

C Programming under Linux

P2T Course, Martinmas 2003–4 *C Lecture 5*

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- A Short Introduction to 'make'
- The Scope of Variables
- Functions
- Recursion
- Structured Programming

<http://www.physics.gla.ac.uk/~kaiser/>

The make Utility

- So far we have compiled our programs by hand using a command line for `gcc`. Before we move on to functions and more complex structures of programs, e.g. multiple source files, we will learn to use the `make` utility.
- `make` handles the details of compiling and linking for us and **only does those steps that are necessary** by checking which source file has been edited when.
- `make` reads in the file `Makefile` in the same directory.
- `make` exists for Windows as well as for Unix. The version typically used under Linux is GNU make `gmake`; usually `make` will be defined as alias for `gmake`.

Simple Makefile for gcc

A simple `Makefile` for `gcc` that can be used with C programs contained in a single file by just editing the name of the source file:

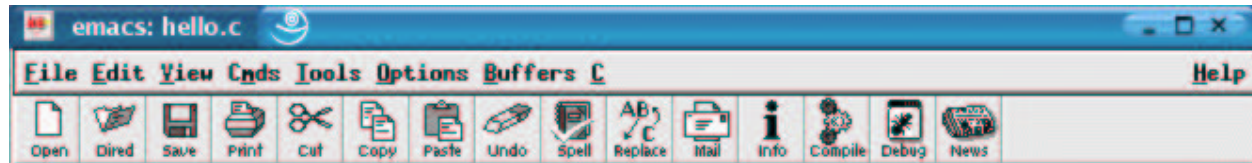
```
#-----#
#       Makefile for unix systems       #
#   using a GNU C compiler               #
#-----#
CC=gcc
CFLAGS=-g -Wall -D__USE_FIXED_PROTOTYPES__ -ansi
#
# Compiler flags:
#   -g          -- Enable debugging
#   -Wall       -- Turn on all warnings
#   -D__USE_FIXED_PROTOTYPES__
#               -- Force the compiler to use the correct headers
#   -ansi       -- Don't use GNU extensions.  Stick to ANSI C.

hello: hello.c
    $(CC) $(CFLAGS) -o hello hello.c

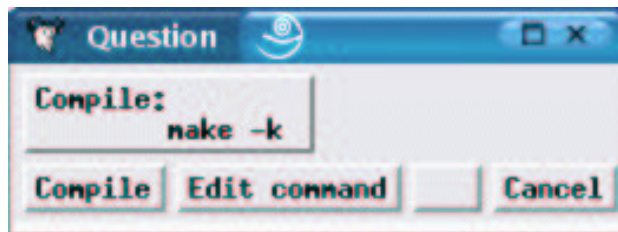
clean:
    rm -f hello
```

Using xemacs with make

- If you are using `xemacs` to edit your source code, you can configure the 'Compile' button easily to call `make`.



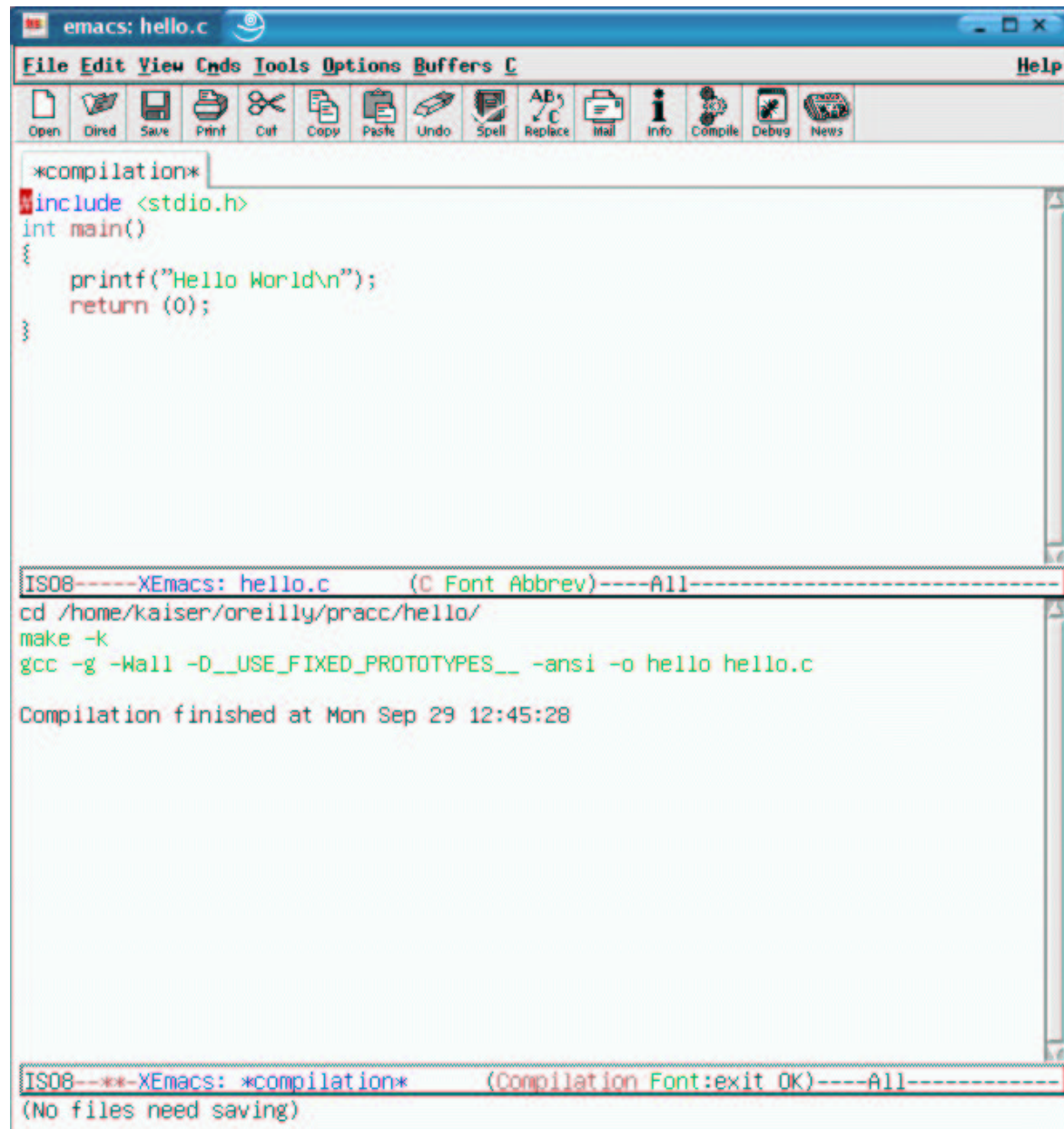
Upon first clicking at the 'Compile' button, a configuration window opens that allows to configure the compilation command:



Afterwards, simply clicking at 'Save' and 'Compile' will compile and link the program.

- The `xemacs` window will split vertically into two, with the bottom one displaying the compiler command and warning messages that otherwise go to `stdout`.

Using xemacs with make



The screenshot shows the XEmacs editor window titled "emacs: hello.c". The menu bar includes File, Edit, View, Cnds, Tools, Options, Buffers, C, and Help. The toolbar contains icons for Open, Dired, Save, Print, Cut, Copy, Paste, Undo, Spell, Replace, Mail, Info, Compile, Debug, and News. The main text area contains the following C code:

```
*compilation*  
#include <stdio.h>  
int main()  
{  
    printf("Hello World\n");  
    return (0);  
}
```

Below the code is a terminal window showing the compilation process:

```
IS08-----XEmacs: hello.c      (C Font Abbrev)-----All-----  
cd /home/kaiser/oreilly/pracc/hello/  
make -k  
gcc -g -Wall -D__USE_FIXED_PROTOTYPES__ -ansi -o hello hello.c  
  
Compilation finished at Mon Sep 29 12:45:28
```

At the bottom of the terminal window, a message indicates the compilation was successful:

```
IS08---***XEmacs: *compilation*      (Compilation Font:exit OK)-----All-----  
(No files need saving)
```

Using xemacs with make cont.

- Depending on the version of `emacs` or `xemacs` there may be an entry 'Compile' in a `pull down menu` instead and the configuration of the command may happen not in an extra window but in the `command line at the bottom of emacs`.
- The `make` command itself of course also has a number of command line options (flags). You can use '`man make`' to find out more. A frequently used option is `make -k`:

```
-k    Continue as much as possible after an error.  While the tar  
      get that failed, and those that depend on it, cannot be  
      remade, the other dependencies of these targets can be pro  
      cessed all the same.
```
- **Warning on Makefile syntax:**
The line '`$(CC) $(CFLAGS) -o hello hello.c`' must start with a **tab**, not eight spaces.

Scope and Class of Variables

- Variables we have used so far were **global** variables. Now we will have a look at **local** variables and the **scope** of variables in general.
- All variables have two attributes besides their type: **scope** and **class**.
- The **scope** of a variable is the area of the program in which the variable is valid.
- A **global** variable is valid everywhere in the program.
- A **local** variable has a scope that is limited to the **block** in which it is declared. It cannot be accessed outside that block.
- A **block** is a section of code enclosed in **curly braces**{ }.
- The class of a variable is either **permanent** or **temporary**. (We'll get back to that.)

Scope of Variables cont.

- Between a global and a local variable with identical names, **the local one takes precedence**. However, this is generally not good practice.

```
int global;
main()
{
    int local;

    global = 1;
    local = 2;

    {
        int very_local;

        very_local = global+local;
    }
}
```

```
int total;
int count;

main()
{
    total = 0;
    count = 0;
    {
        int count;

        count = 0;

        while (1) {
            if (count > 10)
                break;
            total += count;
            ++count;
        }
        ++count;
        return(0);
    }
}
```

Class of Variables

- Global variables are always **permanent**. They are created and initialised before the program starts and remain until it is terminated.
- **Temporary** variables are allocated from a section of memory called the **stack** at the beginning of the block. If you try to allocate too many local variables, you will get a 'Stack overflow' error. Each time the block is entered, the temporary variables are initialised.
- The **size of the stack** depends on system and compiler. under MS-DOS/Windows the stack space must be less than 65,536 bytes, on Linux systems it may be larger, typically up to 8 MByte.
- **gcc** has compiler options that help dealing with the stack size.
- Local variables are **temporary** unless they are declared **static**, in which case they may be **local, but permanent**.

Scope and Class Overview

Declared	Scope	Class	Initialised
outside all blocks	global	permanent	once
<code>static</code> outside all blocks	global	permanent	once
inside a block	local	temporary	each time block is entered
<code>static</code> inside a block	local	permanent	once

- A `static` declaration made outside blocks indicates that the variable is local to the file in which it is declared. (More about this if/when we get around to multiple files.)

Class of Variables - Example

Demonstrate the difference between permanent and temporary local variables (`vars.c`).

```
#include <stdio.h>

int main() {
    int counter;                                /* loop counter */
    for (counter = 0; counter < 3; ++counter) {
        int temporary = 1;                     /* A temporary variable */
        static int permanent = 1;              /* A permanent variable */
        printf("Temporary %d Permanent %d\n",
            temporary, permanent);
        ++temporary;
        ++permanent;
    }
    return (0);
}
```

Output:

```
Temporary 1 Permanent 1
Temporary 1 Permanent 2
Temporary 1 Permanent 3
```

Functions

- **Functions** allow to group commonly used code into a compact unit that can be used repeatedly. We have already encountered one function, the function `main()`.
- The generic syntax for a function is

```
return-type function-name(parameters)
{
    declarations;
    statements;
    return(result);
}
```
- A function consists of a **header** (the first line) and a **body** (the part in curly braces).
- The return-type is not strictly required, if missing it defaults to `int`. However, you should always use one, keeping in mind that a later maintainer otherwise might wonder if you intended `int` or just forgot.

Functions cont.

- Assume we have a function

```
float triangle(float width, float height)
```

It is called e.g. as `triangle(1.3, 8.3)`, in which case C copies the values (1.3 and 8.3) into the functions parameters (width and height). This is known as 'call by value'.

- The alternative would be 'call by reference', where not the value of a variable is passed to the function, but it's address. (This is e.g. the case in FORTRAN.)
- A C function cannot pass data back to the caller using parameters. (Well, it can with some tricks using pointers, but we'll only learn that later.)
- The `return` statement is used to give the result to the caller. The return statement can contain a statement, e.g.

```
return(width*height/2.0);
```

Function Prototypes

- If we want to **use a function before we define it** - because we like to order the functions in the file to be in a particular order, or we just want to be sure and not have to check in which order the functions are - we must declare the function just like a variable to inform the compiler about it.
- We use a declaration like

```
float triangle (float width, float height);
```

This declaration is called the **function prototype**.

- In the function prototype variable names are not required, only their type, so it would also be ok to just write

```
float triangle (float, float);
```

- Strictly speaking, if no prototype is specified the C compiler assumes the function returns an `int` and takes any number of parameters. But we don't want to get into bad habits here.

Functions and void

- A function may have no parameters, but the **function prototype** should not have an empty parameter list. (At least not if it's not `main()` which typically has just that.) In this case the keyword **void** is used to indicate an empty parameter list:

```
int next_index(void)
```

- In an assignment statement you only have to use empty brackets:

```
value = next_index()
```

- A function also does not have to have a **return** value. In this case it gets the return-type **void**, indicating that no value will be returned.

```
void print_answer(int answer){  
    if (answer < 0) {  
        printf("Answer corrupt\n");  
        return;  
    }  
    printf("The answer is %d\n", answer);  
}
```


Functions and Comments

- This is a good moment to come back to the importance of comments. In the example used over the last few lectures we have most of the time left most of the comments out - simply because otherwise not all of the code would fit onto the slide. However, **comments are crucial**. (And expected in your laboratory assignments and exams....).
- Each function should begin with a comment block, containing
 - Name of the function
 - Description of what the function does
 - Description of each of the parameters of the function
 - Description of the return value of the function
 - Formats, references, anything else that would be really useful

Function - Example

Compute area of a triangle (tri-sub/tri-prog.c).

```
#include <stdio.h>

/*****
 * triangle -- compute area of a triangle  *
 *                                           *
 * comments, comments, comments...        *
 *****/
float triangle(float width, float height)
{
    float area;      /* Area of the triangle */

    area = width * height / 2.0;
    return (area);
}

int main()
{
    printf("Triangle #1 %f\n", triangle(1.3, 8.3));
    printf("Triangle #2 %f\n", triangle(4.8, 9.8));
    printf("Triangle #3 %f\n", triangle(1.2, 2.0));
    return (0);
}
```

Recursion

- **Recursion** occurs when a function calls itself, directly or indirectly. Some programming functions, e.g. the factorial, lend themselves naturally to recursive algorithms.
- A recursive function must follow two basic rules
 - It must have an ending point.
 - It must make the problem simpler.
- **Example: Factorial $n!$**
A definition of factorial is

$$0! = 1 \qquad n! = n * (n - 1)!$$

- In C this turns into

```
int fact(int number){  
    if (number == 0)  
        return (1);  
    return (number * fact(number-1));  
}
```

Structured Programming

- There are quite a number of different programming methodologies, some use simple flow-charts, others object-oriented design (OOD), one of the latest fads is 'extreme programming' that involves two programmers working together (and some other stuff). In this course we're not going to get into any of these.
- However, now that we know **functions**, we can go and **structure** our programs.
- Simple rules are to divide the code into functions, that each should be short enough to be not too hard to understand. Three A4 pages is a practical limit. Shorter is easier.
- Also, have an idea of the overall structure that the program has.
- And start with simple versions of the functions and make them more complex when the simple versions work.
- Don't forget about comments.