Conditional compilation

- Everything between `#ifdef name` and the respective `#endif`, is removed, unless **macro** `name` is defined.
  - Using `#ifndef` is the inverse.
- `#if expr` uses an arithmetic C expression over integer literals, arithmetic/boolean operators, and macros.
- There are also `#elif expr` and `#else` for the usual branching.

Example

```
1 #ifdef DEBUG
2 fprintf(stderr, "value x = %d\n", x);
3 #endif
```

This code is only compiled if the `DEBUG` macro is defined.

- GCC understands the command line argument `-Dmacro[=def]`, defining a macro with an optional definition, or `int` literal `1` if omitted.

Compile with debugging on:

```
1 $ pk-cc -DDEBUG main.c
```

Compile production code:

```
1 $ pk-cc main.c
```

- Beware of **Heisenbugs** though!

Examples

- Conditional compilation is heavily used to make code **independent** of compiler and platform:

```
1 #ifndef NULL
2 #ifdef  __GNUG__
3 #define NULL    __null
4 #else
5 #define NULL    0L
6 #endif
7 #endif
```

  - This is typical code, using compiler-defined macros to inspect language features.
  - `__GNUG__` is set when compiling C++ code.

- Sometimes one wants to re-implement an **existing macro** as function:

```
1 #ifdef abs
2 #undef abs
3 #warning abs macro collides with abs() prototype, undefining
4 #endif
5
6 int abs(int j);
```

  - `#undef name` makes the preprocessor forget about the named macro.
  - `#warning message` generates a compiler warning.

Including header files only once

These are called **once-only headers**[29]. General idea:

- On **first visit** of a header file, define a macro with **unique** name.
- Next time, hide the headerfile contents, if the macro is defined.

`stack.h`:

```
1 #ifndef STACK_H_INCLUDED
2 #define STACK_H_INCLUDED
3
4 void push(double);
5 double pop(void);
6
7 #endif
```

- The macro name must be **unique** across **all source files**.
  ⇒ At least include the file name, maybe use random strings as well[30].
- Adapt all your header files accordingly.

- CPP does **optimize**: If the contents of an include file are *entirely* wrapped as described, it may **omit scanning the file repeatedly**.
  - Comments put outside the wrapper will not interfere with this optimization.

[29] http://gcc.gnu.org/onlinedocs/gcc-4.8.2/cpp/Once_002dOnly-Headers.html
[30] try `$ mktemp -u XXXXXXXXXX`, *cf.* `mktemp`(1)

Gory details

- Macro **arguments** are completely expanded before they are substituted into the macro body.
- After that substitution, the entire macro body is **scanned again** for macros to be expanded.
- Self-referential macros **do not loop** infinitely, the expansion simply stops before closing a loop. **No warning** is produced!

```
1 #define x (1 + y)
2 #define y (2 * x)
3 x
4 y
```

gives

```
1 (1 + (2 * x))
2 (2 * (1 + y))
```

- Certainly a **good read**: Section *3.10 Macro Pitfalls*[31] in the CPP manual.

[31] http://gcc.gnu.org/onlinedocs/gcc-4.8.2/cpp/Macro-Pitfalls.html

7.4 Building big programs

- The Calculator project consists of **various source files**:

```
$ ls
calc.c  stack.c  stack.h  token.c  token.h
```

- Compilation by hand is **cumbersome**:

```
$ pk-cc -c calc.c
$ pk-cc -c stack.c
$ pk-cc -c token.c
$ ls
calc.c  calc.o  stack.c  stack.h  stack.o  token.c  token.h  token.o
$ pk-cc calc.o stack.o token.o
```

- Of course, we could simply `pk-cc *.c` to just compile every C-file, but:
- After a modification, is it really necessary to **recompile all sources**?

make

`make` is a tool that helps **manage dependencies** between your sources:

- Generates commands required for compiling the project.
- Resolves dependencies.
- Clears up temporary files.
- Minimize build time, *e.g.*, on recompilation.
- May parallelise compilation steps, exploiting multiple CPUs.
- Does other things while you sleep.

Documentation

- `info make`
- Online[32].

[32] http://www.gnu.org/software/make/manual/

`make` is controlled by a *Makefile*

- Usually named `Makefile`, residing in the source directory.
- A Makefile typically contains several **rules** of the form:

```
1 target : prerequisite...   # dependency line
2     recipe
3     ...
```

  - The target is the thing **to be created**, usually a file.
  - The prerequisites are the things that are **required** to build the target. Usually, these are provided files, or targets to be made by other rules.
  - The recipe lines, each **indented with a tab**, contain the commands to execute for building the target.

- `make` calculates the order in which to build the targets. Goal is the **first target** in the Makefile, or the ones specified on the command line.
- For convenience, make supports **variables**.
  - Definition: `name = value`, although there are many other forms.
  - Usage: `$(name)` or `${name}`.

Example `Makefile` for the Calculator

```
CFLAGS = -std=c99 -g -Wall -Wextra -Wpedantic -Wbad-function-cast \
    -Wconversion -Wwrite-strings -Wstrict-prototypes

calc : calc.o stack.o token.o
    gcc -o calc calc.o stack.o token.o

calc.o : calc.c stack.h token.h
    gcc -c $(CFLAGS) calc.c

stack.o : stack.c stack.h
    gcc -c $(CFLAGS) stack.c

token.o : token.c token.h
    gcc -c $(CFLAGS) token.c
```

```
$ make
gcc -c -std=c99 -g -Wall -Wextra -Wpedantic -Wbad-function-cast -Wconve
rsion -Wwrite-strings -Wstrict-prototypes calc.c
gcc -c -std=c99 -g -Wall -Wextra -Wpedantic -Wbad-function-cast -Wconve
rsion -Wwrite-strings -Wstrict-prototypes stack.c
gcc -c -std=c99 -g -Wall -Wextra -Wpedantic -Wbad-function-cast -Wconve
rsion -Wwrite-strings -Wstrict-prototypes token.c
gcc -o calc calc.o stack.o token.o
```

Recompilation and updates

Run `make` again:

```
$ make
make: 'calc' is up to date.
```

- `make` investigates the **timestamps** of the files required to build the target.
- `make` only recompiles the outdated parts of your project.
- You can update the timestamp of a file by `touch`ing it:

```
$ touch stack.c
$ make
gcc -c -std=c99 -g -Wall -Wextra -Wpedantic -Wbad-function-cast -Wconve
rsion -Wwrite-strings -Wstrict-prototypes stack.c
gcc -o calc calc.o stack.o token.o
```

Speedup Command line flag `-jn` tells `make` to run up to n jobs in parallel[33].

[33] *cf.* the `nproc`(1) command

Phony targets

- A target is not required to be a file, it may just be an **abstract concept** of a target: A *phony* target.
- These are declared in the Makefile with the `.PHONY` "target", and are **not expected to create a file** of that name.

```
1  CFLAGS = #...
2
3  .PHONY : all clean distclean
4
5  all : calc
6
7  clean :
8      rm -f *.o
9
10 distclean : clean
11     rm -f calc
12
13 calc : calc.o stack.o token.o
14 #... the rest of the file
```

- `make all` builds the **entire project**, maybe containing **multiple programs**.
  - Should be the default target, so that just `make` works as well.
  - `.PHONY` pseudo-target is never used as default.
- `make clean` **removes generated files**, but keeps the final program(s).
- `make distclean` should leave only what's **needed for distribution**.

Note These names are nothing but agreed-upon conventions, *cf. GNU Coding Standards*[34].

[34] http://www.gnu.org/prep/standards/html_node/Standard-Targets.html

Advanced Makefile for the Calculator

```
1  CFLAGS = -std=c99 -g -Wall -Wextra -Wpedantic -Wbad-function-cast \
2      -Wconversion -Wwrite-strings -Wstrict-prototypes
3
4  SRC = $(wildcard *.c)
5  OBJ = $(patsubst %.c,%.o,${SRC})
6
7  .PHONY: all clean distclean
8
9  all:    calc
10 clean:
11 ⟶       rm -f ${OBJ}
12 distclean: clean
13         rm -f calc
14
15 calc:   $(OBJ)
16 ⟶       gcc -o $@ ${OBJ}
17
18 %.o:    %.c
19 ⟶       gcc -c ${CFLAGS} -c $<
20
21 calc.o: stack.h token.h
22 stack.o:stack.h
23 token.o:token.h
```

Part II
Unix Programming Environment

8
Arguments and Environment

8.1 Command-line arguments

- The function called at program startup is named `main`.
- It shall be defined with a **return** type of `int`, and either zero, or two parameters:

```
1 int main(void);
2 int main(int argc, char *argv[]);
```

Terminology (allthough other names may be used)

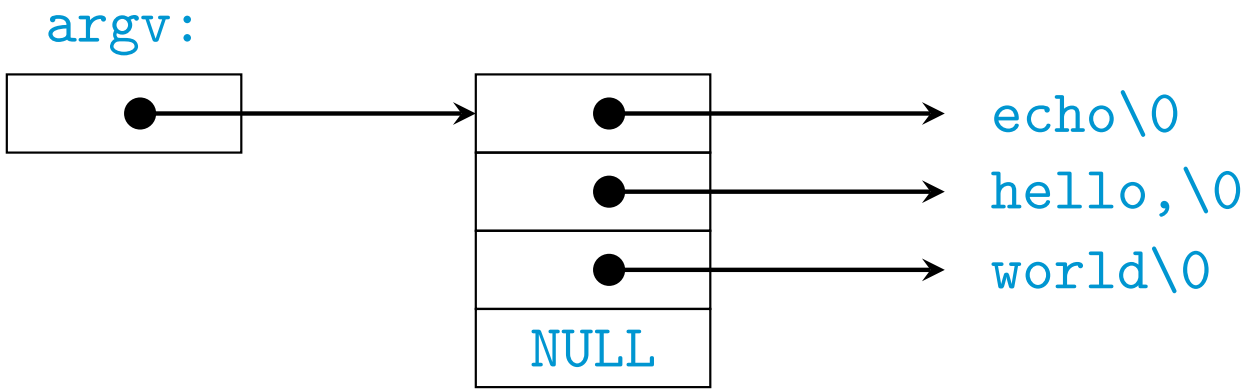
- `argc` stands for *argument count*
- `argv` stands for *argument vector*

If `argc` and `argv` are declared:

- The value of `argc` shall be **nonnegative**
- `argv[0]` represents the **program name** or `argv[0][0]` shall be the null character if the program name is not available.
- `argv[1]` to `argv[argc-1]` represent *program parameters*.
- `argv[argc]` shall be `NULL`, *i.e.*, it may be accessed.

- When a program is executed, the process that starts the new program can pass command-line arguments to it.
- That is the normal operation for UNIX system shells.

```
1 $ echo hello, world
2 hello, world
```



Echo command-line arguments

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++)
        printf("argv[%d]: \"%s\"\n", i, argv[i]);

    return 0;
}
```

```
$ ./a.out dsf ' dfsdf\'' ' t
argv[0]: "./a.out"
argv[1]: "dsf"
argv[2]: " dfsdf\ "
argv[3]: "t"
```

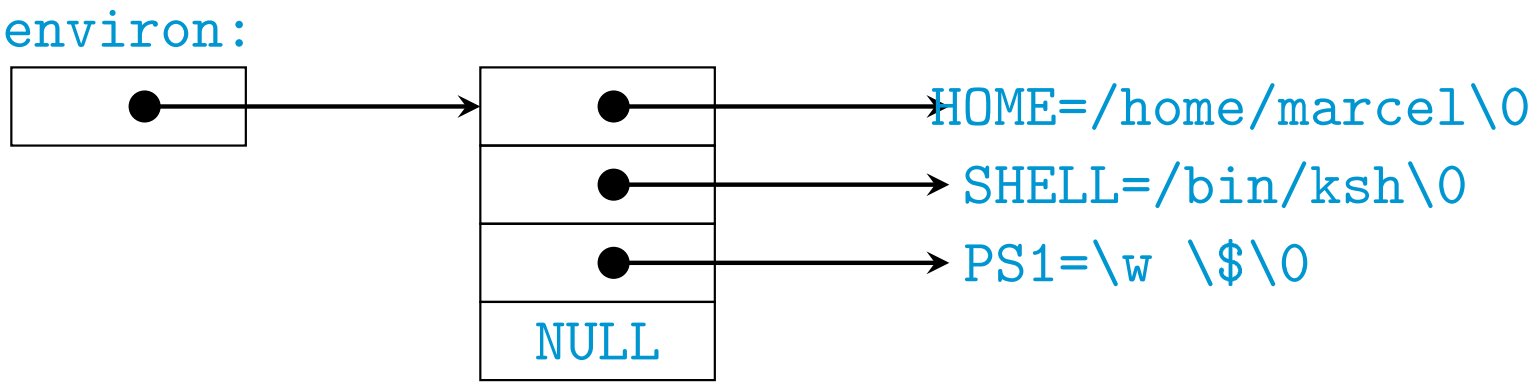
8.2 Environment variables

- Each program is also passed an **environment list**.
- Like the argument list, it is an array of character pointers.
- Each pointing to a null-terminated C string, of the form

  name=value

- The address of the array is contained in a global variable, the **environment pointer**:

```
1 extern char **environ;
```



History There once was an optional third argument to `main`, containing the environment.

Print the environment

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

extern char **environ;

int main(void)
{
    for (char **env = environ; *env; ++env)
        printf("%s\n", *env);

    char *p = getenv("PATH");  /* see getenv(3) */
    if (p)
        printf("Current path is: %s\n", p);

    return 0;
}
```