An Introduction to GCC: for the GNU Compilers gcc and g++
读书笔记

Chapter 2:

1. Compile a C program
   a) **gcc -Wall hello.c -o hello**
      i. **–Wall: show all warning & error. It is recommended that you always use this!**
      ii. **–o: This option is usually given as the last argument on the command line. If it is omitted, the output is written to a default file called '***.out'.**
   b) Error debugging:
      i. The messages produced by GCC always have the form **file: line-number: message.**

2. Compiling multiple source files
   a) Difference between **#include "FILE.h"** and **#include <FILE.h>**
      i. **#include "FILE.h"** Searches in the current directory first then the system header file directory.
      ii. **#include <FILE.h>** only searches in the system header file directory.
   b) **gcc -Wall main.c hello_fn.c -o newhello**
      i. Note that hello.h will be included automatically.

3. Compiling files independently
   a) Why: If a program is stored in a single file then any change to an individual function requires **the whole program to be recompiled** to produce a new executable.
   b) The source files are **compiled separately** and then **linked together**—a two stage process.
      i. First stage: .o, an object file.
      ii. Second stage: the object files are merged together by a separate program called the **linker.**
   c) Creating object files from source files
      i. **gcc -Wall -c main.c**
      -c: compile a source file into an object file
      ii. Containing the machine code for the main function, has an external reference to hello(), but the **exact memory address** is not decided.
      iii. **No** need to **add –o option**
      iv. **No** need to **put the header file 'hello.h' on the command line,**
   d) Creating executables from object files
      i. **gcc main.o hello_fn.o -o hello**
      ii. **No** need to use **the '-Wall' warning option**: file has been compiled.
      iii. Fails only if there are references which cannot be resolved
   e) Link order of object files
      i. **gcc main.o hello_fn.o -o hello**
      ii. Object file which **contains the definition of a function** should **appear after** any files which **call that function.**
   f) Recompiling and relinking
      i. If the **prototype of a function** has changed, it is necessary to modify and recompile all of the other source files which use it.

    ii. Linking is faster than compilation

g) Linking with external libraries
    i. The most common use of libraries is to **provide system functions.**
    ii. Libraries are typically stored in special archive files with the extension '.a', referred to as **static libraries**
    iii. Using ar command to create a static library
    iv. The default library file is lib.a, when we referred other function, such as math.h, we will need **libm.a**
    v. **gcc -Wall calc.c -lm -o calc**
    vi. The compiler option **'-lNAME'** will attempt to link object files with a library file **'libNAME.a'** in the standard library directories.

h) Link order of libraries
    i. **Containing the definition of a function** should appear after **any source files or object files which use it.**
    ii. **A library which calls an external function defined in another library** should appear before the **library containing the function.**
    iii. **gcc -Wall data.c -lglpk -lm**
    libglpk.a calls functions in libm.a

i) Using library header file: If you wrongly omit the include header file, you can only detect this error by **–Wall**

Chapter 3: compilation options

1. Setting search paths
    a) By default, gcc searches
        i. The following directories for **header files:**
            1. /usr/local/include/
            2. /usr/include/
        ii. The following directories for **libraries:**
            1. /usr/local/lib/
            2. /usr/lib/
        iii. A header file found in '/usr/local/include' **takes precedence over** a file with the same name in '/usr/include'.
        iv. The compiler options **'-I' and '-L'** add new directories to the beginning of the include path and library search path respectively.
    b) Search path example
        i. **gcc -Wall -I/opt/gdbm-1.8.3/include -L/opt/gdbm-1.8.3/lib dbmain.c –lgdbm**
        -I: add new directory to the header file search set.
        -L: add new directory to the library search set.
        ii. **You should never place the absolute paths of header files in #include statements in your source code, as this will prevent the program from compiling on other systems**
    c) Environment variables
        i. Set in **.bash_profile**
        ii. Additional directories can be added to the include path using the environment

variable **C_INCLUDE_PATH (for C header files)** or **CPLUS_INCLUDE_PATH (for C++ header files).**

iii. **C_INCLUDE_PATH=/opt/gdbm-1.8.3/include**
C_INCLUDE_PATH=.:/opt/gdbm-1.8.3/include:/net/include
export C_INCLUDE_PATH
LIBRARY_PATH=/opt/gdbm-1.8.3/lib
export LIBRARY_PATH

d) Extended search paths
   i. **gcc -I. -I/opt/gdbm-1.8.3/include -I/net/include -L. -L/opt/gdbm-1.8.3/lib -L/net/lib**
   Add multiple search paths

e) Exact search order.
   i. **Command-line options '-I' and '-L', from left to right**
   ii. **Directories specified by environment variables, such as C_INCLUDE_PATH and LIBRARY_PATH**
   iii. **Default system directories**

2. Shared libraries and static libraries
   a) Static libraries
      i. **'.a' files**
      ii. When a program is linked against a static library, the **machine code** from the object files for any external functions used by the program **is copied from the library into the final executable**
   b) Shared libraries
      i. **extension '.so'**
      ii. An executable file linked against a shared library contains only **a small table** of the functions it requires.
      iii. Before the executable file starts running, the machine code for the external functions is copied into memory from the shared library file on disk by the operating system
      iv. Dynamic linking makes executable files **smaller and saves disk space**, because one copy of a library **can be shared between multiple programs**.
      v. Shared libraries make it possible to update a library **without recompiling the programs which use it**
   c) gcc compiles programs to **use shared libraries by default** on most systems, if they are available
   d) By default the loader searches for shared libraries **only in a predefined set** of system directories, **such as '/usr/local/lib' and '/usr/lib**
   e) **LD_LIBRARY_PATH=/opt/gdbm-1.8.3/lib**
      export LD_LIBRARY_PATH
   f) **gcc -Wall -static -I/opt/gdbm-1.8.3/include/ -L/opt/gdbm-1.8.3/lib/ dbmain.c -lgdbm**
      -static: force static linking
   g) **gcc -Wall -I/opt/gdbm-1.8.3/include dbmain.c /opt/gdbm-1.8.3/lib/libgdbm.so**
      Link directly with individual library files by specifying the full path to the library

on the command line

3. C language standards
   a) ANSI/ISO
      i. **gcc -Wall -ansi ansi.c**

         -ansi: use ansi. This allows programs written for ANSI/ISO C to be compiled without any unwanted effects from GNU extensions.
   b) Strict ANSI/ISO
      i. **gcc -Wall -ansi -pedantic gnuarray.c**

         -pedantic: gcc to reject all GNU C extensions, not just those that are incompatible with the ANSI/ISO standard.
      ii. **This helps you to write portable programs which follow the ANSI/ISO standard.**
   c) Selecting specific standards
      i. **'-std=c89' or '-std=iso9899:1990'**

         Std9899
      ii. **'-std=iso9899:199409**

         Add internationalization support: language
      iii. **'-std=c99' or '-std=iso9899:1999'**

         Std99
      iv. **'-std=gnu89' and '-std=gnu99'.**

         c language standard + GNU extension
   d) Warning options in –Wall
      i. '-Wcomment' (included in '-Wall')

         This option warns about nested comments.
      ii. '-Wformat' (included in '-Wall')

         This option warns about the incorrect use of format strings in functions such as printf and scanf, where the format specifier does not agree with the type of the corresponding function argument.
      iii. '-Wunused' (included in '-Wall')

         This option warns about unused variables.
      iv. '-Wimplicit' (included in '-Wall')

         This option warns about any functions that are used without being declared. **The most common reason for a function to be used without being declared is forgetting to include a header file.**
      v. '-Wreturn-type' (included in '-Wall')

         This option warns about functions that are defined without a return type but not declared void.
   e) Additional warning options
      i. They are not included in '-Wall' because they only indicate possibly problematic or "suspicious" code
      ii. It is more appropriate t**o use them periodically and review the results**, checking for anything unexpected, or to enable them for some programs or files.
      iii. **'-W'**

This is a general option similar to '-Wall' which warns about a selection of common programming errors

In practice, the options '-W' and '-Wall' are normally used together.

iv. **'-Werror'**

Changes the default behavior by converting warnings into errors, stopping the compilation whenever a warning occurs.

v. '-Wconversion'

This option warns about implicit type conversions that could cause unexpected results. For example: unsigned int x = -1;

vi. '-Wshadow'

This option warns about the redeclaration of a variable name in a scope where it has already been declared.

vii. '-Wcast-qual'

This option warns about pointers that are cast to remove a type qualifier, such as const.

viii. -Wtraditional'

This option warns about parts of the code which would be interpreted differently by an ANSI/ISO compiler and a "traditional" pre-ANSI compiler.

Chapter 4: Using the preprocessor

1. Defining macros
   - **a) gcc -Wall -DTEST dtest.c**

     -DTEST: define a macro called TEST and assign 1 as its value.

     The gcc option **'-DNAME**' defines a preprocessor macro NAME from the command line.
2. Origin of the macros
   - a) Specified on the command line with the option '-D',
   - b) In a source file (or library header file) with #define.
   - c) Automatically defined by the compiler—these typically use a reserved namespace beginning with a double-underscore prefix '__'.
3. Macros with values
   - a) This value is inserted into the source code at each point where the macro occurs
   - **b) Note that macros are not expanded inside strings**
   - **c) gcc -Wall -DNUM=100 dtestval.c**

     The '-D' command-line option can be used in the form '-DNAME=VALUE'.
   - **d) Note that it is a good idea to surround macros by parentheses whenever they are part of an expression.**
   - **e) A macro can be defined to a empty value using quotes on the command line, -DNAME=""**
4. Preprocessing source files
   - **a) gcc -E test.c**

     -E option: Use C preprocessor to deal with the source file.
   - b) The preprocessor also inserts lines recording the source file and line numbers in the form # line-number "source-file", to aid in debugging. Do not affect the program

itself

c) Example:

Source file:

```
#include <stdio.h>
Int main (void)
{
    printf ("Hello, world!\n");
    return 0;
}
```

Preprocessed file:

```
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
extern int fprintf (FILE * __stream,
const char * __format, ...) ;
extern int printf (const char * __format, ...) ;
[ ... additional declarations ... ]
# 1 "hello.c" 2
int
main (void)
{
    printf ("Hello, world!\n");
    return 0;
}
```

d) gcc -c -save-temps hello.c

-save-temps: saves '.s' assembly files and '.o' object files in addition to preprocessed '.i' preprocessed files.

Chapter 5: Compiling for debugging

1. '-g' debug option
    a) Store additional debugging information in object files and executables.
    b) This debugging information allows errors to be traced back from a specific machine instruction to the corresponding line in the original source file.
    c) It also allows the execution of a program to be traced in a debugger
    d) Using a debugger also allows the values of variables to be examined while the program is running.
2. Examining core files
    a) Whenever the error message 'core dumped' is displayed, the operating system should produce a file called **'core'** in the **current directory** which contains the in-memory state of the program at the time it crashed.
        i. The line where the program stops
        ii. The values of the variables at that time.

b) **gcc -Wall -g null.c**
   Add –g to trace the fail error.
   gdb EXECUTABLE-FILE CORE-FILE
   Core files can be loaded into the GNU Debugger gdb. Noted that it is not possible to debug a core file without the corresponding source file.
   (gdb) print VARIABLE_NAME

c) Displaying a backtrace
   (gdb) backtrace

```
#0 0x080483ed in a (p=0x0) at null.c:13
#1 0x080483d9 in main () at null.c:7
```

   Show the function calls and arguments up to the current point of execution.

d) **It is possible to move to different levels in the stack trace, and examine their variables**, using the debugger commands **up** and **down**

Chapter 6: Compiling with optimization

1. **GCC is an optimizing compiler**. It provides a wide range of options which aim to **increase the speed**, or **reduce the size**, of the **executable files it generates**.
2. Source-level optimization
   a) Common subexpression elimination
      i. **Computing an expression in the source code with fewer instructions, by reusing already-computed results.**
      ii. **Example:**
      x = cos(v)*(1+sin(u/2)) + sin(w)*(1-sin(u/2))
      t = sin(u/2)
      x = cos(v)*(1+t) + sin(w)*(1-t)
   b) **Function inlining** eliminates this overhead by replacing calls to a function by the **code of the function itself** (known as placing the code in-line).
      i. It can become significant only when there are functions which contain relatively few instructions
      ii. Inlining is always favorable if there is only one point of invocation of a function
   c) Example:

```
for (i = 0; i < 1000000; i++)
{
    double t = (i + 0.5); /* temporary variable */
    sum += t * t;
}
```

   Eliminating the function call and performing the multiplication in-line allows the loop to run with maximum efficiency
   d) The **inline** keyword can be used to request explicitly that **a specific function should be inlined wherever possible**, including its use in other files.
3. Speed-space tradeoffs
   a) Optimizations with a speed-space tradeoff can also be used to make an executable smaller, at the expense of making it run slower.
   b) **Loop unrolling**

more efficient way to write the same code is simply to unroll the loop and execute the assignments directly:

```
y[0] = 0;
y[1] = 1;
y[2] = 2;
y[3] = 3;
y[4] = 4;
y[5] = 5;
y[6] = 6;
y[7] = 7;
```

c) Loop unrolling is also possible when the upper bound of the loop is unknown, provided the start and end conditions are handled correctly

```
for (i = 0; i < (n % 2); i++)
{
        y[i] = i;
}
for ( ; i + 1 < n; i += 2) /* no initializer */
{
        y[i] = i;
       y[i+1] = i+1;
}
```

4. Scheduling: The lowest level of optimization
   a) Compiler determines the best ordering of individual instructions
   b) instructions must be arranged so that their results become available to later instructions at the right time, and to allow for maximum parallel execution.
   c) +speed +size
5. Optimization levels
   a) Gcc provides a range of general optimization levels, **numbered from 0–3**, as well as individual options for specific types of optimization. For most purposes it is satisfactory to **use '-O0' for debugging,** and '**-O2' for development and deployment.**
   b) '-O0' or no '-O' option (default)
   GCC does not perform any optimization and compiles the source code in the most straightforward way possible.
   c) '-O1' or '-O'
   This level turns on the most common forms of optimization that do not require any speed-space tradeoffs.
   d) '-O2'
   This option turns on further optimizations including instruction scheduling. Only optimizations that do not require any speed-space tradeoffs are used, so the executable should not increase in size.
   e) '-O3'
   This option turns on more expensive optimizations, such as function inclining, in addition to all the optimizations of the lower levels 'O2' and '-O1'.
   f) '-funroll-loops'

This option turns on loop-unrolling, and is independent of the other optimization options.

g) **'-Os'**

This option selects optimizations which reduce the size of an executable. In some cases a smaller executable will also run faster, due to better cache usage**.**

6. Optimization and debugging
   a) When optimization is turned on, GCC can produce additional warnings that do not appear when compiling without optimization
   b) Data-flow analysis: the compiler examines the use of all variables and their initial values. **It can detect the use of uninitialized variables**.
   c) **'-Wuninitialized'**

      This option warns about variables that are read without being initialized, but only show when carrying out optimization.

Chapter 7: Compiling a C++ program

1. **Same as gcc except that g++ is used here.**
2. Using C++ standard library templates
   a) In addition to the template classes provided by the C++ standard library you can define your own templates.

Chapter 8: CPU specific compilation: omitted here

Chapter 9: Troubleshooting

1. Help for command-line options
   a) **gcc -v –help**

      gcc -v --help 2>&1 | more
2. Version numbers
   a) **gcc –version**
3. –v: Verbose compilation
   a) **gcc -v -Wall hello.c**

Chapter 10: Compiler-related tools

1. Creating a library with the GNU archiver
   a) **ar cr LIB_NAME OBJECT_FILENAME1 OBJECT_FILENAME2…**

      cr: create and replace

   b) **ar t LIB_NAME**

      A "table of contents" option 't' to list the object files in an existing library

   c) **gcc -Wall main.c libhello.a -o hello**
2. Using the profiler gprof
   a) A useful tool for measuring the performance of a program
   b) It records the number of calls to each function and the amount of time spent there, on a per-function basis.
   c) To use profiling, the program **must be compiled and linked with the '-pg' profiling option:**

gcc -Wall -c -pg collatz.c
gcc -Wall -pg collatz.o
gprof a.out
If the program consists of more than one source file then the '-pg' option should be used when compiling each source file, and used again when linking the object files to create the final executable

3. Coverage testing with gcov
    a) The GNU coverage testing tool gcov **analyses the number of times each line** of a program is executed during a run
    b) Combined with profiling information from gprof the information from coverage testing allows efforts to speed up a program to be concentrated on specific lines of the source code.
    c) gcc -Wall -fprofile-arcs -ftest-coverage cov.c
    The executable must then be run to create the coverage data, the data from the run is written to several files with the extensions '.bb', '.bbg' and '.da' respectively in the current directory.
    gcov cov.c
    Lines which were not executed are marked with hashes '######'

Chapter 11: How the compiler works

1. preprocessing (to expand macros)
    i. cpp hello.c > hello.i
2. compilation (from source code to assembly language)
    i. gcc -Wall -S hello.i
    -S' instructs gcc to convert the preprocessed C source code to assembly language without creating an object file
3. assembly (from assembly language to machine code)
    i. as hello.s -o hello.o
4. linking (to create the final executable)

Chapter 12: Examining compiled files

1. Identifying files
    a) file a.out
    a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), not stripped
    ELF
    The internal format of the executable file (ELF stands for "Executable and Linking Format", other formats such as COFF "Common Object File Format" are used on some older operating systems (e.g. MS-DOS))
    32-bit
    The word size (for some platforms this would be 64-bit).
    LSB
    Compiled for a platform with least significant byte first word ordering, such as Intel and AMD x86 processors (the alternative MSB most significant byte first is used by

other processors, such as the Motorola 680x0)(1). Some processors such as Itanium and MIPS support both LSB and MSB orderings.

Intel 80386

The processor the executable file was compiled for

version 1 (SYSV)

This is the version of the internal format of the file.

dynamically linked

The executable uses shared libraries (statically linked indicates programs linked statically, for example using the '-static' option)

not stripped

The executable contains a symbol table (this can be removed with the strip command).

b) Examining the symbol table
   i. Stores the location of functions and variables by name, and can be displayed with the nm command
   ii. **nm a.out**
   iii. **The most common use of the nm command is to check whether a library contains the definition of a specific function, by looking for a 'T' entry in the second column against the function name.**

c) Finding dynamically linked libraries
   i. ldd examines an executable and displays a list of the shared libraries that it needs.
   ii. **ldd a.out**

```
libc.so.6 => /lib/libc.so.6 (0x40020000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

   iii. **The ldd command can also be used to examine shared libraries themselves, in order to follow a chain of shared library dependencies.**