Systems Programming

Lecture 4: Makefiles and some more C

Stuart James

stuart.a.james@durham.ac.uk

Durham University

Recap

https://PollEv.com/stuartjames

Recap

Recall: A First Program


Recap

Recall: A First Program

In [7]:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
```


Recap

Recall: A First Program

In [7]:

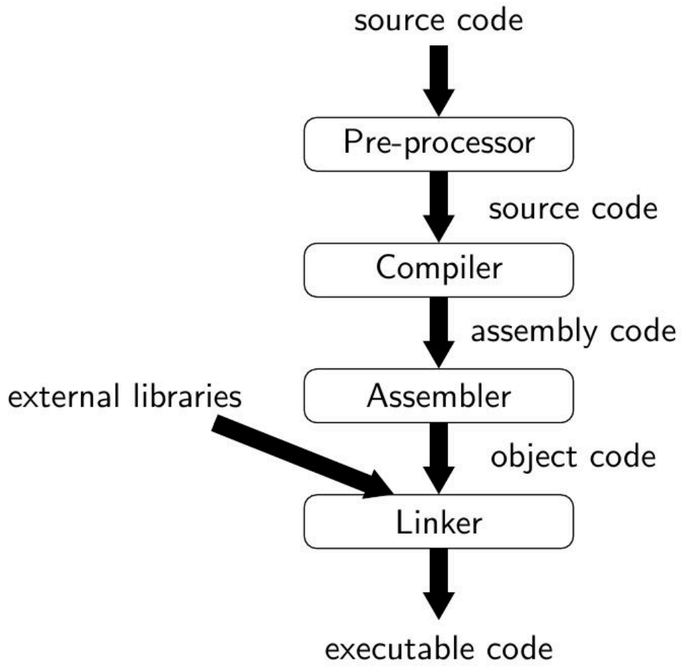
```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
```

Saved in a file with a "`.c`" file extension, for example "`helloworld.c`"


Recap

Compiling



Compiling

```
gcc -o outfile file.c
```

- Use `-o` to name the output
- Use `-E` option to do pre-processing only, or call `cpp`
- Use `-S` option to go as far as compilation only
- Use `-c` option to go as far as assembly only


Compiling

```
gcc -o outfile file.c anotherfile.c -lboost -lncurses -I /some/path/
```

- Use `-l` to link e.g. external libraries
- Add more c files
- Add include path for more header files

You can see that our compiling process can quickly become very complicated!


Makefiles

When we have a number of files to compile together, we need a rule-set.

- The `make` command provides this
- Requires a rule-file called the `Makefile`
- Declarative programming style set of rules for building the program


Installing make

- Under linux no problem (try it on mira)
- On your own computer (if you use windows):
  - You can use the Windows subsystem for linux then you will have access to make and gcc as under linux (https://docs.microsoft.com/en-us/windows/wsl/install)
  - You can install it in VisualStudio: https://docs.microsoft.com/en-us/cpp/build/reference/creating-a-makefile-project?view=msvc-160


Makefiles

- Format of each rule:

```
target [target ...]: [component ...]
        [command 1]
        ...
        [command n]
```

- N.B. Tab character
- `target` - what you want to make
- `component` - something which needs to exist (might need another rule)


Makefiles: Example

- Let's say we have files:
  - `main.c`, `counter.h`, `counter.c`, `sales.h`, `sales.c`


Makefiles: Example

- Let's say we have files:
  - `main.c`, `counter.h`, `counter.c`, `sales.h`, `sales.c`

In [ ]:

```
all: counter.o sales.o main.c
	gcc -o program main.c counter.o sales.o

counter.o: counter.c counter.h
	gcc -c counter.c

sales.o: sales.c sales.h
	gcc -c sales.c

clean:
	rm -rf program counter.o sales.o
```


Makefiles: Macros

- Macros can be used to store definitions
  - `CC=gcc`
- They can be generated from commands
  - `DATE = date`


Makefiles: Macros

- And used in the Makefile

```
all:
        echo This was compiled using $(CC) on $(DATE)
```


Makefiles: Macros

- And used in the Makefile

```
all:
        echo This was compiled using $(CC) on $(DATE)
```

- Running this gives:
  - This was compiled using gcc on Fri Oct 13 08:31:32 AM BST 2023


Makefiles: Pattern Rules

- We can specify a pattern rule which matches multiple files.
- e.g. compile C files into object files:

```
DEPS = counter.h sales.h
%.o: %.c $(DEPS)
	gcc -c $< -o $@
```


Makefiles: Pattern Rules

- contains the character `%` (exactly one of them) in the target
- `%` matches any non-empty substring (similar to * in bash)

Example

- `%.c` as a pattern matches any file name that ends in `.c`.
- `s.%.c` as a pattern matches any file name that starts with `s.`, ends in `.c` and is at least five characters long. (There must be at least one character to match the `%`.)


Makefiles: Automatic Variables

- How do you write a pattern rule? The name of file is different each time the implicit rule is applied.
- Solution: use automatic variables. These variables have values computed for each rule that is executed, based on the target.
- The substring that the `%` matches is called the **stem**.


Makefiles: Automatic Variables

- `$@`: The file name of the target of the rule.
- `$<`: The name of the first prerequisite.
- `$?`: The names of all the prerequisites that are newer than the target, with spaces between them.
- `$^`: The names of all the prerequisites, with spaces between them. Each is listed only once no matter how often it appears, to duplicate use `$+`
- `$*`: The stem with which an implicit rule matches.
    - if the target is `dir.a` and the target pattern is `%.a` then the stem is `dir`.


This would change our original Makefile example to:


This would change our original Makefile example to:

In [ ]:

```
CC=gcc
CFLAGS=-I.
DEPS = counter.h sales.h

all: counter.o sales.o main.c
	$(CC) -o program main.c counter.o sales.o

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

clean:
	rm -rf program counter.o sales.o
```


What have we changed?

- Added CC, CFLAGS and DEPS allowing us to quickly change them.
- first creates the macro DEPS, which is the set of `.h` files on which the `.c` files depend.
- define a rule that applies to all files ending in `.o` which says:
    - `.o` file depends on the `.c` version of the file and the `.h` files included in DEPS.
- The `-c` flag tells the compiler to generate the object file.
- the `-o $@` says to put the output of the compilation in the file named on the left side of the `:`
- the `<=` is the first item in the dependencies list.


Makefiles: A few comments

- Comments - lines starting with `#`
- Lazy evaluation: an expression is not evaluated or computed until its value is actually needed.
- If a target exists and has a timestamp later than all of its components assume it is up to date and don't bother to re-process.
- Nothing to do with C: Although Makefiles are often used with C programs there is no intrinsic link! They can be used with any code/work.
- You can run any specific rule by invoking its target:
  - `make sales.o`


Let's look at a more realistic example!

outside the slide...

How would we do this using the C++ Compilere (g++)

Microsoft CoPilot:

Prompt: `create a makefile for an executable program using g++ with a main.c and a helper.c with respective header files`


How would we do this using the C++ Compilere (g++)

Microsoft CoPilot:

Prompt: `create a makefile for an executable program using g++ with a main.c and a helper.c with respective header files`

Should we have done this?


How would we do this using the C++ Compilere (g++)

Microsoft CoPilot:

Prompt: `create a makefile for an executable program using g++ with a main.c and a helper.c with respective header files`

Should we have done this?

No! Mixing C and C++ is bad practice and our `C` files are still `C` style code.


Beyond Make

- For very large project Makefiles become cumbersome.
- They can automate some aspects of compilation, but not all.
- They can't find and correctly link external libraries.
- Many tools exist, most are difficult to set up but reasonably easy to use.
- You may run into software that you have to install using these tools!


Beyond Make

Autotools

"The first goal of the Autotools is to simplify the development of portable programs. The system permits the developer to concentrate on writing the program, simplifying many details of portability across Unix and even Windows systems, and permitting the developer to describe how to build the program using simple rules rather than complex Makefiles."

- Not really very simple
- only works well under Linux
- used by a fair amount of open-source software


Beyond Make

CMake

"CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice."

- Very flexible
- Works best under Linux or MacOS
- Not always intuitive


Beyond Make

Less common options

- SCons:
  - A little newer than CMake and autotools
  - In the end: pretty similar in terms of usability
- Docker:
  - Instead of having to install anything, many developers now just provide a docker image of their software
  - Advantage: Very easy on the user side
  - Essentially a light-weight VM


Back to C

Summary:

- In the first lecture, we started writing a few simple programs in C.
- We primarily looked at the pre-processor
  - `#include` - for including header files
  - `#define` - to set constants
  - `#ifdef`, `#endif` - if statements
  - `#define MAX(a,b) ((a)<(b)?(b):(a))` - a parameterised macro
- Now, let's get down the basic syntax


True / False and Comparison

- Traditionally, C did not have a boolean type and just uses `int`:
  - 0 - is false
  - Any other `int` is true
- Comparisons `<`, `<=`, `==`, `>=`, `>`, `!=` will evaluate to:
  - `1` if they hold
  - `0` if they don't


True / False and Comparison


True / False and Comparison

In [8]:

```c
#include <stdio.h>

int main(){
    printf("Is 4<5? %d\n", 4<5);
    printf("Is 4==5? %d\n", 4==5);
    printf("Is 4!=5? %d\n", 4!=5);
    return 0;
}
```

```
Is 4<5? 1
Is 4==5? 0
Is 4!=5? 1
```


True / False and Comparison

- C99 introduced `bool`, which is defined in `stdbool.h`.
- You can still use the integer convention if you prefer as many do.
    - aligns with the underlying representation of boolean logic in C!
    - many C standard library functions and operators return integer values that conform to this convention.


Statements and Compound Statements

- A **statement** in C is a single instruction terminated with a semicolon
  - `printf("Hello world!\n");`


Statements and Compound Statements

- A *compound statement* is a set of statements surrounded by curly brackets
  - `{ }`

```
{
printf("Hello ");
printf("world!\n");
}
```

- You can always replace a statement with a compound statement
- C doesn't care about formatting -- but we humans need it!


Iteration statements

C provides three iteration statements:

- The `while` statement is used for loops whose controlling expression is tested before the loop body is executed.
- The `do` statement is used if the expression is tested after the loop body is executed.
- The `for` statement is convenient for loops that increment or decrement a counting variable or iterator.


Iteration statements

The `while` statement

- **The form** (if only one `statement`, we can leave out the `{}`):

```
while ( expression ) {
    statement
   }
```

- `expression` is the controlling expression
- `statement` is the loop body
- The expression is evaluated and if it is nonzero (true), the body is executed.
- The expression is tested before the loop body begins.


The `while` statement

Example


The `while` statement

Example

In [9]:

```c
#include <stdio.h>

int main(){
    int i = 8;
    while(i>0){
        printf("Hello planet\n");
        i = i-1;
    }

    printf("Stay strong Pluto!\n");
    return 0;
}
```

```
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Stay strong Pluto!
```


Iteration statements

The `do` statement

- The `do` statement has the form

```
do {
    statement
  } while ( expression );
```

- `expression` is the controlling expression
- `statement` is the loop body
- The expression is evaluated and if non-zero (true), the body is executed again.
- The expression is tested after the loop body ends.


The `do` statement

Example

The `do` statement

Example

In [10]:

```c
#include <stdio.h>

int main(){
    int i = 8;
    do{
        printf("Hello planet\n");
        i = i-1;
    } while(i>0);

    printf("Stay strong Pluto!\n");
    return 0;
}
```

```
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Stay strong Pluto!
```


Iteration statements

The `for` statement

- The for statement is ideal for loops that have a counting variable, but it's versatile enough to be used for other kinds of loops as well.


Iteration statements

The `for` statement

- General form of the `for` statement:

```
for ( expr1 ; expr2 ; expr3 ){
    statement
  }
```

- `expr1` - initialisation
- `expr2` - conditional
- `expr3` - increment


The `for` statement

Example

The `for` statement

Example

In [11]:

```c
#include <stdio.h>

int main(){
    for(int i = 0; i < 8; i++){
        printf("Hello planet\n");
    }

    printf("Stay strong Pluto!\n");
    return 0;
}
```

```
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Stay strong Pluto!
```


`break`

- this statement causes the innermost enclosing loop to be exited immediately.


`break`

- this statement causes the innermost enclosing loop to be exited immediately.

In [12]:

```c
#include <stdio.h>

int main(){
    for(int i = 0; i < 8; i++){
        printf("Hello planet\n");
        break;
    }

    printf("Stay strong Pluto!\n");
    return 0;
}
```

```
Hello planet
Stay strong Pluto!
```


`continue`

- causes the next iteration for the loop to begin:
  - in the case of a `while` or `do` loop, the test part is executed immediately;
  - in the case of a `for` loop, control first passes to the increment step.


`continue`

- causes the next iteration for the loop to begin:
  - in the case of a `while` or `do` loop, the test part is executed immediately;
  - in the case of a `for` loop, control first passes to the increment step.

In [13]:

```c
#include <stdio.h>

int main(){
    for(int i = 0; i < 8; i++){
        continue;
        printf("Hello planet\n");
    }

    printf("Stay strong Pluto!\n");
    return 0;
}
```

```
Stay strong Pluto!
```

The `if-else` statement

- `if` allows choice between two alternatives by testing an expression.
- An `if` statement may have an `else` clause:

```
if ( expr1 ) {
    statement1
  }
  else {
    statement2
  }
```

- When executed, `expr1` is evaluated;
  - if `expr1` is nonzero, `statement1` is executed
  - otherwise `statement2` (if present) is executed


The `if-else` statement

Example

In [15]:

```c
#include <stdio.h>
#include <stdbool.h>

int main(){

    bool consider_pluto_a_planet = true;

    for(int i = 0; i < 9; i++){
        if(i==8){
            if(consider_pluto_a_planet){
                printf("Hello planet\n");
            }
             else{
                 printf("Stay strong Pluto!\n");
             }
            continue;
        }
        printf("Hello planet\n");
    }
    return 0;
}
```

```
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
Hello planet
```

The `if-else` statement

`if-else` ambiguity

- Technically C allows you to leave out braces, but:
- Sometimes hard to spot this in your code:


The `if-else` statement

`if-else` ambiguity

- Technically C allows you to leave out braces, but:
- Sometimes hard to spot this in your code:

In [1]:

```c
#include <stdio.h>

int main(){

    int n = 1;
    int s = -1;

    if (n > 0)
        for (int i = 0 ; i < n ; i++ )
            if (s > 0)
                printf( "S %d \n", s );
    else
        printf("n is less than zero\n");
    return 0;
}
```

```
/var/folders/4j/xg6vmdqs44z51nqdy93ncv_h0000gn/T/tmpul3kbii5.c:12:5: warning: add explicit braces to avoid dangling else [-W
dangling-else]
   12 |     else
      |     ^
1 warning generated.
```

```
n is less than zero
```

The `if-else` statement

`if-else` ambiguity

- Technically C allows you to leave out braces, but:
- Sometimes hard to spot this in your code:

In [1]:

```c
#include <stdio.h>

int main(){

    int n = 1;
    int s = -1;

    if (n > 0)
        for (int i = 0 ; i < n ; i++ )
            if (s > 0)
                printf( "S %d \n", s );
    else
        printf("n is less than zero\n");
    return 0;
}
```

```
/var/folders/4j/xg6vmdqs44z51nqdy93ncv_h0000gn/T/tmpul3kbii5.c:12:5: warning: add explicit braces to avoid dangling else [-W
dangling-else]
   12 |     else
      |     ^
1 warning generated.
```

```
n is less than zero
```

Summary

- Makefile
- Some basic C syntax
- 3 types of loops:
    - for
    - while
    - do
- continue, break
- if, else