C Tools and Compile-Time Environment

I have to say thanks to Jason Maasson from Frije Universiteit Amsterdam for his excellent script and slides. I've translated most of the slides and added some graphics and text.I would also like to thank Marshall Brain, founder of "www.howstuffworks.com" for his wonderful article on "How C Programming Works" which explains also the c-runtime environment.And last but not least Steven Simpson from Lancaster University for pointing out the differences between both languages on a few excellent pages.Look at the resource section at the end for links.

Introduction

Goals

1. **Learn how a C program gets compiled (The flow from preprocessor through compiler to linker and archiver)**
2. **Understand the necessary tools (Important compiler options, how to generate assembly code etc.)**
3. **learn how to cope with problems (debugger etc.)**
4. **Learn important libraries (network programming, what's in the C library etc.)**

Roadmap

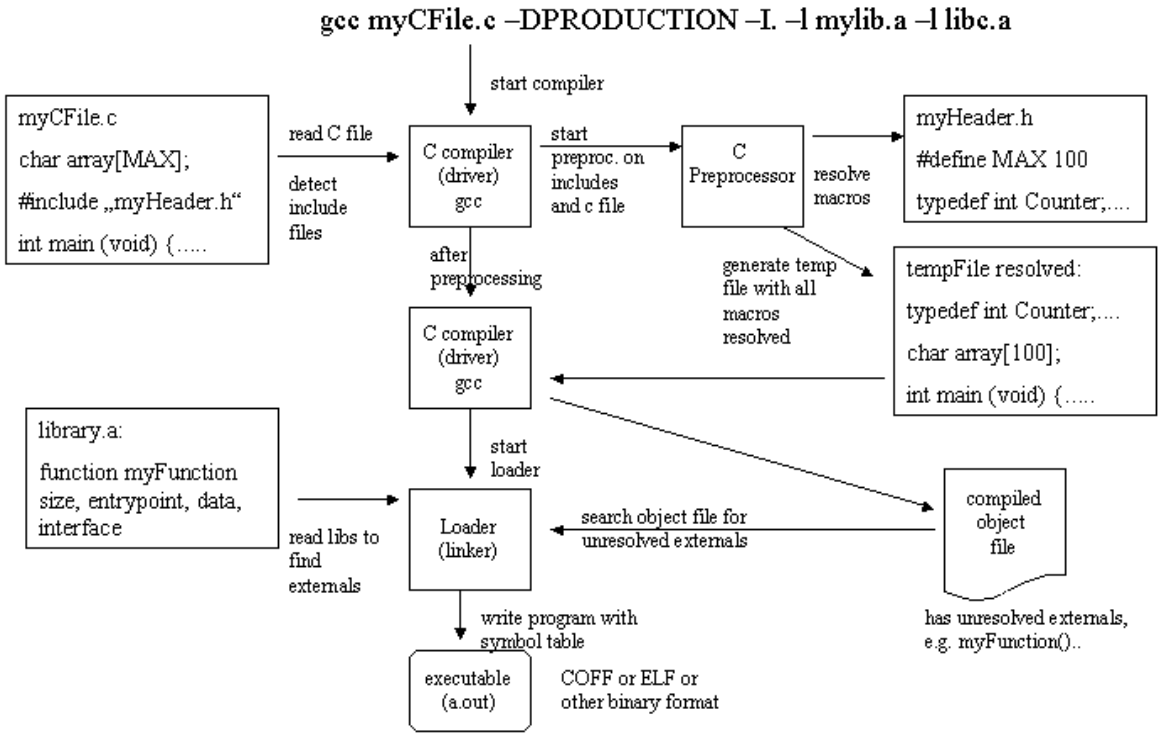
1. **Overview of C processing**
2. **The preprocessor and how it is used for portability**
3. **The C compiler: generating object code or assembler output**
4. **The linker: combining object code and libraries into an executable**
5. **C archives and libraries**
6. **Using Makefiles to control compilation**

Overview of C processing

When a file with C code (usually with a .c extension) needs to be compiled it runs through a number of tools. The main tool is called compiler (or driver) and it drives everything.

1. **First the compiler reads the C file into memory. If it detects include files it starts the C preprocessor which will read those files into memory as well.**
2. **Include files are macro files which need to be processed by a macro processor. This used to be a generic macro processor like m4 but is nowadays aware of C syntax.**
3. **Header files are now concatenated with the processed code from the .c file. Now the compiler can compile the whole code and the result is either a file with object code (a .o extension) or an executable program.**
4. **If the goal is an executable program, the compiler calls the linker to resolve unresolved externals (e.g. if your program code calls functions from other libraries or other files (modules) in your program). Once the linker is done an executable file has been generated.**
5. **In case of the object code (.o file) we now have a component that can go into an archive waiting to be linked together with other object code to form a new executable program.**
6. **Sometimes linking does not produce a big self-contained executable but an executable that needs so called Dynamic Link Libraries (windows: .dll) or Shared Object libraries (Unix: .so) at runtime.**

Diagram of C processing



Generating an executable is a complex process driven by the compiler. The compiler driver lets arguments and intermediate results flow between tools like preprocessor or linker. The final executable is written in a format that the Operating System understands. The format e.g. defines where the program entry point is, what data and stack are etc.

Components of a C program

At compile time a C program usually consists of three kinds of components:

Header or Include Files	**They contain definitions of constants or declarations of functions. Sometimes complete functions are implemented just as a macro, i.e. the code of the macro gets inserted whereever the macro is used. Header files collect information that is useful for more than one C source code file. To avoid duplicated informations with the associated maintenance problems this information is kept in central header files.**
C Files	**A C program can consist of one more more files with .c extension. Only one file can have a "main" function. The C files usually import ("include") header files with common declarations. There is no C rule for organizing the functions across files but usually related functions (e.g. those working on a common data structure or those forming a certain higher level function are kept together in one file which is called a module.**
Libraries	**Libraries contain C function that are already compiled, so called object code. Most C programs do not implement everything they need by themselves. Instead, they use libraries like libsocket.a for network communication or libc.a the standard C library for input/output handling, string handling etc. C libraries are usually**

Components of a C program (Continued)

compatible across compilers. A special form of library are the above mentioned dynamic link libraries where at linktime only a stub for each function is linked to the program and the real function is accessed through this stub at runtime.

Comparing C with Java Components

Some things in Java work similiar to C and some others are radically different.

Header files	**Java does not have a preprocessor nor does it separate declarations of classes or constants into separate files. But Java still has the need to learn declarations of classes in other java files when a new java file is compiled - if those classes should be used in the new file. The "import" statement directs the Java compiler to existing classes. The Java compiler can then extract the declaration information directly from compiled classes (.class files). This is a different mechanism for the same purpose.**
Java Files	**A class per file is the usual Java rule. This is like the module concept of C except that C does not enforce it.**
Libraries	**Java libraries are collections of classes and packages (which are again collections of classes in a separate namespace). All java external references are dynamic: the code referenced is not included in the compiled program. Instead, the Java classloader loads classes at runtime dynamically into the virtual machine. Note that the versioning problem of full dynamic link libraries has now turned into a versioning problem for individual classes.**

An example C header file

```
myHeader.h:
#ifndef myHeader_h    // protects header from being included
#define myHeader_h    // multiple times

#include "stdio.h"
#define MAXARRAY 100  // a constant
#define add(a,b)  a+b  // an inline function

#define PRODUCTION      // just sets PRODUCTION as true for ifdef
                       // could also be set via compiler arguments

#ifdef TEST            // a conditional compilation example
  #define FOO "bar"
#else
  #define FOO "foobar"
#endif

extern int myFunction(char*, int); // function prototype
typedef struct {     // declarations and type definitions
    int i;
    char* s;
} MyStruct;
#endif myHeader_h
```

The C Preprocessor

- **Removes comments of the form /* */ and //**
- **Reads preprocessing commands starting with the hash (#) sign. Examples are: #ifdef, #if, #elsif, #endif, #define**
- **Replace macros for constants with the real value: #define SIZE 100 becomes 100.**

The advantage of a preprocessing phase is e.g. that conditional compilation can take place. This means that code that is exclude with a #ifdef ....#endif bracket is NOT compiled and therefore does not show up in the final executable. This allows different code for production and test versions or different platforms.

Conditional compilation with C and Java

```
file.java:
private static final boolean DEBUG = false;  // this statement is now ALWAYS false
  if (DEBUG) {System.out.println(.....); }  // A good java compiler does NOT
                                             // compile this into bytecode.
```

```
#undef DEBUG
#ifdef DEBUG
     #define myPrint(a,b,c)  printf(a,b,c)
#else
     #define myPrint()
#endif
```

In both cases the code for the debug case should not be compiled. In the java case the compiler should recognize that the if statement is always false and therefore the code cannot be reached. This is of course an IMPLICIT mechanism relying on a good Java compiler. The C case is clear: it depends on the macro variable DEBUG. A statement like `#define a //nothing` will effectively just remove "a" from the source code and replace it with empty space.

A C File using the above header

```
in myCfile.c:
#include "myHeader.h"   // searches for include files
                        // in include path

char myArray[MAXARRAY] =100;  // allocates array of size 100

char* myString = "someString";

int main(int argc, char** argv)  {
  int result;
#ifdef TEST
  result = add(2,4);
#else
  result = myFunction(FOO,5);
#endif
  return 0;
}
```

Including header files

Typical C code uses the `#include "filename"` or `#include <filename>` syntax to include the header files with definitions into the program. What really happens is a COPY of the files referenced is placed at the top of the program, in the order of reference. You will need to take care that nothing is included twice if it would cause a problem. This is why header files are protected using the conditional compilation mechanism.

```
header.h:
  #ifndef HEADERFILENAME_H
  #define HEADERFILENAME_H
  .....
  #endif

cfile.c:
  #include "header.h"
....
  #include "header.h" // not included because HEADERFILENAME
                      // already defined from first include
```

Standard C include files

The header files belonging to a standard C environment are usually placed in `/usr/include/..` directories. They are referenced from within a C file using `<` and `>` brackets. The following lists popular header files and what they contain.

stddef.h	**Macros and type declarations**
stdlib.h	**access to environment, memory allocation (malloc..), utilities**
stdio.h	**Streaming input and output of characters**
string.h	**String function prototypes**
ctype.h	**Character handling (upper/lower case, alpha/numeric etc.)**
wchar.h	**wide characters for internationalization (I18n) and localization**
wctype.h	**like ctype only for wide characters**
limits.h	**says how big integer types are on THIS platform**
float.h	**implementation limits for floating-point numbers**
math.h	**math functions**
assert.h	**diagnose problems using assert statements**
errno.h	**many error codes. check it if you receive an error code from a program**
locale.h	**important functions and constants for localization of programs (language codes, message catalog functions)**

Standard C include files (Continued)

stdarg.h **support for functions with variable arguments**

signal.h **run-time exceptions sent by the kernel**

Usually there are also system related include files under /usr/include/sys which contain device descriptions, kernel structure descriptions etc.

Note

These header files list a lot of function prototypes. Those prototypes are program code (object code) and NOT contained in the header files. They are only declared there. The functions themselves live in libraries which can be found under /usr/lib on many platforms.

Program code after preprocessing

Note that the comments are removed, some source code pathes have been eliminated, constant values are resolved.

```
in intermediate file:
typedef struct {
    int i;
    char* s;
} MyStruct;

char myArray[100];  // allocates array of size 100

int main(int argc, char** argv)  {
  int result;
  result = myFunction("foobar",5);
  return 0;
}
```

The compilation step

```
gcc -c file.c creates an object file file.o
gcc file.c -lm would create an executable a.out. If file.c uses math functions (e
```

-c	**do not link yet. The resulting object code file can be put into an archive (library) and later linked into a program.**
-I<dir>	**look for include files also in <dir> Allows you to install e.g. a cross-compilation environment for Lego Robots on your system and generate code for this platform by directing the C compiler to use different header files.**
-l<lib>	**link the library <lib> to the program. If you have unresolved external errors look for the library which contains the necessary functions. You can use the `nm <libxxxx.a> to search the libraries for those functions.a`**
-L<dir>	**look for libraries also in <dir>**
-O	**Optimize code (usually differnt subparameters). Optimization can introduce subtle bugs or require that all libraries have been compiled the same way. That's why this is optional.**
-g	**Create debugging code (needed e.g. for symbolic debugging)**
-pg	**Create profiling code which tells you how much processing time functions**

The compilation step (Continued)

	take. Required to optimize your code properly.
-S	**Tell compiler to stop after producing assembler code. Let's you see how your code looks in assembler. Useful for debugging or device programming.**

The code after compilation but before linking

Most C compilers generate object code (processor dependent machine code) directly. It is also possible to first generate assembler code and have the assembler generate the object code. Tools like "nm" let you view the contents of an object file.

```
nm myCfile.o
00000000 b .bss
00000000 d .data
00000000 t .text
         U ___main
         U __alloca
00000012 T _main
00000000 D _myArray
         U _myFunction
00000064 D _myString
```

Note the three memory segments of a program: text, data, bss (heap is dynamically allocated during runtime). This object file is not yet executable because of the "U" parts: These are undefined external refernces. One is pretty clear: the function myFunction() has been used in myCfile.c but the code is not there. It must be in a library. It is in the next step - linking - that those unresolved externals are found and resolved. Java does this automatically at program start via the classloader. So all Java classes are in this senses "unfinished" what you will soon learn when external classes are not in the classpath and cannot be found at runtime, causing a ClassNotFoundException to be thrown.

Linking: Resolving the unresolved function

First the function has to be written and compiled into an object code file. Then it can be put into an archive (library) which is added to the compilers commandline.

in myfunction.c:

```
int myFunction(char*, int) {
  return 1;
}
```

This file gets compiled with `gcc -c myfunction.o` which looks like this when viewed with `nm myfunction.o : 00000000 b .bss 00000000 d .data 00000000 t .text 00000000 T _myFunction` . Note that myFunction now as a "T" in front meaning that it is defined and resides in the text segment of a program, currently at address 0

Creating a library

So far we have two .o files with object code. One has a main function and will become our program. The other one contains only one function which is used by our program. We can now decide to create a library and store all our helper functions there. This is done with `ar myStuff.a myfunction.o` . Use `ar -t myStuff.a` to look at the contents of the archive myStuff.a

```
ar -t myStuff.a:

myfunction..
```

That's the only function in our library for now. But there where more undefined externals in myCfile.o: `U ___main U __alloca` . These are functions that live in the standard C library libc.a. The compiler (gcc) will automatically search for unresolved symbols in this library.

How to find unresolved functions in libraries

Beginning C programmers usually have a problem finding unresolved functions in libraries. Experience tells you e.g. that socket calls are in libsocket.a. But what if you don't know? You can use a combination of standard unix commands to locate those functions. The most useful utilities are "nm" and "grep" together with the powerful "find" command.

```
at the top of directories with libraries (files with .a extension):
  find . -name "*.a" -exec nm | grep yourUnresolvedFunction) {} \; -print
```

Look for the one line where your function name shows up with a "T" symbol in front. This means your functions code is here in this text segment of this library. Of course you can use a full-feaured Integrated Development Environment (IDE) but even then it is good to know how things really work if things go wrong.

Generating a complete program

All in one go

`gcc -myCfile.c -lmystuff.a` will generate myCfile.o and link it to both our own library myStuff.a (to get myfunction.o) and the default standard C library. This results in an executable file a.out being created. Our header file would have to be in a standard place to be found by gcc.

Incremental steps

```
gcc -c myCfile.c  -> generates myCfile.o
gcc -c myfunction.c  -> generates myfunction.o
gcc -o myProgram myCfile.o myfunction.o
            -> generates myProgram executable
```

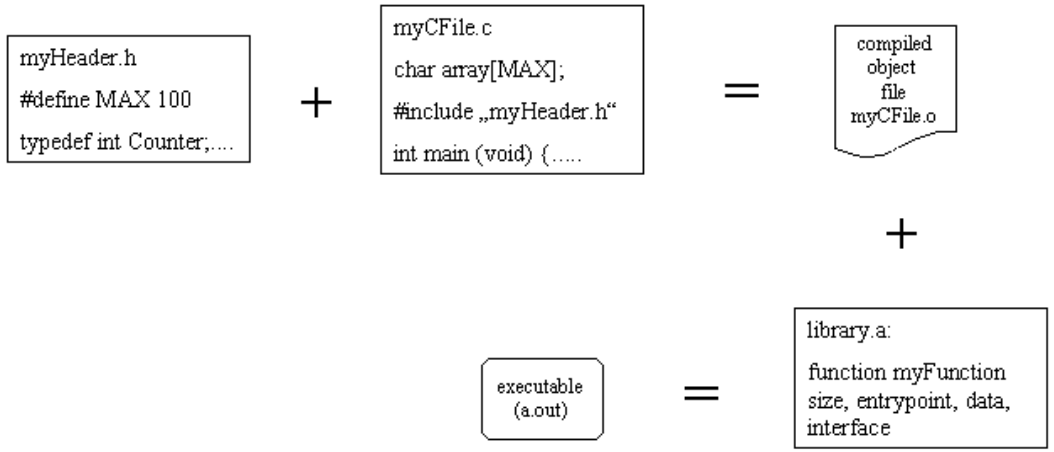
Object file formats

We have seen that a program consists of several different memory areas and it has a startup function (not main, but something similiar). The operating system needs to understand these things, that's why object file formats where invented. Those formats describe exactly the structure of a program and how it is started. The operating system will read the program file and perform the necessary steps to run the program.

Use `objdump -x myProgram.exe` to get a feeling for the information contained in object files.

A look at /usr/include/elf.h shows you the structures of an object file in ELF format. Compiler tools etc. need to know these structures.

Source Code Dependencies



All involved artifacts can change independently. A header file change may force a complete recompile. A C file change as well. A change in the library may force a re-linking of the program and a new executable. This means there are LOGICAL dependencies and TIME related dependencies. A tool that knows the logical relations between those artifacts can look at the timestamps and bring everything into a consistent state. This is what makefiles do.

Rules for compilation

By now it should be clear that C-programming involves sequential processing of several different software artifacts. .C files import .h files. .o files go into libraries (.a files). .S files need to become .o files. a.out programs depend on all those .xxx files. We can derive the following rules:

1. **If a header file changes, all C files including this header need to be recompiled to object files or executables.**
2. **If a C file changes it depends on whether it is part of a library (in other words: it offers public functions to other modules) or not. If it is independent then all executables which use it should be removed and recompiled**
3. **If the c file is part of a library the library needs to be updated with the new object code and ALL executables which use this library need to be generated.**

Add Team development

It is already hard for an individual developer to keep track of those dependencies. But once you develop as part of a team several even more critical issues arise:

- **You might use header files from a colleague which change frequently. How do you know when to recompile?**
- **When you integrate your parts with those of your colleagues to create the final product it turns out that the parts do not fit together. Functions are defined in several different incompatible ways (static integration problems). At runtime the program crashes. It takes days to find the problems which turn out to be caused by incompatible compiler options (alignment etc.) or duplicate dynamic link libraries installed in different locations. The program just takes the first - perhaps outdated library - and crashes.**

Note

Makefiles do not solve ALL those problems. It takes a full-blown build environment with generated makefiles for this. But makefiles are a start.

Makefile syntax

```
file myMakefile:
  # This is a comment
  # myprogram is built from file1.c file2.c and headerfile file.h

  # set compile variables for tools and environment
  # use gnu compiler
  CC = gcc
  # use full warnings
  CFLAGS = -Wall
  OBJS = file1.o file2.o
  HEADERS = file.h

  # let make now that objs depend on headers. make knows what to do then.
  $(HEADERS): $(OBJS)
  # the space before $(CC) MUST BE A TAB!!!
  myprogram: $(OBJS)
      $(CC) -o myprogram $(OBJS)
```

Use a Makefile

Try the makefile script. Let file1.c include file.h. Then use `touch file.h` to change the date of your header file to a newer date than the dependent .c files. Now issue `make -f myMakefile` and you should see gcc recompile file.c and create the executable myprogram.

```
 make -f myMakefile
 gcc -Wall -c file1.c
 gcc -Wall -c file2.c
 gcc -o myprogram file1.o file2.o

make myprogram
 make: 'myprogram' is up to date.

  touch file.h
  make myprogram
  gcc -Wall -c file1.c
  gcc -o myprogram file1.o file2.o
```

Program crashes and core files

On Unix platforms if a program crashes a so called "core file" is created. It contains the memory status of the program when it crashed. This information can be used for post-mortem debugging using a debugger like gdb.

For best debugging results a program needs to be compiled with debugging turned on. This creates additional debugging instructions in the object code of the program and preserves all symbol and line information. This allows a debugger to show the proper source code pieces.

`man gcc` lists a large number of debugging and optimization options. Some compilers do not allow mixing debugging and optimization.

Generating assembler code

Sometimes when you are working on a device driver or some kernel functions - or just if you need to speed-up one small function in your program you might want to create the function in assembler code. Writing assembler is hard but you can let the C compiler do the hard stuff for you and then just optimize the result. Sometimes you might hit an optimizer bug which forces you to look at the generated code to see where the problem is - even compilers have bugs.

```
gcc -S myCfile.c
```

produces assembler output.

Resources

The resources cover freely available information as well as excellent books right to the topic. All entries are commented to let you know what a paper or book is all about. I also expect participants to use this literature in case of questions.

Open Source Information on C programming

The following information is freely available and taken together is an excellent introduction to the subject.

1. **Jason Maassen, C for Java Programmers. A complete introduction to C for Java people. I have used and extended his slides for this lecture but you should read the background information here as well.60+ pages. Good if you need to prepare for a test or need some more information about a feature from my slides. Find his C course with other materials here: http://www.cs.vu.nl/~jason/course.html [http://www.cs.vu.nl/~jason/course.html].**
2. **Steven Simpson, Learning C from Java. An experienced Java programmer will get the most from this short paper focussing on the differences. Excellent. http://www.comp.lancs.ac.uk/computing/users/ss/java2c/diffs.htm [http://www.comp.lancs.ac.uk/computing/users/ss/java2c/diffs.html].**
3. **Marshal Brain, How C programming works. Another excellent paper from www.howstuffworks.com. This really explains complicated memory problems using pointers, how array overwriting can happen etc. And many useful pointers to other computing related topics like memory organization, operating systems etc. www.howstuffworks.com**

Books

I always try to have all recommended books available in our library. Also take a look at my special section there where I collect books which should be present at all times.

1. **Kernighan/Ritchie, Programming in C. The bible of c-programming from the inventors. A classic text. Very short- compare this to nowadays documentation bloat.**
2. **The C pocket reference. A short book covering the latest developments in C. In my library.**
3. **The C Puzzle Book. Alan R. Feuer. A very short and nice book full of examples with C.**

Code Repositories

I always use kind of code repositories for my work. I do NOT start with an empty page writing a program. Instead, I try to find example code that works and then adjust it to my purposes. It takes a long time to write something from scratch - you have to remember every detail about APIs etc. Take something that works and change it.

1. **David Flanagen, Java by Example. The best java examples in source I have found. Need to know how to create a file or write to a socket? Go there. All examples are downloadable from his web site.**
2. **Linux source tree. Download and install a linux source tree with kernel and utility source. There is plenty of C code to browse and search.**