CS Basics
12) Additional Topics

Fall Term 2023-24

E.Benoist, C.Fuhrer, Ch. Grothoff, L. Ith, P.Mainini | BFH-TI

Additional Topics

- **Macros**
  Conditional Compilation
- **Processes and Pipes**
- **Conclusion**

Macros

Introduction

`#define`: used so far to define symbolic constants

- Numbers and strings to be referred by name
- Often used to configure code behavior in a central place

But can also be used to define *macros*

- Macros look like functions but are different

Differences

- Can contain any valid C expression
- But are only textual substitutions, inserted during preprocessing
- Different from real functions (i.e. no `call` / `ret`)

Example 1

```c
#define area length * width

int main(void) {
  int length = 0, width = 0;

  printf("length = ");
  scanf("%d", &length);
  printf("width = ");
  scanf("%d", &width);

  printf("area = %d\n", area);
  // real code (run "cpp macro1.c"):
  // printf("area = %d\n", length * width);
}
```

Example 2

```
// A rather useless example maybe?

#define BEGIN { printf("Begin!\n");
#define END printf("End!\n"); }

int main(int argc, char *argv[]) {
  if (argc == 2)
  BEGIN
    puts(argv[1]);
  END
}
```

Example 3

```c
#define LOG printf("At position %s:%u\n", __FILE__, __LINE__);

int main(void) {
  LOG
  puts("Some output!");
  LOG
}
```

Output:

```
At position macro3.c:6
Some output!
At position macro3.c:8
```

Macros with Parameters

Syntax

```
#define macro_name([param_list]) replacement_text
```

- Beware: no space between `macro_name` and "`(`"
- Also just performs textual substitution
- Enclose params in `()` when referencing!

Example (`macro4.c`)

```
#define area(x, y) (x) * (y)

int main(void) {

  // ...

  printf("area = %d\n", area(length, width));
  // real code (run "cpp macro1.c"):
  // printf("area = %d\n", (length) * (width));
}
```

Macros with Parameters: Caveats I

```c
#define square(x) x * x

int getv() {
  int x = 0;
  scanf("%d", &x);
  return x;
}

// ...

printf("Square: %d\n", square(getv()));
// real code: printf("Square: %d\n", getv() * getv());
```

Macros with Parameters: Caveats II

```c
#define square(x) x * x

// ...

printf("Square of 6: %d\n", square(3 + 3));
// output: "Square of 6: 15"
```

Macros with Parameters: Solution II

```c
#define square(x) (x) * (x)

// ...

printf("Square of 6: %d\n", square(3 + 3));
// real code:
// printf("Square of 6: %d\n", (3 + 3) * (3 + 3));
```

Macros with Parameters: Caveats III

```
#define printpos(x)                                                    \
  if ((x) > 0)                                                         \
    puts("Positive!")

int main(void) {
  int x = 0;
  if (1 != scanf("%d", &x))
    return EXIT_FAILURE;

  if (x % 2 == 0)
    printpos(x);
  else
    puts("Number is odd...");
}
```

…Results in:

```
if (x % 2 == 0)
if ((x) > 0) puts("Positive!");
else
  puts("Number is odd...");
```

Macros with Parameters: Solution III

A `do...while(0)`-loop can be used to “encapsulate” the macro:

```
#define printpos(x)                                                    \
  do {                                                                 \
    if ((x) > 0)                                                       \
      puts("Positive!");                                               \
  } while (0)

int main(void) {
  int x = 0;
  if (1 != scanf("%d", &x))
    return EXIT_FAILURE;

  if (x % 2 == 0)
    printpos(x);
  else
    puts("Number is odd...");
}
```

Macros over Multiple Lines

Macros can span multiple lines

- Each line must be terminated by a backslash (“\”)
- Newline and white-space is removed by preprocessor

Example

```c
#define printpos(x) \
  do { \
    if ((x) > 0) \
      puts("Positive!"); \
  } while (0)

// real code:
do { if ((x) > 0) puts("Positive!"); } while (0);
```

Token Pasting and Stringify I

is the *stringify operator*

- Unary operator, turning its argument into a string (adds "")

is the *token pasting operator*

- Binary operator, joining both operands into a single token
- If the result contains a macro, it is also expanded

Example

```
#define calculate(exp) printf(#exp " is %d\n", (exp))
#define TEXT_HI "Hello!"
#define TEXT_BYE "Good bye!"
#define say(txt) puts(TEXT_##txt)

int main(void) {
  say(HI);
  calculate(3 * 4 + 3);
```

Token Pasting and Stringify II

```
  say(BYE);
}
```

After preprocessing

```
int main(void) {
  puts("Hello!");
  printf("3 * 4 + 3" " is %d\n", 3 * 4 + 3);
  puts("Good bye!");
}
```

Output

```
Hello!
3 * 4 + 3 is 15
Good bye!
```

Macros: The Joy of `max()`[1]

```
#define __typecheck(x, y) \
    (!!(sizeof((typeof(x)*)1 == (typeof(y)*)1)))
    #define __is_constant(x) (sizeof(int) == \
    sizeof(*(1 ? ((void*)((long)(x) * 0l)) : (int*)1)))
#define __no_side_effects(x, y) \
    (__is_constant(x) && __is_constant(y))
#define __safe_cmp(x, y) \
    (__typecheck(x, y) && __no_side_effects(x, y))
#define __cmp(x, y, op) ((x) op (y) ? (x) : (y))
#define __cmp_once(x, y, op) ({ typeof(x) __x = (x);\
      typeof(y) __y = (y); __cmp(__x, __y, op); })
#define __careful_cmp(x, y, op)                    \
    __builtin_choose_expr(__safe_cmp(x, y),        \
    __cmp(x, y, op), __cmp_once(x, y, op))
#define max(x, y)       __careful_cmp(x, y, >)
```

[1] "That is either genius, or a seriously diseased mind. I can't quite tell which." —Linus Torvalds
See also: `https://lwn.net/Articles/750306/`

Conditional Compilation

The preprocessor can do more than just include files and evaluate macros

Using *conditional compilation*, parts of the code can be removed completely before compiling

- Useful to include sources only once, remove debugging code in production or separate architecture specific code etc.

Preprocessor directives used

- Conditional parts are defined using `#if`, `#elif` (i.e. else if), `#else` and `#endif`
- Any nonzero integer constant value evaluates to `true`
- `#ifdef` and `#ifndef` may be used to test for macro existence (alternatively, the "`defined`" operator may also be used, e.g. `#if defined(x)`)
- Using `#undef`, macro definitions can be unset

Define if not Already Existing

Conditionally define certain parts of code

- Example: `struct car` could already be defined in another module
- Possibly implemented differently

Example

```
#ifndef STRUCT_CAR
#define STRUCT_CAR

struct car {
  char id_number[30];
  char brand[80];
  char type[80];
  double price;
};

#endif
```

Include Guards

Prevent including a file multiple times

- Often used for header files (`.h`)
- Called an *include guard*[2]

Example (`library.h` from c-3 functions example):

```
// header file for library.c
#ifndef H_LIBRARY
#define H_LIBRARY

#include <stdint.h>

uint64_t factorial(uint64_t);
uint64_t logarithm(uint64_t);

#endif
```

[2] Besides using `#ifndef...#endif`, compilers normally also support the non-standard "`#pragma once`" directive

Remove Debug Code

Parts of the code may be enabled/disabled on request

- Adding/removing code for debugging
- Configure code depending on CPU architecture (e.g. autotools)
- ...

Example

```
// some production code above here...
#ifdef DEBUG
  printf("Important variable: %d\n", x);
#endif
// ... and even more production code below here...
```

`DEBUG` can be set while compiling

- `gcc $(CFLAGS) -DDEBUG ...`
- Or as separate target in `Makefile`

```
debug: CFLAGS += -DDEBUG
debug: all
```

Processes and Pipes

Introduction

A *process* is a single, running instance of a program

- Multiple processes of the same program can be running at the same time
- Each process has its own, separate virtual memory with stack, heap, etc.

On POSIX systems, new processes are created by *forking* the current process In C, the function `fork()` can be used for this

- Creates a copy of the current process
- The process from which this function is called becomes the *parent process*
- The newly created process is the *child process*

`fork()`: Spawning a Child Process

Syntax (see "`man 2 fork`" for details)

```
#include <unistd.h>
pid_t fork(void);
```

If the `fork()` function is successful, it returns "twice"

- …once in the child process with return value 0 and
- …once in the parent process with the PID of the child process as return value
- If `fork()` fails, it returns `-1` and `errno` is set accordingly

The child process is a copy of the parent, but there are some differences

- It has a different, unique *Process ID (PID)*
- Its parent PID is set to the PID of the process which created it
- Its resource utilization and CPU time counters are reset to zero
- The set of pending signals is empty
- It does not inherit any timers from its parent

exec(): Replace Program

The `exec()` function (see "`man 3 exec`") replaces the current process by executing a new (different) program Multiple versions exist, depending on desired invocation

- `execl*`: arguments to be provided in argument list, terminated with a `NULL` pointer
- `execv*`: arguments to be provided as NULL-terminated arrays
- `exec*p*`: may search program to execute in `$PATH`
- `exec*e`: allow specifying the *environment*

Returns *only* on error (`-1`, `errno` is set)! Syntax

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...  /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ... /* (char  *) NULL */);
int execle(const char *path, const char *arg,
           ... /* (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Example:

```
execlp("/bin/ls", "/bin/ls", "-al", "/", NULL);
```

Example: `fork1.c`

```c
static int global;

int main(int argc, char *argv[]) {
  (void)argc; // we do not use argc
  int local = 0;

  pid_t pid = fork();
  if (pid < 0) {
    printf("Fork failed!\n");
    perror(argv[0]);
    return errno;
  } else if (0 == pid) {
    // we are the child process
    printf("Child process: local=%d, global=%d\n", ++local, ++global);
  } else {
    // we are the parent process
    local = 10, global = 20;
    printf("Parent process: local=%d, global=%d\n", local, global);
    printf("PID of child: %d\n", pid);
  }
}
```

Output

```
Parent process: local=10, global=20
PID of child: 18844
Child process: local=1, global=1
```

Example: `fork2.c`

```c
#define LOOPS 10
#define ITERATIONS (long)2e9

int main(int argc, char *argv[]) {
  (void)argc; // we do not use argc

  pid_t pid = fork();
  if (pid < 0) {
    printf("Fork failed!\n");
    perror(argv[0]);
    return errno;
  } else if (0 == pid) {
    // we are the child process
    for (int i = 0; i < LOOPS; i++) {
      for (long j = 0; j < ITERATIONS; j++)
        ;
      printf("Child: %d\n", i);
    }
  } else {
    // we are the parent process
    for (int i = 0; i < LOOPS; i++) {
      for (long j = 0; j < ITERATIONS; j++)
        ;
      printf("Parent: %d\n", i);
    }
  }
}
```

Example: `fork2.c` (Output)

```
Child: 0
Parent: 0
Child: 1
Parent: 1
Parent: 2 <-- !
Child: 2
Parent: 3
Child: 3
Parent: 4
Child: 4
Parent: 5
Child: 5
Parent: 6
Child: 6
Parent: 7
Child: 7
Parent: 8
Child: 8
Parent: 9
Child: 9
```

Waiting for Child Completion

`wait()` and `waitpid()` enable a parent process to wait until a child process changes state (e.g. exits)

Syntax

```
#include <sys/wait.h>
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- `pid -1` waits for *any* child process
- `wait(&wstatus)` is equal to `waitpid(-1, &wstatus, 0);`
- If `wstatus` is not `NULL`, status information (e.g. exit status) of the child is written to it and may be analyzed using macros (e.g. `WIFEXITED()`, `WEXITSTATUS()`, …)
- See "`man 2 wait`" for details

Example:

```
int status;
waitpid(child_pid, &status, 0);
```

Pipes

Every open file in a process has a *file descriptor (fd)*, typically a number starting at 0 for the first file opened

Normally, 0 = `STDIN_FILENO`, 1 = `STDOUT_FILENO` and 2 = `STDERR_FILENO` are opened on program start

The `pipe()` syscall creates a unidirectional data channel, called a *pipe*, between two file descriptors

One descriptor is used for writing to- and the other for reading from the pipe

Syntax

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

- `pipefd[0]` is the read end, `pipefd[1]` the write end
- Returns 0 on success, `-1` on error
- See "`man 2 pipe`", for details

★ dup(): Duplicating File Descriptors

- **The `dup` syscall creates a copy of a given file descriptor**
- **Syntax**

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

  - See "`man 2 dup`", for details
- **Example:**

```
int pipefd[2];
pipe(pipefd);
// pipefd contains now 2 new descriptors "piped" together

close(0); // close current standard input
          // alternative: close(STDIN_FILENO);
dup2(pipefd[0], 0); // duplicate read end of pipe to fd 0
close(pipefd[0]); // close unused read end fd

// writing to pipefd[1] now writes to stdin!
```

★ Example: Connecting Processes

Example: Pipe output of one process to another

```
pipe(pipefd);
pid_t pid = fork();
if (pid < 0) {
  printf("Fork failed!\n");
} else if (0 == pid) {
  // we are the child process
  close(pipefd[1]); // parent uses write end
  close(STDIN_FILENO);
  dup2(pipefd[0], STDIN_FILENO);
  close(pipefd[0]);
  // data from parent can now be read from stdin
  // ... continue work ...
} else {
  // we are the parent process
  close(pipefd[0]); // read end belongs to child
  // pipefd[1] can be used to write to child's stdin
  // ... continue work ...
}
```

★ Example: I/O Redirection

Pipes can also be used for file redirection (e.g. redirect a file to `stdin` or `stdout` to a file)

```
int fd = open(filename, O_RDONLY);
close(STDIN_FILENO);
dup2(fd, STDIN_FILENO);
close(fd);
// stdin now reads from file filename!

// ... continue work ...
```

Conclusion

Conclusion

A lot of different topics

- Macros: Powerful text substitution, but tricky
- Conditional compilation: Include or exclude certain parts of the source depending on conditions
- Processes