 3. Each time the linker reads an input file, it puts all of the file's global symbols to the symbol table, keeping a chain of the places where the symbol is defined or referenced. When the first pass is done, every global symbol should have exactly one definition and zero or more references. (This is a minor oversimplification, since UNIX object files disguise common blocks as undefined symbols with non-zero values, but that's a straightforward special case for the linker to handle.)

Figure 3: Global symbol table

```
/* from gnu ld a.out */
  struct glosym
    {
      /* Pointer to next symbol in this symbol's hash bucket.  */
      struct glosym *link;
      /* Name of this symbol.  */
      char *name;
```

```
    /* Value of this symbol as a global symbol.  */
    long value;
    /* Chain of external 'nlist's in files for this symbol, both defs
and refs.  */
    struct nlist *refs;
    /* Nonzero means definitions of this symbol as common have been seen,
and the value here is the largest size specified by any of them.  */
    int max_common_size;
    /* For relocatable_output, records the index of this global sym in the
symbol table to be written, with the first global sym given index 0.*/
    int def_count;
    /* Nonzero means a definition of this global symbol is known to exist.
Library members should not be loaded on its account.  */
    char defined;
    /* Nonzero means a reference to this global symbol has been seen
in a file that is surely being loaded.
A value higher than 1 is the n_type code for the symbol's
definition.  */
    char referenced;
    /* 1 means that this symbol has multiple definitions.  2 means
       that it has multiple definitions, and some of them are set
elements, one of which has been printed out already.  */
    unsigned char multiply_defined;
  }
```

As the symbols in each file are added to the global symbol table, the linker links each entry from in the file to its corresponding global symbol table entry, Figure 4. Relocation items generally refer to symbols by index in the module's own symbol table, so for each external reference, the linker has to be able to tell that, for example, symbol 15 in module A is named `fruit`, while symbol 12 in module B is also named `fruit`, that is, it's the same symbol. Each module has its own set of indices and needs its own vector of pointers.

Figure 4: Resolving a symbol from a file to the global sym-

bol table

Each module entry points to vector of symbols from input file, each of which is set to point to global symbol table entry.

Symbol resolution

During the second pass of linking, the linker resolves symbol references as it creates the output file. The details of resolution interact with relocation (Chapter 7), since in most object formats, it's relocation entries that define where the references in the program to the symbol are. In the simplest case, in which the linker is creating an output file with absolute addresses (which includes data references in most Unix linkers) the address of the symbol simply replaces the symbol reference. If the symbol is resolved to address 20486, the linker replaces the reference with 20486.

Real situations are more complex. For one thing, there are many ways that a symbol might be referred to, in a data pointer, in an instruction, or even synthesized from multiple instructions. For another, the output of the linker is itself frequently relocatable. This means that if, say, a symbol is resolved to offset 426 in the data section, the output file has to contain a relocatable reference to data+426 where the symbol reference was.

The output file will usually have a symbol table of its own, so the linker needs to create a new vector of indexes of the symbols to be used in the output file, then map symbol numbers in outgoing relocation entries to those new indices.

Name mangling

The names used in object file symbol tables and in linking are often not the same names used in the source programs from which the object files were compiled. There are three reasons for this: avoiding name collisions, name overloading, and type checking.

Simple C and Fortran name mangling

In older object formats (before maybe 1970), compilers used names from the source program directly as the names in the object file, perhaps truncating long names to a name length limit. This worked reasonably well, but caused problems due to collisions with names reserved by compilers and libraries. For example, Fortran programs that do formatted I/O implicitly call routines in the library to do their reads and writes. Other routines handle arithmetic errors, complex arithmetic, and everything else in a programming language that's too complicated to be generated as in-line code.

The names of all of these routines are in effect reserved names, and part of the programming folklore was to know what names not to use. As a particularly egregious example, this Fortran program would for quite a few years crash an OS/360 system:

```
CALL MAIN
END
```

Why? The OS/360 programming convention is that every routine including the main program has a name, and the name of the main program is MAIN. When a Fortran main program starts, it makes a bunch of operating system calls to catch a variety of arithmetic error traps, and it turns out that each trap catch call allocated some space in a system table. But this program called itself recursively over and over again, each time establishing another nested set of trap calls, the system table ran out of space, and the system crashed. OS/390 is a lot more robust than its predecessors were 30 years ago, but the reserved name problem remains. It's even worse in mixed language programs, since code in all languages has to avoid using any name used by any of the language runtime libraries in use.

One approach to the reserved name problem was to use something other than procedure calls to call the runtime library. On the PDP-6 and -10, for example, the interface to the Fortran I/O package was through a variety of system call instruction that trapped back to the program rather than to the operating system. This was a clever trick, but it was quite specific to the PDP-6/10 architecture and didn't scale well, since there was no way for mixed language code to share the trap.

The approach taken on UNIX systems was to *mangle* the names of C and Fortran procedures so they wouldn't inadvertently collide with names of library and other routines. C procedure names were decorated with a leading underscore, so that `main` became `_main`. Fortran names were further mangled with both a leading and trailing underscore so that `calc` became `_calc_`. (This particular approach made it possible to call C routines whose names ended with an underscore from Fortran, a desirable situation that made it possible to write Fortran libraries in C.) The only significant disadvantage of this scheme is that it shrank the C name space from the 8 characters permitted by the object format to 7 characters for C and six characters for Fortran, but Fortran standards only required six character names, so it wasn't much of an imposition at the time.

On a lot of other systems, compiler designers took an opposite tack. Most assemblers and linkers permit characters in symbols that are forbidden in C and C++ identifiers such as . and $. Rather than mangling names from C or Fortran programs, the runtime libraries use names with forbidden characters that can't collide with application program names. The choice of name mangling vs. collision-proof library names is one of developer convenience. At the time UNIX was rewritten in C in about 1974, its authors already had extensive assembler language libraries, and it was easier to mangle the names of new C and C compatible routines than to go back and fix all the existing code. Now, twenty years later, the assembler code has all been rewritten five times and UNIX C compilers, particularly ones that create COFF and ELF object files, no longer prepend the underscore.

C++ type encoding: types and scopes

Another use for mangled names is to encode scope and type information, which makes it possible to use existing linkers to link programs in C++, Ada, and other languages that have more complex naming rules than do C, Cobol, or Fortran.

In a C++ program, the programmer can define many functions and variable with the same name but different scopes and, for functions, argument types. A single program may have a global variable `V` and a static member of a class `C::V`. C++ permits function name overloading, with several functions having the same name but different arguments, such as `f(int` `x)` and `f(float x)`. Class definitions can include functions, including overloaded names, and even functions that redefine built-in operators, that is, a class can contain a function whose name is in effect >> or any other built-in operator.

C++ was initially implemented as a translator called cfront that produced C code and used an existing linker, so its author used name mangling to produce names that can sneak through the C compiler into the linker. All the linker had to do with them was its usual job of matching identically named defined and undefined global names. Since then, nearly all C++ compilers generate object code or at least assembler code directly, but name mangling remains the standard way to handle overloaded names. Modern linkers now know enough about name mangling to demangle names reported in error messages, but otherwise leave mangled names alone.

The influential Annotated C++ Reference Manual described the name mangling scheme that cfront used, which with minor variations has become a de-facto standard. We describe it here.

Data variable names outside of C++ classes don't get mangled at all. An array called `foo` has a mangled name of `foo`. Function names not associated with classes are mangled to encode the types of the arguments by appending `__F` and a string of letters that represent the argument types and type modifiers listed in Figure 5. For example, a function `func(float, int, unsigned char)` becomes `func__FfiUc`. Class names are considered types, and are encoded as the length of the class name followed by the name, such as `4Pair`. Classses can contain names of internal classes to multiple levels; these "qualified" names are encoded as Q, a digit indicating the number of levels, and the encoded class names, so `First::Second::Third` becomes `Q35First6Second5Third`. This means that a function that takes two class arguments `f(Pair, First::Second::Third)` becomes `f__F5PairQ35First6Second5Third`.

Figure 5: Type letters in C++ mangled names

Type	Letter
void	v
char	c
short	s
int	i
long	l
float	f
double	d
long double	r
varargs	e
unsigned	U
const	C
volatile	V
signed	S
pointer	P
reference	R
array of length *n*	A*n*_
function	F
pointer to nth member	M*n*S

Class member functions are encoded as the function name, two underscores, the encoded class name, then F and the arguments, so `cl::fn(void)` becomes `fn__2clFv`. All of the operators have four or five character encoded names as well, such as `__ml` for `*` and `__aor` for `|=`, and special functions including constructor, destructor, new, and delete have encodings as well `__ct`, `__dt`, `__nw`, and `__dl`. A constructor for class Pair taking two character pointer arguments `Pair(char*,char*)` becomes `__ct__4PairFPcPc`.

Finally, since mangled names can be so long, there are two shortcut encodings for functions with multiple arguments of the same type. The code `Tn` means "same type as the nth argument" and `Nnm` means "n arguments the same type as the mth argument. A function `segment(Pair, Pair)` would be `segment__F4PairT1` and a function `trapezoid(Pair, Pair, Pair, Pair)` would be `trapezoid__F4PairN31`.

Name mangling does the job of giving unique names to every possible C++ object at the cost of tremendously long and (lacking linker and debugger support) unreadable names in error messages and listings. Nonetheless, C++ has an intrinsic problem that it has a potentially huge namespace, and any scheme for representing the names of C++ objects has nearly as verbose as name mangling, and mangled names do have the advantage of being readable by at least some humans.

Early users of mangled names often found that although linkers in theory supported long names, in practice the long names didn't work very well, and performance was dreadful when linking programs that contained many long names that were identical up to the last few characters. Fortunately, symbol table algorithms are a well-understood subject, and now one can expect linkers to handle long names without trouble.

Link-time type checking

Although mangled names only became popular with the advent of C++, the idea of linker type checking has been around for a long time. (I first encountered in the Dartmouth PL/I linker in about 1974.) The idea of linker type checking is quite straightforward. Most languages have procedures with declared argument types, and if the caller doesn't pass the number and type of arguments that the callee expects, it's an error, often a hard-to-diagnose error if the caller and callee are in separately compiled files. For linker type checking, each defined or undefined global symbol has associated with it a string representing the argument and return types, similar to the mangled C++ argument types. When the linker resolves a symbol, it compares the type strings for the reference and definition of the symbol, and reports an error if they don't match. A nice property of this scheme is that the linker need not understand the type encoding at all, just whether the strings are the same or not.

Even in an environment with C++ mangled names, this type checking would still be useful, since not all C++ type information is encoded into a mangled name. The types that functions return, and types of global data could profitably be checked by a scheme like this one.

Weak external and other kinds of symbols

Up to this point, we've considered all linker global symbols to work the same way, and each mention of a name to be either a definition or a reference to a symbol. Many object formats can qualify a reference as weak or strong. A strong reference must be resolved, while a weak reference may be resolved if there's a definition, but it's not an error if it's not. Linker processing of weak symbols is much like that for strong symbols, except that at the end of the first pass an undefined reference to one isn't an error. Generally the linker defines undefined weak symbols to be zero, a value that application code can check. Weak symbols are primarily useful in connection with libraries, so we revisit them in Chapter 6.

Maintaining debugging information

Modern compilers all support source language debugging. That means that the programmer can debug the object code referring to source program function and variable names, and set breakpoints and single step the program. Compilers support this by putting information in the object file that provides a mapping from source file line numbers to object code addresses, and also describes all of the functions, variables, types, and structures used in the program.

UNIX compilers have two somewhat different debug information formats, stab (short for symbol table) that are used primarily in a.out and COFF files, and DWARF that was defined for ELF files. Microsoft has defined their own formats for their Codeview debugger, with CV4 being the most recent.

Line number information

All symbolic debuggers need to be able to map between program addresses and source line numbers. This lets users set breakpoints by line number with the debugger placing the breakpoint at the appropriate place in the code, and also lets the debugger relate the program addresses in call stack tracebacks and error reports back to source lines.

Line number information tends not to be very complicated execpt that optimizing compilers can move code around so that the sequence of code in the object file doesn't match the sequence of source lines.

For each line in the source file for which the compiler generated any code, the compiler emits a line number entry with the line number and the beginning of the code. If a program address lies between two line number entries, the debugger reports it as being the lower of the two line numbers. The line numbers need to be scoped by file name, both source file name and include file name. Some formats do this by creating a list of files and putting a file index in each line number entry. Others intersperse "begin include" and "end include" items in the list of line numbers, implicitly maintaining a stack of line numbers.

When compiler optimization makes the generated code from a single statement discontiguous, some object formats (notably DWARF) let the compiler map each byte of object code back to a source line, using a lot of space in the process, while others just emit approximate locations.

Symbol and variable information

Compilers also have to emit the names, types, and locations of each program variable. The debug symbol information is somewhat more complex than mangled names are, because it needs to encode not just the type names, but for structure types the definitions of the types so the debugger can correctly format all of the subfields in a structure.

The symbol information is an implicit or explicit tree. At the top level in each file is a list of types, variables, and functions defined at the top level, and within each of those are the fields of structures, variables defined within functions, and so forth. Within functions, the tree includes "begin block" and "end block" markers referring to line numbers, so the debugger can tell what variables are in scope at each point in the program.

The trickiest part of the symbol information is the location information. The location of a static variable doesn't change, but a local variable within a a routine may be static, on the stack, in a register, or in optimized code, moved from place to place in different parts of the routine. On most architectures, the standard calling sequence for routines maintains a chain of saved stack and frame pointers for each nested routine, with the local stack variables in each routine allocated at known offsets from the frame pointer. In leaf routines or routines that allocate no local stack variables, a common optimization is to skip setting the frame pointer. The debugger needs to know about this in order both to interpret call stack tracebacks correctly and to find local variables in a routine with no frame pointer. Codeview does this with a specific list of routines with no frame pointer.

Practical issues

For the most part, the linker just passes through debug information uninterpreted, perhaps relocating segment-relative addresses on the way through.

One thing that linkers are starting to do is detecting and removing duplicated debug information. In C and particularly C++, programs usually have a set of header files that define types and declare functions, and each source file includes the headers that define all of the types and functions that file might use.

Compilers pass through the debug information for everything in all of the header files that each source file includes. This means that if a particular header file is included by 20 source files that are compiled and linked together, the linker will receive 20 copies of the debug information for that file. Although debuggers have never had any trouble disregarding the duplicated information, header files, particularly in C++, can be large which means that the amount of duplicated header info can be substantial. Linkers can safely discard the duplicated material, and increasingly do so, both to speed the linker and debugger and to save space. In some cases, compilers put the debug information directly into files or databases to be read by the debugger, bypassing the linker, so the linker need only add or update information about the relative locations of the segments contributed by each source file, and any data such as jump tables created by the linker itself.

When the debug information is stored in an object file, sometimes the debug information is intermixed with the linker symbols in one big symbol table, while sometimes the two are separate. Unix systems added debug information to the compilers a little at a time over the years, so it all ended up in one huge symbol table. Other formats including Microsoft's ECOFF tend to separate linker symbols from debug symbols and both from line numbers.

Sometimes the resulting debug information goes into the output file, sometimes into a separate debug file, sometimes both. The advantage of putting all of the debug information into the output file is simplicity in the build process, since all of the information used to debug the program is present in one place. The most obvious disadvantage is that it makes the executable file enormous. Also if the debug information is separated out, it's easy to build a final version of a program, then ship the executable but not the debug files. This keeps the size of the shipped program down and discourages casual reverse engineering, but the developers still have the debug files if needed to debug errors found in the shipping project. UNIX systems have a "strip" command that removes the debugging symbols from an object file but doesn't change the code at all. The developers keep the unstripped file and ship the stripped version. Even though the two files are different, the running code is the same and the debugger can use the symbols from the unstripped file to debug a core dump made from the stripped version.

Exercises

Write a C++ program with a lot of functions whose mangled names differ only in the last few characters. See how long they take to compile. Change them so the mangled names differ in the first few characters. Time a compile and link again. Do you need a new linker?

Investigate the debug symbol format that your favorite linker uses. (Some on-line resources are listed in the bibiography.) Write a program to dump the debugging symbols from an object file and see how much of the source program you can reconstruct from it.