

## Practical guide for C/C++ programming (2008/2009)

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

### Getting started (Linux)

Since your programs are required to work under Linux, it is best that you develop them under Linux as well. The VU offers Linux workstations in the rooms P3.37 and P4.47. Login using your VU-net-id. In case you have trouble logging in, contact the helpdesk in room S4.09.

Developing your program is similar to how you did it in the course "Inleiding Programmeren." You edit your source file with a text editor, and use a console window (shell, terminal) to compile and run your programs.

### Getting started (Windows)

If you don't have access to a Linux workstation on the VU, it is also possible to develop under Windows. jEdit is available and you can save files in your home directory. You still need to make sure your files compile under Linux so you need to open a secure shell-connection (ssh) to one of the computerservers of the VU. For Windows, you can download the program PuTTY from:

<http://www.chiark.greenend.org.uk/~sgtatham/putty/>

For host name, use `kits.few.vu.nl` (or if you're connecting from inside the VU, just `kits` is fine). Login, and you will have a terminal to enter commands in.

Note that Windows saves files differently compared to Unix. You need to make sure that files are saved in Unix (`\n`) format, not Windows (`\r\n`). In jEdit, you can set this for all files in Utilities > Global Options > General > Default line separator. For files you already have open, see Buffer Options.

### Your first C++ program

Here are the basic steps to create and run a C++ program.

Create a new file with any text editor, such as jEdit, Emacs or Vim. Under Linux, you can start jEdit by typing `ipe` on the console.

Type the following 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 (or perhaps you already made some errors while copying the code). 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`.

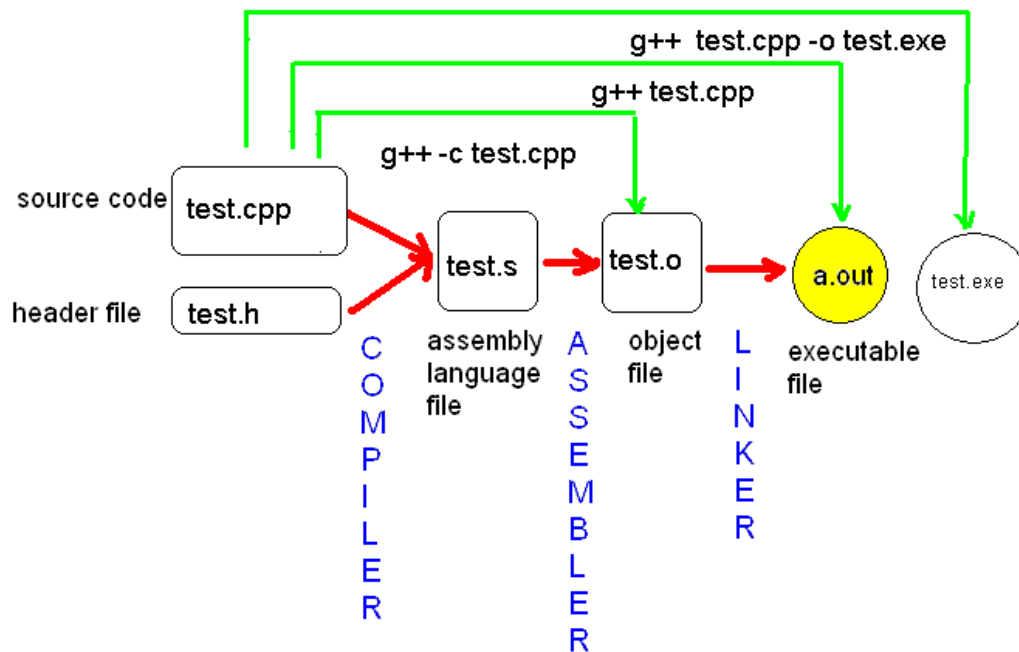
### The compiler `g++`

The compiler used under Linux at the VU is the GNU C/C++ compiler `gcc`. `g++` is actually a script that calls the `gcc` compiler with options to recognize C++.

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

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 that 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 that 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 these 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 produce 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 you to rename the executable file with another name than the default 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 only have warnings and no errors, an executable will be still created. But this option is useful when you have run-time errors and you cannot find the reason. We suggest that you always use this option and treat every warning as an error to be fixed.

### Other compiler options

```
-Wredundant-decls -Wcast-qual -Wcast-align -W -Wconversion -Wshadow
-Wcast-qual -Wwrite-strings -ansi -Wextra -pedantic
-Wswitch-enums -Wunreachable-code
```

A complete description for the compiler g++ options you can find with the Linux command `man g++`. A lot of g++ tutorials can be found on the Internet, for example on:

<http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

### 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.

### The make 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 Linux `make` utility allows you to divide the program's component functions among various files. Whenever you alter a file it takes note of that 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`

`make` is the name of the GNU `make` utility available at VU/FEW. Further on we will talk only about `make`.

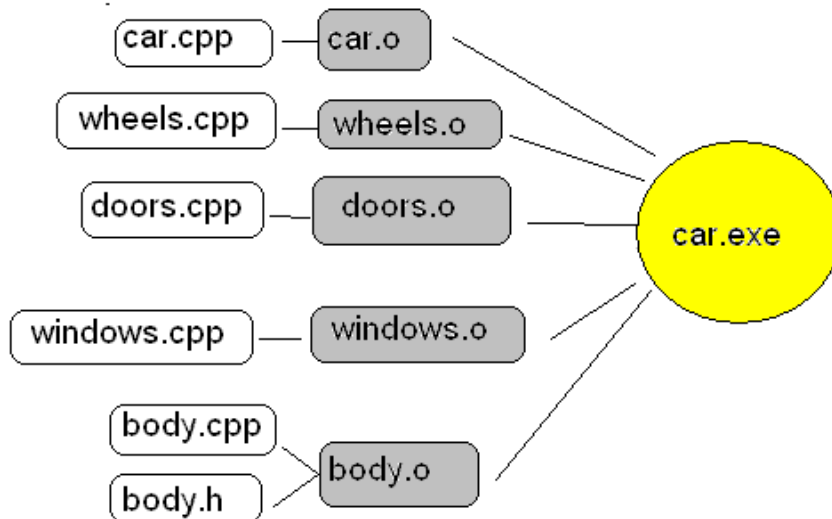
### Building an executable by hand

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`.

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 compiler takes the source 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 make and a makefile

A much easier solution is to use the `make` utility. `make` reads what it has to do from the file `Makefile`. This file contains rules that each have the structure:

```
target: files needed for target
        how to build the target from these files
```

A simple `Makefile` for our example is shown below:

```
car.exe: car.o wheels.o doors.o windows.o body.o
    g++ car.o wheels.o doors.o windows.o body.o -o car.exe
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 is as follows:

**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.exe`

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 (*no 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 Linux shell commands that get executed in order to create or update the target of the rule.

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

### 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 in handy when you want to change the compiler or the compiler options.

First a variable has to be assigned, for example `CC=/usr/local/bin/g++`. Then a variable can be referenced 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 comment says that CC will be the compiler to use
CC = /usr/local/bin/g++

# This comment says that CFLAGS specifies compiler flags
CFLAGS = -c -Wall

car.exe: car.o wheels.o doors.o windows.o body.o
    $(CC) car.o wheels.o doors.o windows.o body.o -o car.exe
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 of the Makefile

make is executed by typing:

```
make
```

It will then look for the default Makefile. (If this file does not exist, you can specify the name of another Makefile by typing: `make -f my_makefile`)

Important!! If you want to compile the programs in the FEW environment, then you should use `/usr/local/bin/g++` instead of simply `g++`.