G++ for beginners
Version 0.11.2, DRAFT

D. Vermeir
Dept. of Computer Science
Free University of Brussels, VUB
dvermeir@vub.ac.be

29 September, 2008

Abstract

This document is a very short tutorial on using the **GNU** C++ compiler (which is called g++).

A postscript version of this text is available in the file **gpp.pdf**.

Contents

1 Compiling and linking **3**

2 Using libraries **5**

2.1 Static libraries 5

2.2 Dynamic libraries 6

3 Using g++ **7**

3.1 Compiling with g++ 7

3.1.1 Finding #include files 8

3.1.2 Warnings, debugging 8

3.1.3 Optimization 8

3.2 Linking with g++ 9

3.2.1 Dynamic linking with g++ 9

3.2.2 Static linking with g++ 10

3.3 Creating libraries with g++ 10

3.3.1 Creating static libraries 10

3.3.2 Creating dynamic libraries 10

3.4 Miscellaneous 11

3.4.1 Preprocessing 11

3.4.2 Profiling 11

3.4.3 Generating make-dependencies 11

4 Summary of most popular options **13**

Index **14**

1 Compiling and linking

Figure 1 illustrates the relationship between the various files (indicated by oval boxes) that are needed to construct an executable program.

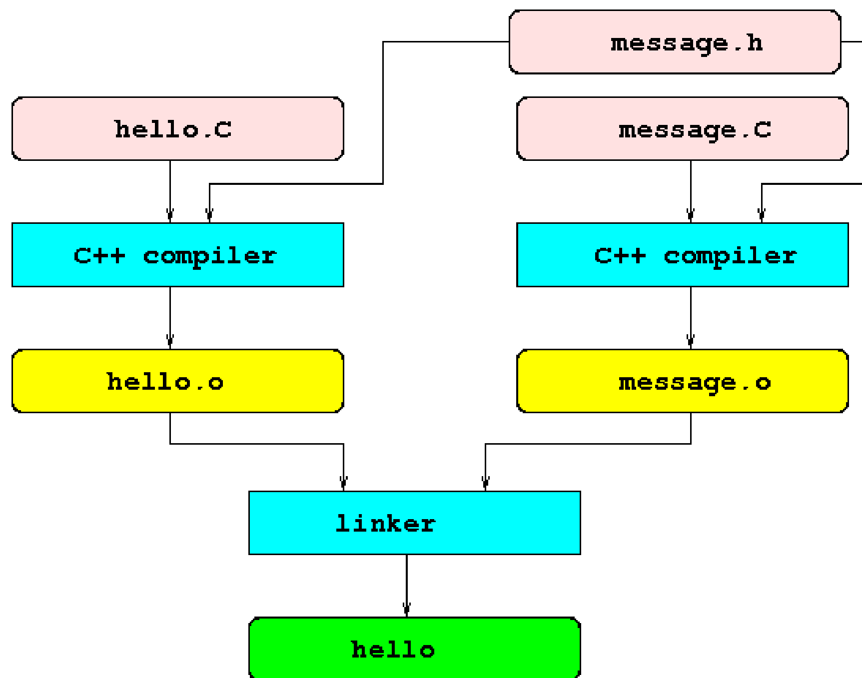

Figure 1: Compiling and linking

- The *source files*, in the example the files `hello.C` and `message.C`, contain C++ *source code* instructions written in the *source language*, in this case C++. Such instructions can be divided into two kinds:
  - *definitions*, which, roughly, cause the compiler to “output” something, e.g. assembler instructions or memory reservation for variables. E.g. the following C++ statements are definitions.

    ```
    int v;
    int f() { ... }
    ```

  - *declarations*, which only inform the compiler of the existence or properties of functions, variables, types etc., without causing any (immediate) output.

```
extern int v;
int f();
```

Definitions should only appear in `.C` files. *Header files* such as `message.h` should **only** contain declarations, **never** definitions. In the example, `message.h` contains a declaration of a symbol ("`message`") that is used but not defined in `hello.C`. On the other hand, `message.C` contains the *definition* of "`message`".

- The *object files*, in the example the file `hello.o` and `message.o` contain *machine language* instructions, possibly (very likely) with *unresolved references*. E.g. in the example, the file `hello.o` refers to an external variable symbol "`message`" that was declared but not defined in the source file `hello.C`[1]. Any reference to the "`message`" symbol in `hello.o` is unresolved.

- The *executable file*, also called *program*, contains fully linked machine language instructions where all references to function and variable symbols have been resolved by the *linker*. The linker combines several object files into one executable file and fills in all unresolved references in all these files, using information on what is available in the other files (or in libraries, see Section 2 below).

The various C++ source files are shown below.

```
1 // $Id: hello.C,v 1.2 1999/08/02 10:47:11 dvermeir Exp $
2 #include <iostream>
3 #include "message.h"
4
5 int
6 main(int argc,char *argv[]) {
7   std::cout << message << std::endl;
8 }
```

In `hello.C`, the "`#include`" directive on `message.h` ensures that "`message`" is declared so that the compiler knows its type etc.

```
1 // $Id: message.C,v 1.2 1999/08/02 10:47:11 dvermeir Exp $
2 #include        "message.h"
3
4 const std::string message("hello world");
```

Also `message.C` uses the "`#include`" directive on `message.h`. This is useful because, when compiling `message.C`, the compiler first sees a declaration

[1]The declaration of "`message`" was actually read from the include file `message.h`.

(from message.h) and then a definition of "message" which allows it to check the consistency of one with the other. Chaos would ensue if message.C would not include message.h and e.g. "message" was declared as an integer in message.h and defined as a string in message.C).

```
1 #ifndef MESSAGE_H
2 #define MESSAGE_H
3 // $Id: message.h,v 1.3 1999/08/07 08:54:50 dvermeir Exp $
4 #include <string>
5
6 extern const std::string message;
7 #endif
```

Note that the entire file message.h is conditionally included, using a preprocessor #ifndef...#endif directive. Together with the #define MESSAGE_H on the second line, this ensures that message.h will be included at most once in any compilation.

2 Using libraries

Looking at hello.C, we notice that it also includes (and uses) iostream. This file contains declarations of various standard library classes dealing with input/output. Thus, hello.o will also contain unresolved references to e.g. output functions. The linker resolves these references by consulting the standard C++ library, as shown in Figure 2.

2.1 Static libraries

In the example, the linker looked at the standard library in /usr/lib/libstdc++.a. (by convention, files with a ".a" suffix are *static libraries*). This means that the linker will copy the code (or data) corresponding to the unresolved references from the library file, and add them to the executable file. Simply put, the linker will extract those object files from the library (a library can be thought of as a collection of object files) that are (recursively) needed by the other files and add them to the resulting executable file. This process is called *static linking* (with libraries).

If, after linking, we were to remove the library file /usr/lib/libstdc++.a, the executable file hello would still work.

It is possible to create your own libraries, see Section 3.3.

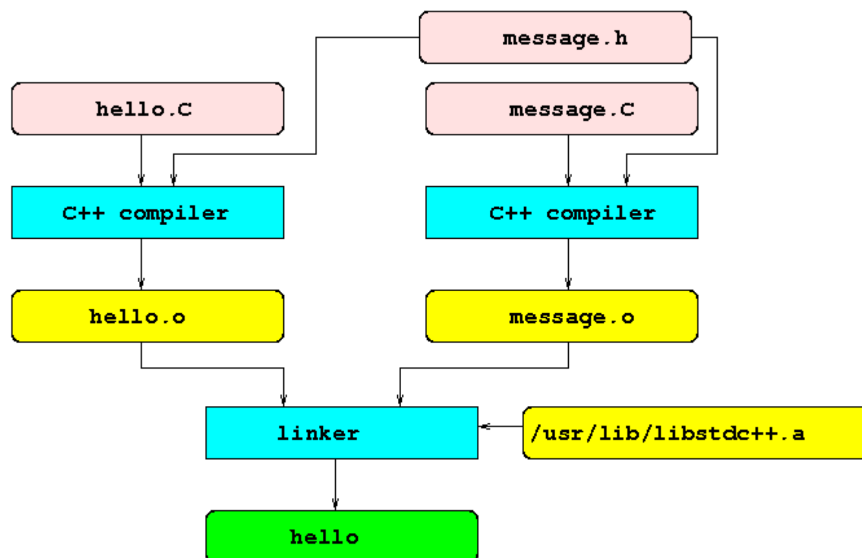

Figure 2: Compiling and linking with a library

2.2 Dynamic libraries

Static linking has the disadvantage that it increases the size of the executable file. E.g. almost every statically linked executable file contains its own copy of the C++ iostream code, which may use a considerable amount of disk space. Also, if a new version of the library is available, statically linked executable files will need to be linked again in order to use the new version of the library.

An alternative way to link with libraries is to use so-called *dynamic* (also called *shared*) libraries. Conventionally, dynamic library files have names ending with "`.so`", like `/usr/lib/libstdc++.so`. When linking with a dynamic library, the linker does not copy any objects; it just remembers which parts of which library are needed by the executable file. Thus, dynamically linked executable files tend to be smaller. At *run time*, i.e. when the *program*[2] is actually executed, the *dynamic loader* will step in and add the missing parts of the dynamic library to the program.

To find, at run time, the appropriate library, the dynamic loader (called *ld.so* under Linux) proceeds as follows:

1. First, it is possible to store a *runpath* in the executable file. This *runpath* is a list (much like the *PATH* shell variable) of directories, separated by

[2]In this text, we use "program" as a synonym for "executable file".

colons (“`:`”). If such a *runpath* exists, the dynamic loader will try to find any needed shared libraries in each of the *runpath* directories.

2. If the previous step fails, the dynamic loader will try to find the libraries in the directories mentioned in the *LD_LIBRARY_PATH* shell variable, if it exists. Also the *LD_LIBRARY_PATH* variable contains a list of directories, separated by colons (“`:`”). Note that the value of *LD_LIBRARY_PATH* is ignored for setuid/setgid programs.

3. Under Linux, try to find the libraries in */etc/ld.so.cache* which contains a compiled list of candidate libraries. See also the *ldconfig* command.

4. If any libraries remain to be found, the dynamic loader will look for them in the directories `/lib` and `/usr/lib`.

The procedure above is very flexible: e.g. if the runpath is not used, it is possible, by simply redefining *LD_LIBRARY_PATH*, to run the same program with a different version of a shared library.

You can used the **ldd** command to find out which shared libraries are needed by an executable file and where they would be found (using the current definition of *LD_LIBRARY_PATH*).

```
tinf2% ldd hello
    libstdc++.so.2.9.0 => /usr/lib/libstdc++.so.2.9.0
    libm.so.1 => /usr/lib/libm.so.1
    libc.so.1 => /usr/lib/libc.so.1
    libdl.so.1 => /usr/lib/libdl.so.1
tinf2% echo $LD_LIBRARY_PATH
/usr/lib:/usr/openwin/lib:
tinf2%
```

3 Using g++

G++ is used both for compiling and linking[3] In this section, we explain the most common options that are useful for compiling, linking (dynamic and static) and for producing libraries.

3.1 Compiling with g++

To produce an object file from a source file, use the *-c* option. E.g.

[3]Actually, when appropriate, *g++* arranges to call the operating system's native linker.

```
g++ -c hello.C message.C
```

will produce both `hello.o` and `message.o`.

3.1.1 Finding #include files

The *-I* option can be used to tell g++ where it may look for include files. E.g. if the source file contains

```
#include <tbcc/err.h>
```

where the full pathname of the file to be included is `/usr/local/include/tbcc/err.h`, one would use

```
g++ -c -I/usr/local/include hello.C
```

3.1.2 Warnings, debugging

It is good practice to use the *-Wall* option to tell g++ to warn about all suspicious or non-standard constructs. Strangely, to warn about suspicious conversions, you need to use, in addition, the *-Wconversion* option.

```
g++ -c -Wall -Wconversion hello.C
```

The *-g* option tells g++ to include in the object file extra debug information that is needed by run-time debuggers such as **gdb** or *ddd*.

```
g++ -c -g hello.C
```

3.1.3 Optimization

If necessary (and only then), the *-O*, *-O2* or *-O3* options can be used to trigger code optimization (*-O3* will optimize more than *-O2* which is itself “stronger” than *-O*). Of course, using these options will slow down compilations.

```
g++ -c -O2 hello.C
```

3.2 Linking with g++

To produce an executable file from some object files, use the *-o* option to tell g++ the name of the resulting program. E.g.

```
g++ -o hello hello.o message.o
```

will produce the executable file "`hello`".

If you need to link with libraries[4], use the *-L* and *-l* options. The *-L* option is followed by the directory of the library file while the *-l* option is followed by the identification of the library: use "`-labc`" to link with the library "`libabc.a`" (for static linking) or "`libabc.so`" (for dynamic linking).

3.2.1 Dynamic linking with g++

By default, g++ will use dynamic linking. E.g.

```
g++ -o hello hello.o -L/usr/local/lib -ltbcc -L/usr/local/lib/mysql
-lmysql
```

will produce the executable file `hello`, which will be dynamically linked with `libtbcc.so` and `libmysql.so`.

Note that, since we did not specify any runpath, at run time, the dynamic loader will attempt to find `libtbcc.so` and `libmysql.so` in a directory from the *LD_LIBRARY_PATH* variable.

To set the *runpath*:

- Under *Linux*, use the *-Wl,–rpath -Wl,dirs* option[5]:

  ```
  g++ -o hello hello.o -Wl,--rpath
  -Wl,$HOME/mylib:/usr/local/lib -L/usr/local/lib -ltbcc
  ```

  will ensure that, at run time, the dynamic loader will try to find `libtbcc.so` in `$HOME/mylib` or `/usr/local/lib` (note that `$HOME` is expanded before it is stored in the runpath).

- Under *Solaris*, use the *-R* option:

[4] g++ will automatically link with the C++ standard library.

[5] *-Wl,argument* passes `argument` to the linker, several arguments can be passed if they are separated by comma's as in

```
-Wl,rpath,/usr/local/lib
```

```
g++ -o hello hello.o -R$HOME/mylib:/usr/local/lib
-L/usr/local/lib -ltbcc
```

will ensure that, at run time, the dynamic loader will try to find `libtbcc.so` in `$HOME/mylib` or `/usr/local/lib`.

3.2.2 Static linking with g++

G++ can be forced to link all libraries statically by using the *-static* flag in front of the library specifications.
E.g.

```
g++ -o hello hello.o message.o -static -L/usr/local/lib
-ltbcc
```

will result in a "stand-alone" executable file.
To statically link some static libraries, just list them as you would any object file:

```
g++ -o hello hello.o message.o mylibs/mylib.a
mylibs/junk.a L/usr/local/lib -ltbcc
```

will include the needed objects from the static libraries `mylibs/mylib.a` and `mylibs/junk.a` into the resulting executable.

3.3 Creating libraries with g++

3.3.1 Creating static libraries

To create a static library, use the *ar* (archive) command to collect several object files into a library file. Don't forget to follow the naming convention: the name of the library should be of the form `libXYZ.a` where you can fill in `XYZ`. E.g.

```
ar cru libmine.a f1.o f2.o f3.o
```

will create a static library `libmine.a`.

3.3.2 Creating dynamic libraries

Dynamic libraries can only be created from object files that have been compiled using the *-fpic*[6] option. E.g.

[6] The *fpic* option causes g++ to create "position independent" code.

```
g++ -c -fpic f1.C f2.C f3.C
```

To actually create the library, it suffices to use the *-o* and *-shared* options.

```
g++ -shared -o libmine.so f1.o f2.o f3.o
```

3.4 Miscellaneous

3.4.1 Preprocessing

Use the *-E* option to limit g++ to the preprocessor. This will cause g++ to process `#include` and `#define` directives and put the resulting source file on standard output.

```
g++ -E message.C >message.e
```

3.4.2 Profiling

Use the *-pg* option to build an executable file that, when executed, will generate *profiling* information in a file `gmon.out`.

```
g++ -pg -g -o hello hello.C message.C
```

Use the **gprof** command to display this information in a textual format.

```
hello
gprof -C hello >hello.profile
```

3.4.3 Generating make-dependencies

Using the *-MM* option causes g++ to (only) generate a list of dependencies for all argument source files on standard output. This option is extremely useful in Makefiles such as the one below. The *-M* option has a similar effect but generates also dependencies on system-level include files (e.g. from `/usr/include`).

```
1  # $Id: Makefile,v 1.1 1999/12/21 13:02:17 dvermeir Exp $
2  CCFILES=        hello.C message.C
3  hello:          $(CCFILES:%.C=%.o)
4                  g++ -o hello $^
5  # Dependencies are supposed to be in a file ``make.depend''
6  # which is inclduded by make.
7  include make.depend
```

```
8  # Because of the following rule, ``make'' will attempt to
9  # create ``make.depend'' if it does not exist or if one
10 # of the files in $(CCFILES) is more recent than ``make.depend''
11 make.depend:    $(CCFILES)
12                 g++ -M $^ >$@
```

See the Section on **make** in **"Unix for beginners"** for more information on the **make** program.

```
76 dv2$ make
Makefile:5: make.depend: No such file or directory
g++ -M hello.C message.C >make.depend
g++    -c hello.C -o hello.o
g++    -c message.C -o message.o
g++ -o hello hello.o message.o
77 dv2$
```

A fragment of the `make.depend` file generated for the example is shown below.

```
hello.o: hello.C /usr/local/include/g++/iostream \
 /usr/local/include/g++/iostream.h /usr/local/include/g++/streambuf.h \
 /usr/local/include/g++/libio.h \
 /usr/local/i386-pc-solaris2.7/include/_G_config.h \
 /usr/local/lib/gcc-lib/i386-pc-solaris2.7/egcs-2.91.66/include/stddef.h \
 message.h /usr/local/include/g++/string \
 /usr/local/include/g++/std/bastring.h /usr/local/include/g++/cstddef \
```

4 Summary of most popular options

Option	Description
`-c`	compile, don't link
`-o out`	result of linking is *out* instead of *a.out*
`-Ldir`	try to find libraries (now, **not** at run time) also in *dir*
`-lxyz`	link with library *libxyz.so* or *libxyz.a*
`-static`	link statically
`-Rdir1:dir2`	(solaris) add *dir1* and *dir2* to **runpath**
`-Wl,--rpath,dir1:dir2` **`-Idir`**	(linux) add *dir1* and *dir2* to **runpath** try to find **`#include`** files also in *dir*
`-Wall`	generate warnings for suspicious constructs
`-Wconversion`	generate warnings about suspicious type conversions
`-O2`	optimize generated code
`-g`	generate debug info for use by *gdb* and *ddd*
`-pg`	generate profiling code for use by *gprof*
`-fpic`	generate position-independent code, necessary when making shared libraries
`-shared`	create shared library instead of program
`-E`	just run c++ preprocessor, send output to *stdout*
`-MM`	generate Makefile-format list of dependencies, send output to *stdout*

List of Figures

1 Compiling and linking 3
2 Compiling and linking with a library 6