C Recap for Pintos

Nuri Cingillioglu[1]

https://www.doc.ic.ac.uk/~nuric
Imperial College London

2020

---

[1] thanks to Ioannis Papagiannis, Pedro Mediano, Feroz Salam, Mark Wheelhouse

The C Preprocessor

- The *preprocessing* is the first stage in the compilation of any C program.
- It carries out the tokenization and comment removal.
- In general, directives starting with # are preprocessor instructions.
  - Including the obvious `#include`
- You can call it with `gcc -E`

#define

What?

- `#define identifier replacement`
- Example: `#define PI 3.0`
- Subsequent occurences of `identifier` will be replaced by `replacement`

Usage

- Use `#define` to replace small amounts of code to make it more readable or avoid magic numbers
- Use it to define constants at compile time:
  `#define DEBUG 1`

Macros

What?

- `#define` can also be used with parameters.
- `#define identifier( params ) replacement`
- Example: `#define P(X) printf(‘‘%d’’,(X))`
- Subsequent occurences of `identifier` will be replaced by `replacement`, with `params` substituted in.

Usage

- Use `#define` to replace small amounts of code to make it more readable or define simple functionality
- Macros run like inline functions

Macro Safety

Some general points to be aware of when using macros in C:

- You should surround each term of a macro with parentheses:

```
#define TWICE(x) x * 2
TWICE(3 + 5)
   result:  3 + 5 * 2
```

- You should surround the whole macro replacement with parentheses:

```
#define TWICE(x) (x) * 2
10 / TWICE(5)
   result:  10 / (5) * 2
```

- The correct version of this macro is:

```
#define TWICE(x) ((x) * 2 )
```

Macro Safety contd.

- For macros that execute full statements, a dumb `do-while` loop helps encapsulation:

  ```
  #define P(X) do { printf(``\%d'',(X)); } while (0)
  ```

  (the final semicolon is missing on purpose so you can write `P(n);` in your code)
- Beware: debugging macros can be a nightmare
- Beware: macros do not have types
- You should always try to keep macros simple
- Do not use them as function replacements

Include Guards

Multiple definition problems can occur when multiple files include the same header

- `my_lib.h`

```
  struct my_struct {int x;};
```

- `my_extended_lib.h`

```
#include "my_lib.h"
```

- `my_program.c`

```
#include "my_lib.h"
#include "my_extended_lib.h"
```

Include Guards

- Solution: `#ifdef` and `#ifndef`
- Used to control preprocessing with conditional statements that are evaluated during preprocessing, allowing selective inclusion of code

```
#ifndef MY_LIB_H  /* If not defined... */
#define MY_LIB_H  /* Define the macro MY_LIB_H */

  struct my_struct {int x;};

#endif
```

Conditional Compilation

- `#ifdef` and `#ifndef` are also used for conditional compilation
- Example:

```
struct my_struct {
  int x;
  #ifdef VERBOSE
  char buffer[1000];
  #endif
};
```

Common uses are:

- Debug/test/verbose
- Platform-dependent code
- To address dependency issues

Defining with gcc

- You can also define flags in the terminal.

This:

```
#define DEBUG 1
```

Is the same as this:

```
$ gcc -D DEBUG main.c
```

- Very useful when used in Make or CMake!

Warnings

- Warnings are diagnostic messages that report constructions that are not inherently erroneous but that are *risky* or suggest there may have been an *error*.

By default, always call `gcc` as

```
$ gcc -Wall -Wextra -Werror main.c
```

- **WARNINGS ARE BUGS**, so fix them!

Pointers

- A **pointer** is a special variable type in C
- A pointer contains a memory address you can access through it
- Any variable type has a pointer associated to it

Declaring pointers

- To declare a pointer prepend a ∗ to the variable's name

```
double *doublePtr; /* Pointer to a double */
int *intPtr;       /* Pointer to an int */
int *a, b;         /* Pointer to int ‘a’ and ‘b’ */
int **intPtrPtr;   /* Pointer to pointer to an int */
```

- Pointers can have arbitrary levels of indirection
  – i.e. you can have a pointer to a pointer.

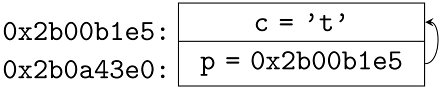
Pointer operators

We mostly use two operators to deal with pointers:

- The **address operator &** takes a value and returns an address.
  Can be read in English as “address of.”
- The **dereference operator *** takes a pointer and returns the value it points to.
  Can be read in English as “content of.”

Pointer operators contd.

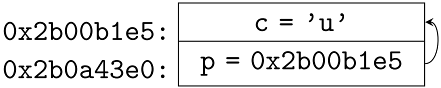
```
char c = 't';
char *p;
p = &c;
```



0x2b00b1e5: c = 't'

0x2b0a43e0: p = 0x2b00b1e5

```
*p = 'u';
```



0x2b00b1e5: c = 'u'

0x2b0a43e0: p = 0x2b00b1e5

Stack and heap

In C we handle two kinds of memory:

The stack

- Handled entirely by the CPU
- Emptied at the end of current scope
- Slightly faster access

The heap

- Handled mostly by the programmer
- Lives forever (or until freed)
- Much bigger space

Memory allocation

Stack

```
float v[10];
```

Heap

```
float *v;
v = (float *) malloc(10*sizeof(float));
if (v == NULL) { PANIC("malloc failed"); } /* Check malloc success */
...
free(v)  /* Remember to free the memory! */
```

- It is good practice to check if `malloc()` succeeded, to avoid surprises later
- Other than that, `float v[]` and `float *v` can *usually* be treated equivalently.
- *But* it can get messy if mixed – agree with your teammates.

Memory leaks

- Failure to free memory will result in a *memory leak*.
- We say a chunk of memory is leaked when it is still reserved but all references to it are lost.

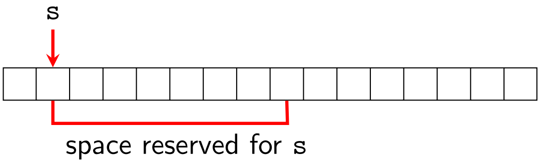
```
void f(void) {
  float *a = malloc(sizeof(float) * 5);
  foo(a);
}
```

- Use tools like valgrind to detect memory leaks.
- Or simply comment the free for every malloc

Pointers and arrays

- There's no array type in C
- We use pointers with reserved memory locations

```
char *s = (char *) malloc(8*sizeof(char));
```



- For any integer `n`, `*(s+n)` is equivalent to `s[n]`

Pointer Arithmetic

When adding to pointers, the type of the pointer is important.

```
struct my_struct {
    int a;
    int b;
};
```

```
struct my_struct s[2];
struct my_struct *sp = &s[0];
int *ip = (int *) sp;
```

```
sp++; /* now points to the second my_struct in s */
ip++; /* now points to second int of first my_struct */
```

A word about const

- With pointers, you can have constant pointers, pointers to constant values or both.

```
int val = 5;
const int *ptr1 = &val;       /* ptr1 can change, val cannot */
int *const ptr2 = &val;       /* val can change, ptr2 cannot */
const int *const ptr3 = &val; /* neither val nor ptr3 can change */
```

- Very powerful resource.
- Gets messy *very* quickly with multiple levels of indirection.
- Use const where you can...

Pointers as arguments

Pass by reference

When you want to modify an argument *inside* the function, pass a pointer.

```
void setInt(int *v, int i) { *v = i; }
```

Passing pointer-const instead of value

Very useful trick!

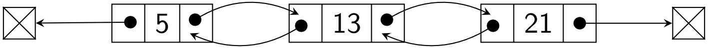
```
void foo(int n, const float *X)
```

Benefits:

- Enforce const-ness of input
- Avoid potentially expensive useless copies

Linked lists

- Anatomy of a (doubly) linked list:

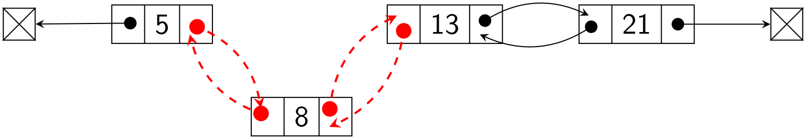


List element structure

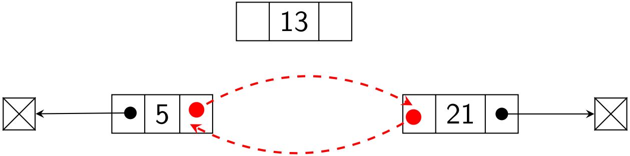
```
struct list_elem {
  struct list_elem *prev; /* Previous list element. */
  struct list_elem *next; /* Next list element. */
};
```

Operations on linked lists

- Insertion:



- Deletion:



Implementation of a linked list

List element structure

```
struct list_elem {
  struct list_elem *prev; /* Previous list element. */
  struct list_elem *next; /* Next list element. */
};
```

Declaring a struct to be used in a list

```
struct my_element {
  int a;
  struct list_elem item; };
```

The list structure itself

```
struct list {
  struct list_elem head;      /* List head. */
  struct list_elem tail;      /* List tail. */
```

Lists in Pintos

- It's **very** important that you understand how to use lists in Pintos
- The Pintos list implementation in `<list.h>` contains several useful functions.
- Don't reinvent the wheel – **use them!**

Declaring and initialising a list

```
struct list my_list;
list_init(&my_list);
```

Lists in Pintos contd.

Inserting an item

```
/* thing_before is the list_elem of the struct that
   you want to insert the new thing item after */
list_insert (&(thing_before.item) , &(thing.item));
```

Fetching the front element

```
struct list_elem *my_item = list_front (&my_list);
```

Casting a list_elem to its parent struct

```
struct my_element *thing =
    list_entry(my_item, struct my_element, item);
```

Function Pointers

Just like you can have a pointer to a variable in C, you can also have a pointer to a function.

Declare Function Pointer

```
int (*fp)( int );
```

Define Functions

```
int f1( int x ) { return x; }
int f2( int x ) { return x+1; }
```

Pair Function Pointer to Function

```
fp = f1;        /* fp = &f1; also accepted */
```

Execute Function Using Pointers

```
int r = (*fp)( 5 );
```

Function Pointers contd.

- Function pointers are everywhere in the standard library
- Example: sort array wrt function `compar`:

```
void qsort(void *base, size_t nitems, size_t size,
  int (*compar)(const void *, const void*))
```

- They're also in Pintos:

```
void list_sort (struct list *,
                list_less_func *, void *aux);
```

- Function pointers are cool and make other functions generic.

`typedef` and function pointers

- Declaring function pointers can get cumbersome
- Enter `typedef`
- Example from Pintos:

```
typedef bool list_less_func(const struct list_elem *a,
                            const struct list_elem *b,
                            void *aux);
```

- Careful with the naming. It's very easy to lose track and confuse them with variable pointers

C99 data types

Booleans

- `stdbool.h` defines type ‘bool’
- ‘true’ expands to 1
- ‘false’ expands to 0
- Examples of use in code same as in any other language, really

The `stdbool.h` source

```
#define bool _Bool
#define true 1
#define false 0
```

Coding Standards

Maintaining a consistent coding standard will help you:

- Debug large blocks of code
- Work with other people's code
- Get more marks
  (remember we're marking the style of your code!)

Coding Standards

Some general points to keep in mind:

- Comment your code
  (but don't leave old code commented-out in your source files, this will only be confusing for the markers *and* your group)
- Common sense goes a long way, if you think something is messy or over-complicated, it probably is!
- Bear in mind the golden rules: **KISS**, **RTFM**, **DRY**
- Write code that tells you *how*
  and comments that tell you *why*

The End

Don't be afraid

We had a student finish Pintos alone in 3 days. Don't do this!

Useful books

- *The Pragmatic Programmer*, by Hunt and Thomas
  Addison Wesley, 1st edition
- *The C Programming Language*, by Kernighan and Ritchie
  Prentice Hall, 2nd edition
- *C Traps and Pitfalls*, by Andrew Koenig
  Addison Wesley, 1st edition