The Makeheaders Program

This document describes *makeheaders*, a tool that automatically generates ``.h'' files for a C or C++ programming project.

Table Of Contents

1.0 Background

A piece of C source code can be one of two things: a *declaration* or a *definition*. A declaration is source text that gives information to the compiler but doesn't directly result in any code being generated. A definition is source text that results in executable machine instructions or initialization data. (These two terms are sometimes used inconsistently by other authors. In particular, many people reverse the meanings of these words when discussing Pascal or Ada code. The meanings described here are the same as used in the ANSI-C standards document.)

Declarations in C include things such as the following:

- Typedefs.
- Structure, union and enumeration declarations.
- Function and procedure prototypes.
- Preprocessor macros and #defines.
- ``extern'' variable declarations.

Definitions in C, on the other hand, include these kinds of things:

- Variable definitions.
- The bodies of functions and procedures.
- Initialization data.

The distinction between a declaration and a definition is common in modern software engineering. Another way of looking at the difference is that the declaration is the *interface* and the definition is the *implementation*.

In C programs, it has always been the tradition that declarations are put in files with the ``.h'' suffix and definitions are placed in ``.c'' files. The .c files contain ``#include'' preprocessor statements that cause the contents of .h files to be included as part of the source code when the .c file is compiled. In this way, the .h files define the interface to a subsystem and the .c files define how the subsystem is implemented.

1.1 Problems With The Traditional Approach

As the art of computer programming continues to advance, and the size and complexity of programs continues to swell, the traditional C approach of placing declarations and definitions in separate files begins to present the programmer with logistics and maintenance problems. To wit:

1. In large codes with many source files, it becomes difficult to determine which .h files should be included in which .c files.

2. It is typically the case the a .h file will be forced to include another .h files, which in turn might include other .h files, and so forth. The .c file must be recompiled when any of the .h files in this chain are altered, but it can be difficult to determine what .h files are found in the include chain. A frequent Makefile error is to omit some .h files from a dependency list even though those files are on the include file chain.

3. Some information is common to both the declaration and the definition of an object in C, and so must be repeated in both the .h and the .c files for that object. In a large project, it can become increasingly difficult to keep the two files in sync.

4. When a .c file includes a .h file and the .h files changes, the .c file must be recompiled, even if the part of the .h file that changed is not actually used by the .c file. In a large program, it is generally the case that almost every .c file ends up depending on one or two of the more important .h files, and so when those .h files change, the entire program must be recompiled. It also happens that those important .h files tend to be the ones that change most frequently. This means that the entire program must be recompiled frequently, leading to a lengthy modify-compile-test cycle and a corresponding decrease in programmer productivity.

5. The C programming language requires that declarations depending upon each other must occur in a particular order. In a program with complex, interwoven data structures, the correct declaration order can become very difficult to determine manually, especially when the declarations involved are spread out over several files.

1.2 The Makeheaders Solution

The makeheaders program is designed to ameliorate the problems associated with the traditional C programming model by automatically generating the interface information in the .h files from interface information contained in other .h files and from implementation information in the .c files. When the makeheaders program is run, it scans the source files for a project, then generates a series of new .h files, one for each .c file. The generated .h files contain exactly those declarations required by the corresponding .c files, no more and no less.

The makeheaders programming model overcomes all of the objections to the traditional C programming model.

1. Because all declarations needed by a .c file are contained in a single .h file, there is never any question about what .h files a .c will need to include. If the .c file is named `alpha.c` then it must include only the single .h file named `alpha.h`. (The .c file might also use some include files from the standard library, such as `<stdio.h>`, but that is another matter.)

2. The generated .h files do not include other .h files, and so there are no include chains to worry about. The file `alpha.c` depends on `alpha.h` and nothing more.

3. There is still duplication in the .h and the .c file, but because the duplicate information is automatically generated, it is no longer a problem. Simply rerun makeheaders to resynchronize everything.

4. The generated .h file contains the minimal set of declarations needed by the .c file. This means that when something changes, a minimal amount of recompilation is required to produce an updated executable. Experience has shown that this gives a dramatic improvement in programmer productivity by facilitating a rapid modify-compile-test cycle during development.

5. The makeheaders program automatically sorts declarations into the correct order, completely eliminating the wearisome and error-prone task of sorting declarations by hand.

In addition, the makeheaders program is fast and unintrusive. It is a simple matter to incorporate makeheaders into a Makefile so that makeheaders will be run automatically whenever the project is rebuilt. And the burden of running makeheaders is light. It will easily process tens of thousands of lines of source code per second.

2.0 Running The Makeheaders Program

The makeheaders program is very easy to run. If you have a collection of C source code and include files in the working directory, then you can run makeheaders to generate appropriate .h files using the following command:

```
makeheaders *.[ch]
```

That's really all there is to it! This command will generate one .h file for every .c file. Any .h files that were generated by a prior run of makeheaders are ignored, but manually entered .h files that contain structure declarations and so forth will be scanned and the declarations will be copied into the generated .h files as appropriate. But if makeheaders sees that the .h file that it has generated is no different from the .h file it generated last time, it doesn't update the file. This prevents the corresponding .c files from having to be needlessly recompiled.

There are several options to the makeheaders program that can be used to alter its behavior. The default behavior is to write a single .h file for each .c file and to give the .h file the same base name as the .c file. Instead of generating a whole mess of .h files, you can, if you choose, generate a single big .h file that contains all declarations needed by all the .c files. Do this using the -h option to makeheaders. As follows:

```
makeheaders -h *.[ch] >common.h
```

With the -h option, the .h file is not actually written to a disk file but instead appears on standard output, where you are free to redirect it into the file of your choice.

A similar option is -H. Like the lower-case -h option, big -H generates a single include file on standard output. But unlike small -h, the big -H only emits prototypes and declarations that have been designated as ``exportable''. The idea is that -H will generate an include file that defines the interface to a library. More will be said about this in section 3.4.

Sometimes you want the base name of the .c file and the .h file to be different. For example, suppose you want the include file for `alpha.c` to be called `beta.h`. In this case, you would invoke makeheaders as follows:

```
makeheaders alpha.c:beta.h
```

Any time a filename argument contains a colon, the name before the colon is taken to be the name of the .c file and the name after the colon is taken to be the name of the .h file. You can't use the shell's wildcard mechanism with this approach, but that normally isn't a problem in Makefiles, which is where this stuff comes in handy.

If you want a particular file to be scanned by makeheaders but you don't want makeheaders to generate a header file for that file, then you can supply an empty header filename, like this:

```
   makeheaders alpha.c beta.c gamma.c:
```

In this example, makeheaders will scan the three files named ``alpha.c'', ``beta.c'' and ``gamma.c'' but because of the colon on the end of third filename it will only generate headers for the first two files. Unfortunately, it is not possible to get makeheaders to process any file whose name contains a colon.

In a large project, the length of the command line for makeheaders can become very long. If the operating system doesn't support long command lines (example: DOS and Win32) you may not be able to list all of the input files in the space available. In that case, you can use the ``-f'' option followed by the name of a file to cause makeheaders to read command line options and filename from the file instead of from the command line. For example, you might prepare a file named ``mkhdr.dat'' that contains text like this:

```
  src/alpha.c:hdr/alpha.h
  src/beta.c:hdr/beta.h
  src/gamma.c:hdr/gamma.h
  ...
```

Then invoke makeheaders as follows:

```
  makeheaders -f mkhdr.dat
```

The ``-local'' option causes makeheaders to generate of prototypes for ``static'' functions and procedures. Such prototypes are normally omitted.

Finally, makeheaders also includes a ``-doc'' option. This command line option prevents makeheaders from generating any headers at all. Instead, makeheaders will write to standard output information about every definition and declaration that it encounters in its scan of source files. The information output includes the type of the definition or declaration and any comment that preceeds the definition or declaration. The output is in a format that can be easily parsed, and is intended to be read by another program that will generate documentation about the program. We'll talk more about this feature later.

If you forget what command line options are available, or forget their exact name, you can invoke makeheaders using an unknown command line option (like ``--help'' or ``-?'') and it will print a summary of the available options on standard error. If you need to process a file whose name begins with ``-'', you can prepend a ``./'' to its name in order to get it accepted by the command line parser. Or, you can insert the special option ``--'' on the command line to cause all subsequent command line arguments to be treated as filenames even if their names beginn with ``-''.

3.0 Preparing Source Files For Use With Makeheaders

Very little has to be done to prepare source files for use with makeheaders since makeheaders will read and understand ordinary C code. But it is important that you structure your files in a way that makes sense in the makeheaders context. This section will describe several typical uses of makeheaders.

3.1 The Basic Setup

The simpliest way to use makeheaders is to put all definitions in one or more .c files and all structure and type declarations in separate .h files. The only restriction is that you should take care to chose basenames for your .h files that are different from the basenames for you .c files. Recall that if your .c file is named (for example) ``alpha.c'' makeheaders will attempt to generate a corresponding header file named ``alpha.h''. For that reason, you don't want to use that name for any of the .h files you write since that will prevent makeheaders from generating the .h file automatically.

The structure of a .c file intented for use with makeheaders is very simple. All you have to do is add a single ``#include'' to the top of the file that sources the header file that makeheaders will generate. Hence, the beginning of a source file named ``alpha.c'' might look something like this:

```
/*
 * Introductory comment...
 */
#include "alpha.h"

/* The rest of your code... */
```

Your manually generated header files require no special attention at all. Code them as you normally would. However, makeheaders will work better if you omit the ``#if'' statements people often put around the outside of header files that prevent the files from being included more than once. For example, to create a header file named ``beta.h'', many people will habitually write the following:

```
#ifndef BETA_H
#define BETA_H

/* declarations for beta.h go here */

#endif
```

You can forego this cleverness with makeheaders. Remember that the header files you write will never really be included by any C code. Instead, makeheaders will scan your header files to extract only those declarations that are needed by individual .c files and then copy those declarations to the .h files corresponding to the .c files. Hence, the ``#if'' wrapper serves no useful purpose. But it does make makeheaders work harder, forcing it to put the statements

```
#if !defined(BETA_H)
#endif
```

around every declaration that it copies out of your header file. No ill effect should come of this, but neither is there any benefit.

Having prepared your .c and .h files as described above, you can cause makeheaders to generate its .h files using the following simple command:

```
makeheaders *.[ch]
```

The makeheaders program will scan all of the .c files and all of the manually written .h files and then automatically generate .h files corresponding to all .c files.

Note that the wildcard expression used in the above example, ``*.[ch]'', will expand to include all .h files in the current directory, both those entered manually be the programmer and others generated automatically by a prior run of makeheaders. But that is not a problem. The makeheaders program will recognize and ignore any files it has previously generated that show up on its input list.

3.2 What Declarations Get Copied

The following list details all of the code constructs that makeheaders will extract and place in the automatically generated .h files:

- When a function is defined in any .c file, a prototype of that function is placed in the generated .h file of every .c file that calls the function.

  If the ``static'' keyword of C appears at the beginning of the function definition, the prototype is suppressed. If you use the ``LOCAL'' keyword where you would normally say ``static'', then a prototype is generated, but it will only appear in the single header file that corresponds to the source file containing the function. For example, if the file `alpha.c` contains the following:

  ```
  LOCAL int testFunc(void){
    return 0;
  }
  ```

  Then the header file `alpha.h` will contain

  ```
  #define LOCAL static
  LOCAL int testFunc(void);
  ```

  However, no other generated header files will contain a prototype for `testFunc()` since the function has only file scope.

  When the ``LOCAL'' keyword is used, makeheaders will also generate a #define for LOCAL, like this:

  ```
   #define LOCAL static
  ```

  so that the C compiler will know what it means.

  If you invoke makeheaders with a ``-local'' command-line option, then it treats the ``static'' keyword like ``LOCAL'' and generates prototypes in the header file that corresponds to the source file containing the function defintiion.

- When a global variable is defined in a .c file, an ``extern'' declaration of that variable is placed in the header of every .c file that uses the variable.

- When a structure, union or enumeration declaration or a function prototype or a C++ class declaration appears in a manually produced .h file, that declaration is copied into the automatically generated .h files of all .c files that use the structure, union, enumeration, function or class. But declarations that appear in a .c file are considered private to that .c file and are not copied into any automatically generated files.

- All #defines and typedefs that appear in manually produced .h files are copied into automatically generated .h files as needed. Similar constructs that appear in .c files are considered private to those files and are not copied.

- When a structure, union or enumeration declaration appears in a .h file, makeheaders will automatically generate a typedef that allows the declaration to be referenced without the ``struct'', ``union'' or ``enum'' qualifier. In other words, if makeheaders sees the code:

  ```
  struct Examp { /* ... */ };
  ```

  it will automatically generate a corresponding typedef like this:

  ```
  typedef struct Examp Examp;
  ```

- Makeheaders generates an error message if it encounters a function or variable definition within a .h file. The .h files are suppose to contain only interface, not implementation. C compilers will not enforce this convention, but makeheaders does.

As a final note, we observe that automatically generated declarations are ordered as required by the ANSI-C programming language. If the declaration of some structure ``X'' requires a prior declaration of another structure ``Y'', then Y will appear first in the generated headers.

3.3 How To Avoid Having To Write Any Header Files

In my experience, large projects work better if all of the manually written code is placed in .c files and all .h files are generated automatically. This is slightly different for the traditional C method of placing the interface in .h files and the implementation in .c files, but it is a refreshing change that brings a noticable improvement to the coding experience. Others, I believe, share this view since I've noticed recent languages (ex: java, tcl, perl, awk) tend to support the one-file approach to coding as the only option.

The makeheaders program supports putting both interface and implementation into the same source file. But you do have to tell makeheaders which part of the source file is the interface and which part is the implementation. Makeheaders has to know this in order to be able to figure out whether or not structures declarations, typedefs, #defines and so forth should be copied into the generated headers of other source files.

You can instruct makeheaders to treat any part of a .c file as if it were a .h file by enclosing that part of the .c file within:

```
#if INTERFACE
#endif
```

Thus any structure definitions that appear after the ``#if INTERFACE'' but before the corresponding ``#endif'' are eligable to be copied into the automatically generated .h files of other .c files.

If you use the ``#if INTERFACE'' mechanism in a .c file, then the generated header for that .c file will contain a line like this:

```
#define INTERFACE 0
```

In other words, the C compiler will never see any of the text that defines the interface. But makeheaders will copy all necessary definitions and declarations into the .h file it generates, so .c files will compile as if the declarations were really there. This approach has the advantage that you don't have to worry with putting the declarations in the correct ANSI-C order -- makeheaders will do that for you automatically.

Note that you don't have to use this approach exclusively. You can put some declarations in .h files and others within the ``#if INTERFACE'' regions of .c files. Makeheaders treats all declarations alike, no matter where they come from. You should also note that a single .c file can contain as many ``#if INTERFACE'' regions as desired.

3.4 Designating Declarations For Export

In a large project, one will often construct a hierarchy of interfaces. For example, you may have a group of 20 or so files that form a library used in several other parts of the system. Each file in this library will present two interfaces. One interface will be the routines and data structures it is willing to share with other files in the same library, and the second interface is those routines and data structures it wishes to make available to other subsystems. (The second interface is normally a subset of the first.) Ordinary C does not provide support for a tiered interface like this, but makeheaders does.

Using makeheaders, it is possible to designate routines and data structures as being for ``export''. Exported objects are visible not only to other files within the same library or subassembly but also to other libraries and subassemblies in the larger program. By default, makeheaders only makes objects visible to other members of the same library.

That isn't the complete truth, actually. The semantics of C are such that once an object becomes visible outside of a single source file, it is also visible to any user of the library that is made from the source file. Makeheaders can not prevent outsiders for using non-exported resources, but it can discourage the practice by refusing to provide prototypes and declarations for the services it does not want to export. Thus the only real effect of the making an object exportable is to include it in the output makeheaders generates when it is run using the -H command line option. This is not a perfect solution, but it works well in practice.

But trouble quickly arises when we attempt to devise a mechanism for telling makeheaders which prototypes it should export and which it should keep local. The built-in ``static'' keyword of C works well for prohibiting prototypes from leaving a single source file, but because C doesn't support a linkage hierarchy, there is nothing in the C language to help us. We'll have to invite our own keyword: ``EXPORT''

Makeheaders allows the EXPORT keyword to precede any function or procedure definition. The routine following the EXPORT keyword is then eligable to appear in the header file generated using the -H command line option. Note that if a .c file contains the EXPORT keyword, makeheaders will put the macro

```
   #define EXPORT
```

in the header file it generates for the .c file so that the EXPORT keyword will never be seen by the C compiler.

But the EXPORT keyword only works for function and procedure definitions. For structure, union and enum definitions, typedefs, #defines and class declarations, a second mechanism is used. Just as any declarations or definition contained within

```
   #if INTERFACE
   #endif
```

are visible to all files within the library, any declarations or definitions within

```
   #if EXPORT_INTERFACE
   #endif
```

will become part of the exported interface. The ``#if EXPORT_INTERFACE'' mechanism can be used in either .c or .h files. (The ``#if INTERFACE'' can also be used in both .h and .c files, but since it's use in a .h file would be redundant, we haven't mentioned it before.)

3.5 Local declarations processed by makeheaders

Structure declarations and typedefs that appear in .c files are normally ignored by makeheaders. Such declarations are only intended for use by the source file in which they appear and so makeheaders doesn't need to copy them into any generated header files. We call such declarations ``private''.

Sometimes it is convenient to have makeheaders sort a sequence of private declarations into the correct order for us automatically. Or, we could have static functions and procedures for which we would like makeheaders to generate prototypes, but the arguments to these functions and procedures uses private declarations. In both of these cases, we want makeheaders to be aware of the private declarations and copy them into the local header file, but we don't want makeheaders to propagate the declarations outside of the file in which they are declared.

When this situation arises, enclose the private declarations within

```
  #if LOCAL_INTERFACE
  #endif
```

A ``LOCAL_INTERFACE'' block works very much like the ``INTERFACE'' and ``EXPORT_INTERFACE'' blocks described above, except that makeheaders insures that the objects declared in a LOCAL_INTERFACE are only visible to the file containing the LOCAL_INTERFACE.

3.6 Using Makeheaders With C++ Code

You can use makeheaders to generate header files for C++ code, in addition to C. Makeheaders will recognize and copy both ``class'' declarations and inline function definitions, and it knows not to try to generate prototypes for methods.

In fact, makeheaders is smart enough to be used in projects that employ a mixture of C and C++. For example, if a C function is called from within a C++ code module, makeheaders will know to prepend the text

```
   extern "C"
```

to the prototype for that function in the C++ header file. Going the other way, if you try to call a C++ function from within C, an appropriate error message is issued, since C++ routines can not normally be called by C code (due to fact that most C++ compilers use name mangling to facilitate type-safe linkage.)

No special command-line options are required to use makeheaders with C++ input. Makeheaders will recognize that its source code is C++ by the suffix on the source code filename. Simple ".c" or ".h" suffixes are assumed to be ANSI-C. Anything else, including ".cc", ".C" and ".cpp" is assumed to be C++. The name of the header file generated by makeheaders is derived from by the name of the source file by converting every "c" to "h" and every "C" to "H" in the suffix of the filename. Thus the C++ source file ``alpha.cpp'' will induce makeheaders to generate a header file named ``alpha.hpp''.

Note that makeheaders does not understand some of the more obscure C++ syntax such as templates. Perhaps these issued will be addressed in future revisions.

3.7 Conditional Compilation

The makeheaders program understands and tracks the conditional compilation constructs in the source code files it scans. Hence, if the following code appears in a source file

```
  #ifdef UNIX
  #  define WORKS_WELL  1
  #else
  #  define WORKS_WELL  0
  #endif
```

then the next patch of code will appear in the generated header for every .c file that uses the WORKS_WELL constant:

```
  #if defined(UNIX)
  #  define WORKS_WELL 1
  #endif
  #if !defined(UNIX)
  #  define WORKS_WELL 0
  #endif
```

The conditional compilation constructs can be nested to any depth. Makeheaders also recognizes the special case of

```
  #if 0
  #endif
```

and treats the enclosed text as a comment.

3.8 Caveats

The makeheaders system is designed to be robust but it is possible for a devious programmer to fool the system, usually with unhelpful consequences. This subsection is a guide to helping you avoid trouble.

Makeheaders does not understand the old K&R style of function and procedure definitions. It only understands the modern ANSI-C style, and will probably become very confused if it encounters an old K&R function. You should take care to avoid putting K&R function defintions in your code, therefore.

Makeheaders does not understand when you define more than one global variable with the same type separated by a comma. In other words, makeheaders does not understand this:

```
   int a = 4, b = 5;
```

The makeheaders program wants every variable to have its own definition. Like this:

```
   int a = 4;
   int b = 5;
```

Notice that this applies to global variables only, not to variables you declare inside your functions. Since global variables ought to be exceedingly rare, and since it is good style to declare them separately anyhow, this restriction is not seen as a terrible hardship.

The makeheaders program processes its source file prior to sending those files through the C preprocessor. Hence, if you hide important structure information in preprocessor defines, makeheaders might not be able to successfully extract the information it needs from variables, functions and procedure definitions. For example, if you write this:

```
  #define BEGIN {
  #define END }
```

at the beginning of your source file, and then try to create a function definition like this:

```
  char *StrDup(const char *zSrc)
    BEGIN
      /* Code here */
    END
```

then makeheaders won't be able to find the end of the function definition and bad things are likely to happen.

For most projects the code constructs that makeheaders cannot handle are very rare. As long as you avoid excessive cleverness, makeheaders will probably be able to figure out what you want and will do the right thing.

4.0 Using Makeheaders To Generate Documentation

Many people have observed the advantages of generating program documentation directly from the source code:

- Less effort is involved. It is easier to write a program than it is to write a program and a document.
- The documentation is more likely to agree with the code. When documentation is derived directly from the code, or is contained in comments immediately adjacent to the code, it is much more likely to be correct than if it is contained in a separate unrelated file in a different part of the source tree.
- Information is kept in only one place. When a change occurs in the code, it is not necessary to make a corresponding change in a separate document. Just rerun the documentation generator.

The makeheaders program does not generate program documentation itself. But you can use makeheaders to parse the program source code, extract the information that is relavant to the documentation and to pass this information to another tool to do the actual documentation preparation.

When makeheaders is run with the ``-doc'' option, it emits no header files at all. Instead, it does a complete dump of its internal tables to standard outputs in a form that is easily parsed. This output can then be used by another program (the implementation of which is left as an exercise to the reader) that will use the information to prepare suitable documentation.

The ``-doc'' option causes makeheaders to print information to standard output about all of the following objects:

- C++ Class declarations
- Structure and union declarations
- Enumerations
- Typedefs
- Procedure and function definitions
- Global variables
- Preprocessor macros (ex: ``#define'')

For each of these objects, the following information is output:

- The name of the object.
- The type of the object. (Structure, typedef, macro, etc.)
- Flags to indicate if the declaration is exported (contained within an EXPORT_INTERFACE block) or local (contained with LOCAL_INTERFACE).
- A flag to indicate if the object is declared in a C++ file.
- The name of the file in which the object was declared.
- The complete text of any block comment that preceeds the declarations.
- If the declaration occurred inside a preprocessor conditional (``#if'') then the text of that conditional is provided.
- The complete text of a declaration for the object.

The exact output format will not be described here. It is simple to understand and parse and should be obvious to anyone who inspects some sample output.

5.0 Compiling The Makeheaders Program

The source code for makeheaders is a single file of ANSI-C code, less than 3000 lines in length. The program makes only modest demands of the system and C library and should compile without alteration on most ANSI C compilers and on most operating systems. It is known to compile using several variations of GCC for Unix as well as Cygwin32 and MSVC 5.0 for Win32.

6.0 Summary And Conclusion

The makeheaders program will automatically generate a minimal header file for each of a set of C source and header files, and will generate a composite header file for the entire source file suite, for either internal or external use. It can also be used as the parser in an automated program documentation system.

The makeheaders program has been in use since 1994, in a wide variety of projects under both UNIX and Win32. In every project where it has been used, makeheaders has proven to be a very helpful aid in the construction and maintenance of large C codes. In at least two cases, makeheaders has facilitated development of programs that would have otherwise been all but impossible due to their size and complexity.