Practical guide for C/C++ programming (2005)

This file contains a few practical tips that can help you develop your C++ programs in the VU/FEW environment under UNIX.

1. Getting started

Here are the basic steps to test your C++ program.

Edit a simple C++ source code file using any available text editor, like ipe, emacs or vi.
You can try this program:

```
#include <iostream>
using namespace std ;

int main()
{
  cout << “Testing 1, 2, 3 \n” ;
  return 0 ;
}
```

Save the source code in a file with the extension cpp, for example test.cpp.

You must reside in the directory containing your source code.

Compile your file with GNU project C++ compiler, g++ , by typing:

```
>/usr/local/bin/g++  test.cpp
```

If your file does not have errors, an executable file is generated with the standard name a.out

Run this program by typing :

```
>a.out
```

The result should look like this on your screen:

```
>Testing 1, 2, 3
>
```

Try to edit your program again and introduce deliberately some syntax errors. Compile again and look at the error messages.

Important: If you want to compile the programs in the VU FEW environment, then you should use:

/usr/local/bin/g++ instead of simply g++.

The same path should be used everywhere in the makefile.

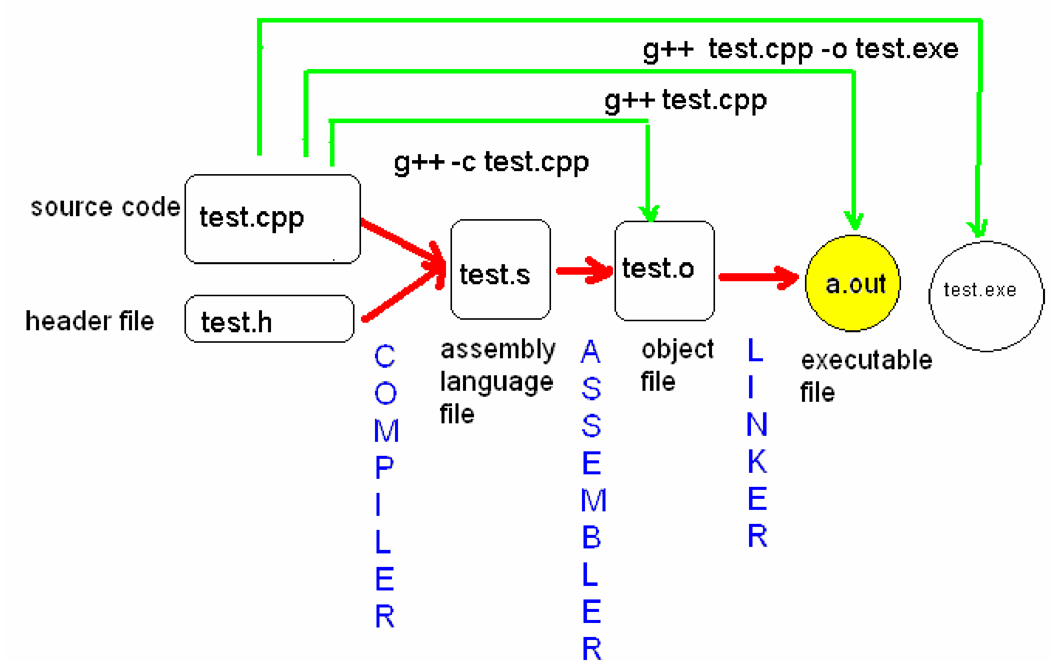
2. The compiler g++

The compiler used under Unix at VU is GNU project C/C++ compiler gcc.
g++ is actually a script to call the gcc compiler with options to recognize C++.

Basically there are 3 steps to obtain the final executable program, as shown:

1. **Compiler** stage: All C/C++ language code in the *.cpp* file is converted into a lower-level language called Assembly language; making **.s* files.
2. **Assembler** stage: The assembly language code made by the previous stage is then converted into *object code* which are fragments of code which the computer understands directly. An object code file ends with *.o*.
3. **Linker** stage: The final stage in compiling a program involves linking the object code to code libraries which contain certain "built-in" functions. This stage produces an executable program, which is named by default *a.out* .

The compiler g++ can process an input file through all theses stages and directly create the executable file or can stop after the assembler stage, creating an object file. How far the compiler will go in one step depends on the flag options you use (see figure).



g++ compiler options

g++ allows for options to be flagged during the compilation. We will show a few useful options.

-c

This flag forces the compiler to produces object files from source files without linking . For example typing:

```
>g++ -c test.cpp
```

will produce an object file test.o

-o

This flag allows to rename the executable file with another name than a.out. For example by typing

```
>g++ test.cpp -o test.exe
```

an executable with the name test.exe will be created.

-Wall

This flag forces the compiler to display all the warning messages. If you have only warnings and no errors an executable will be still created. But the option is useful when you have errors and you cannot find the reason. We suggest you always use this option and treat every earning as an error to be fixed.

Multiple files: g++ can compile multiple files into one executable. For example the file functions.cpp and test.cpp are compiled and the executable will become test.exe.

```
>g++  functions.cpp test.cpp –o test.exe
```

Note! Header files *.h don't have to be compiled.

A complete description for the compiler g++ options you can find with the Unix command `man  g++`. A lot of g++ tutorial scan be found on the Internet.

3. The gmake utility

Suppose you are working on a large program that has many component functions. You can put all functions in one file, but then you will have to recompile the entire file even for a minor change in one of the functions. You can divide the functions among more files, but then you have to remember which file to recompile when you made a change. Then you have to link the object modules to produce an executable file.

The UNIX **make** utility allows you to divide the program's component functions among various files. Whenever you alter a file it takes note of the fact and recompiles only the altered file. It then links the resulting object modules with the others to produce an executable program. Make gets its information from a file named by default makefile.

gmake is the name of the GNU project make utility available at VU/FEW. Further one we will talk only about gmake.

Example

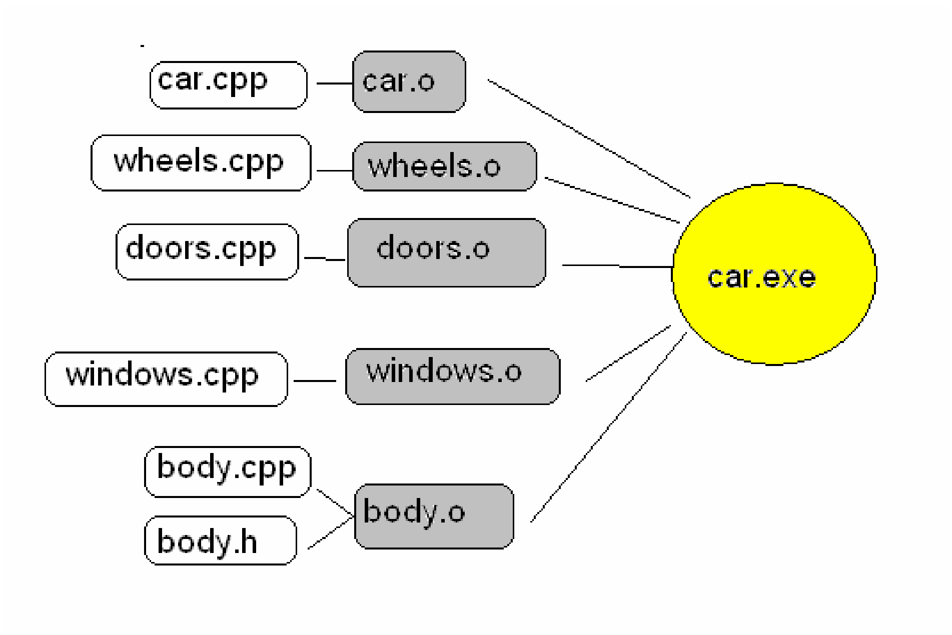
Let's take an example of a pretty large program where the functions are divided in various files. The main program is called car.cpp and the rest of the modules are named: wheels.cpp, doors.cpp, windows.cpp, body.cpp, body.h.

Building an executable by hand

You can build the executable file car.exe by hand, by typing:

```
g++ car.cpp wheels.cpp doors.cpp windows.cpp body.cpp –o car.exe
```

The build process will have two steps: the compilers takes the sources files and outputs object files, after that the linker takes the object files and creates an executable.



But any time you modify something in one module, for example in doors.cpp, you will have to retype the whole command, and recompile even modules that have not been changed.

Using gmake and a makefile

A much easier solution is to use gmake utility. gmake reads what he has to do from a file named by default makefile.

A simple makefile for our example is shown below:

```
car: car.o  wheels.o  doors.o  windows.o body.o
      g++ car.o wheels.o doors.o  windows.o  body.o –o car
car.o: car.cpp
      g++ -c –Wall car.cpp
wheels.o: wheels.cpp
      g++ -c –Wall wheels.cpp
doors.o: doors.cpp
      g++ -c –Wall doors.cpp
windows.o: windows.cpp
      g++ -c –Wall windows.cpp
body.o: body.cpp body.h
      g++ -c –Wall body.cpp
```

An informal translation of this file should begin like this:

Line 1: The main target is car.exe. In order to build it, the following object files are needed: car.o, wheels.o, doors.o, windows.o and body.o.
Line 2: What command do we use to build the car.exe target? `g++ car.o wheels.o doors.o  windows.o  body.o –o car`
Now is the question what do we need to obtain these object files?
Line 3: For car.o we need the source file car.cpp.
Line 4 : What command do we use to obtain car.o from car.cpp? g++ -c car.cpp,
Line 5: For wheels.o we need the source file wheels.cpp
etc.

The file consists of a sequence of 6 rules. Each rule consists of a line containing 2 lists of names separated by a colon (:), followed by one line beginning with **a tab character (not spaces!).**
The name preceding the colon (:) are known as **targets**. They are most often the names of the files to be produced. The names after the colon are known as **dependencies** of the targets. They usually denote other files that must be present and up-to date before the target can be processed. The lines starting with tabs are called **actions**. They are Unix shell commands that get executed in order to create or update the target of the rule.

Each rule has the structure:

```
target: which files are needed for this target?
[tab] how to build the target from these files?
```

The first target of the first rule is the default. In our example the default target is car.

Cleaning the mess

A common use is to put a standard clean-up operation into each makefile, specifying how to get rid of files that can be reconstructed if necessary (like *.o files).
You should always put a rule like this in a makefile:

```
clean:
      rm –f *.o
```

Variables and comments

You can also use variables and comments when writing makefiles. It comes handy when you want to change the compiler or the compiler options.
First a variable has to be assigned, for example CC=g++. Then a variable can be used with $CC.
In the example below we show a makefile where we replaced g++ with the variable CC and the compilers options with the variable CFLAGS.
A cleaning rule is also attached.

```
# This is a comment to say that the variable CC will be the compiler
to use

CC=g++

#This is a comment to say that CFLAGS is a variable to specify the
compiler flags

CFLAGS = -c -Wall

car: car.o  wheels.o  doors.o  windows.o body.o
      $(CC) car.o wheels.o doors.o  windows.o  body.o –o car
```

```
car.o: car.cpp
      $(CC) $(CFLAGS) car.cpp
wheels.o: wheels.cpp
      $(CC) $(CFLAGS)  wheels.cpp
doors.o: doors.cpp
      $(CC) $(CFLAGS) doors.cpp
windows.o: windows.cpp
      $(CC) $(CFLAGS) windows.cpp
body.o: body.cpp body.h
      $(CC) $(CFLAGS) body.cpp

clean: rm –f *.o
```

Execution

The utility gmake is executed by typing :

```
gmake
```

It will then look for the default makefile.
If this file does not exist, you can specify the name of another makefile by typing:

```
gmake –f my_makefile
```

More information on gmake can be found on the Internet or by typing `man gmake`.

Important!! If you want to compile the programs in the VU FEW environment, then you should use:

`/usr/local/bin/g++` instead of simply `g++`.

The same path should be used everywhere in the makefile.