Basic Programming in C

This is mainly just to look back at basic C code structure - some of these are never taught in the basic programming course. The codes in this topic emphasize on C language standard - you should be able to compile and run the codes on any platform that has a C compiler. You just have to remember that some libraries only exists on specific platforms - so, make sure you know your target and development platforms. There are also some info on project management using makefiles.

C Programming: Preparations

Obviously, a text editor (some may want to call it code editor, to be more spcific) to write code, a C compiler to compile the code and a linker to create the executable. What is most commonly used would probably be an Integrated Development Environment (IDE) which combines all of them into a single environment which is mainly the editor - the compiler and linker are just menu items or buttons somewhere within.

***Note**: It is advisable to manage source files in proper folder/path hierarchy. One thing about naming files and paths is NEVER use a whitespace character (' ') in them, no matter what Windows says about them not being a problem (Apparently, even Windows have ditched 'My Documents').*

Code Editing

The source file is just a text file - so, it is actually possible to use any text editor. On Windows platform, `notepad` or even the console-based `edit` (relic from DOS era) can be used to write code. The downside of using these editors are they lack syntax-highlighting which is very useful to start learning programming by making it easier to spot keyword/structure-related syntax errors early on. There are a lot of free code editors (text editor specifically for programming purpose that provides syntax highlighting) - a recommended editor is Geany.

On Linux, IDEs are also available but many veteran programmers simply use Linux text editors which naturally supports syntax highlighting. Among the popular editors are `Emacs` and `Vim`. On popular distributions like Ubuntu, the supplied GUI-based editor (GNOME editor a.k.a. `gedit`, which is available on all distributions that use GNOME desktop) should be sufficient. On KDE desktop, `kwrite` or `kate` does the job pretty well.

Compiling (and Linking)

The instructions in this section require the use of a terminal. If using an IDE, just look for a menu item or button that says compile or build or make or something similar (they actually do the same thing as what is mentioned here).

For a single source file `this.c`, this process can be done in a single step:

```
gcc this.c
```

By default, the `gcc` (GNU Compiler Collection instead of GNU C Compiler) compiler will create an executable called `a.out`. If there is a need to specify a name (e.g. `this`) for the binary, the -o parameter can be used:

```
gcc -o this this.c
```

It does not need be similar to the name of the source file:

```
gcc -o what_the this.c
```

Note that both compiling and linking can be done in this single command. They are actually separate processes and most of the time, it is very useful to execute them in separate stages especially when managing a software with multiple source files.

Managing a Project

If using a full-fledged IDE, it is common to find everything but the kitchen sink available (sometimes a sink can also be found 😬). One of the things that `make` an IDE seems to work better is having a project management flow, i.e. keeping tracks of source files required to build the project, which compiler to use, etc. For an old school developer, using a `makefile` (requires `make` tool) makes more sense. This is what going to be introduced here.

Like other configuration files in Unix/Linux systems, writing a `makefile` (which is also a simple text file) is relatively easy. Any strings following the '#' character are treated as comments. A simple example of a `makefile` that can be used to compile a C program from a single 'single.c' source file is shown below.

***Note**: This is based on GNU make utility. There are other variation(s) such as the BSD make.*

makefile

```
# makefile to compile a c program
single: single.c
    gcc -o single single.c
```

The syntax will be explained after the next example. Take note that there should be a single hard tab before the gcc command. The example given above, however, is not very convenient because a lot of things have to be changed (three words at least in the example) if there is a change to file or project name. The example below gives a more generic `makefile`.

makefile

```
# makefile to compile a c program
PROJECT = single
SOURCE = $(PROJECT).c
CC = gcc
$(PROJECT): $(SOURCE)
```

```
    $(CC) -o $(PROJECT) $(SOURCE)
```

The two common things done in a makefile is to define build rules and to define variables. The lines that contains a '=' is usually a variable definition. Any environment variable is imported into `make` environment. A rule is defined by specifying a target that is done by specifying a label (or target object name) followed by a ':' character. In the given example, the target is $(PROJECT) - which is a variable and expands to `single`. Anything that comes after ':' will be considered as a prerequisite. If a target is older than any of its prerequisites, it will be rebuilt by `make` tool. The build process is defined by the lines following the target (there can be more than 1 line). Notice that the MUST BE a hard tab character at the beginning of a process line.

Imagine now that there are a hundred source files assigned to 'SOURCE'. Using the above rule, changing one source file will cause all the other source files to be recompiled again. This is not very efficient. It is actually useful to know that an executable is built by combining object files and linking them to the respective libraries. Object files (*.o or *.obj) can be built independently. So, to have that, the `makefile` needs some additional line(s) to accommodate this.

makefile

```
# makefile to compile a c program
PROJECT = single
OBJECTS = $(PROJECT).o
CC = gcc
CFLAGS =
LDFLAGS =
$(PROJECT): $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $+ $(LDFLAGS)
%.o: %.c
    $(CC) $(CFLAGS) -c $<
```

There are a few new things here. Special variables like $@, $+ and $< are now used. To know what they mean, refer to the note at the end of this section. The prerequisites are now object files that needs to be built as well. In order to build these, `make` will refer to the second rule `%.o: %.c` - which says that any object files (*.o) needs to be built depends on a source C file (*.c) that will be built using gcc with a `-c` parameter (compile only).

One important thing to know is `make` will use the first matching rule that it finds. So, if you have some object files that depends on both source (*.c) and header (*.h) files, you need to create the rule (`%.o: %.c %.h`) *before* the rule given above.

The given information so far about makefiles should be sufficient for basic projects. You will learn, in time, of more advanced ways to specify rules that will show you how the `make` tool can be a very powerful build management tool.

Some notes on special `make` variables:

```
# $@ ==> target file
# $? ==> changed dependents
# $* ==> shared target/dependent prefix
```

```
# $< ==> first dependent
# $^ ==> all dependents, duplicates omitted
# $+ ==> all dependents, order retained
```

Practical makefile

This is what I use (extracted from my1codeapp repo) to quickly compile a C source code (usually to test some things).

makefile

```
# makefile for single c source file app

ALLSRC = $(subst src/,,$(wildcard src/*.c))
ALLSRC += $(wildcard *.c)
ALLAPP = $(subst .c,,$(ALLSRC))

TARGET ?=
OBJLST ?=
DOFLAG ?=
DOLINK ?=

CC = gcc

DELETE = rm -rf

CFLAGS += -Wall -Isrc
# i prefer to build static bin
#CFLAGS += -static
LFLAGS += $(LDFLAGS)
OFLAGS +=
# when working with large files
#XFLAGS += -D_LARGEFILE_SOURCE=1 -D_FILE_OFFSET_BITS=64
ifneq ($(DOFLAG),)
    CFLAGS += $(DOFLAG)
endif
ifneq ($(DOLINK),)
    LFLAGS += $(DOLINK)
endif

# i can still squeeze in some external code(s) => OBJLST!
EXTPATH = ../my1codelib/src
ifneq ($(wildcard $(EXTPATH)),)
    CFLAGS += -I$(EXTPATH)
endif

.PHONY: dummy all clean

# TARGET can be temporary code (reside at top level)
```

```
$(TARGET): $(OBJLST) $(TARGET).o
	$(CC) $(CFLAGS) -o $@ $^ $(LFLAGS) $(OFLAGS)

dummy:
	@echo "Run 'make <app>' or 'make TARGET=<app>'"
	@echo "  <app> = { $(ALLAPP) }"
	@echo
	@echo "To set compiler flag (e.g. static), do 'make <app> DOFLAG=-
static'"
	@echo "To link a library (e.g. math), do 'make <app> DOLINK=-lm'"

all: $(ALLAPP)

%: src/%.c
	$(CC) $(CFLAGS) -o $@ $< $(LFLAGS) $(OFLAGS)

%: %.c
	$(CC) $(CFLAGS) -o $@ $< $(LFLAGS) $(OFLAGS)

# to compile those external code(s) => OBJLST!
%.o: $(EXTPATH)/%.c $(EXTPATH)/%.h
	$(CC) -c $(CFLAGS) -o $@ $<

clean:
	-$(DELETE) $(ALLAPP) *.o
```

2026/08/11 09:31 · azman