TDT4200 Parallel programming

Introduction to C

Maren Wessel-Berg

August 2023

Main focus of this introduction

- The basic syntax of C looks a lot like other statically typed languages like Java.
  - Will cover: Hello world example and `printf`.
  - Will not cover: If-loops, for-loop, switch statements, functions, etc.
- Important concepts for working with C code.
  - Memory regions.
  - Scope and lifetime of variables.
  - Dynamic memory management.
  - Handling pointers, structs and arrays.
  - The preprocessor and macros

Note: Some of the slides are based on other courses around the world—Stanford, University of Edinburgh, NTNU, etc.

Hello World

- Install gcc or another C-compiler.
- Write some code in `hello.c`:

```
#include <stdio.h>

int main ( int argc, char** argv ) {
    printf ( "Hello world\n" );
    return 0;
}
```

- Inside of a terminal/shell:

```
gcc hello.c -o hello
./hello
```

printf

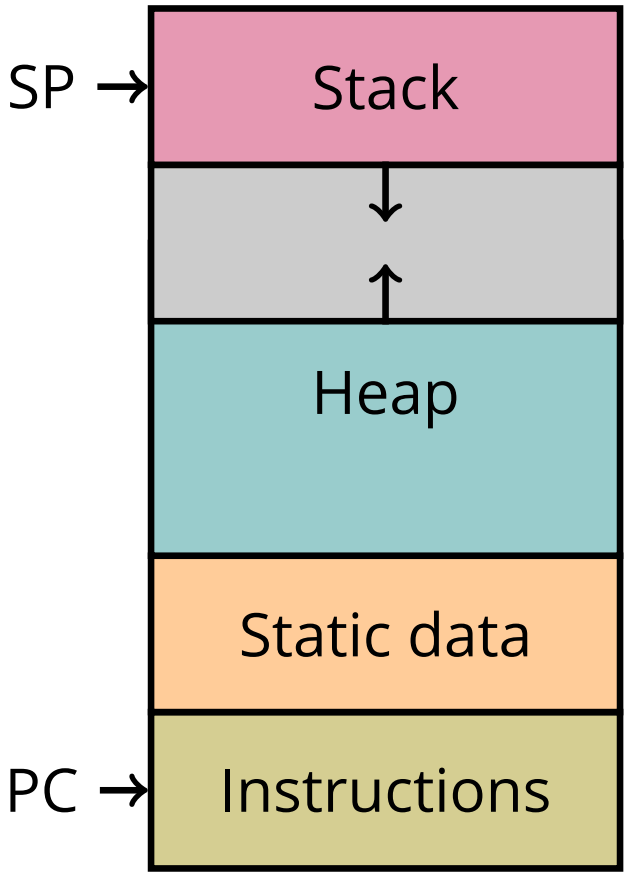
```
printf ( "These␣%d␣+␣%lf␣%s.\n", 3, 1.0,
            "symbols␣are␣mostly␣what␣you␣need");
```

- `%d` (digit) for integers
- `%lf` (long float) for doubles
- `%s` for strings
- `\n` for newline

Philosophy of C

- Assumes competent programmers who deserve full freedom and access.
- With great power comes great chance of weird bugs, so be careful.

Memory regions



- *Stack*

  For data local to functions.
  Managed by the compiler.
- *Heap*

  For dynamically allocated data.
  Managed by the programmer (in C) or by the system (in Java).
- *Static*

  For data with program lifetime.
  Initialised when the program starts.

Memory regions and memory management

- In Java
  - All objects are dynamically allocated in the heap.
  - Dynamically allocated objects in the heap are **recycled automatically** by the garbage collector when they are no longer needed.
- In C
  - No objects, only data structures in all 3 memory regions.
  - Some data structures are statically allocated, others dynamically.
  - Programs must **explicitly manage dynamically allocated data structures in the heap**.
  - Explicit memory management is a major source of error, particularly when the programmer forgets to free the memory, which results in memory leaks.

Stack and local variables

- When you have some statements in C, a { basic block } defines a local scope for the variables.

```
{
int a = 1, b = 0;

    {
        int a = 64;
        b = a - 32;
    }

printf ("a is %d, b is %d\n", a, b);
}
```

- `a` and `b` **live on the stack and disappear when the local scope is exited**.
- This code will print “a is 1, b is 32”; the `a` declared inside the basic block shadows the exterior `a`, but is gone when the block ends.

Static data region and global variables

```
#include <stdio.h>

int x = 32;

int main ( int argc, char **argv ) {
    print_x_value();
}

void print_x_value ( void ) {
    printf ( "x is %d\n", x );
}
```

- This works because `x` **lives in the static data region of memory** and is in the object code's name table.
- `x` **has lifetime of the program and scope of the file**.

Pointers, in principle

Before we get into explicit memory management, we need some notation for writing about memory.

- A C pointer is a variable that holds **the memory address of a piece of data**.
- You can examine its value, but you will only see a Large Meaningless Integer, which isn't very informative.
- The idea is to have the *value* of one memory location coincide with the *index* of another memory location:

Location	0×0	0×1	0×2	0×3	0×4	0×5
Value	0×2	20	0×5	12	64	42

  - Location 0×2 contains a pointer, and it points at the value 42 (in location 0×5)
  - Location 0×0 contains a pointer-to-a-pointer, it points at the value 0×5 (in location 0×2), which in turn points at the value 42 (in location 0×5)
  - We could continue...

Pointers, in practice

- The concept itself is not too bad, but manipulation of pointers can get tricky:

```
double value = 0.0; // A double-precision scalar
double *ptr1;       // A pointer to a double-precision scalar
double **ptr2;      // A pointer-to-a-pointer to a double-precision scalar

ptr1 = &value;      // Get the address of our 0.0-valued variable
ptr2 = &ptr1;       // And get the address of its address

value = 3.14;       // Assign with 0 levels of indirection
*ptr1 = 2.71;       // Assign with 1 level of indirection
**ptr2 = 3.14;      // Assign with 2 levels of indirection
```

Note that all the assignments change the same memory location, e.g., after `*ptr1=2.71`, printing `value` will give you `2.71`.

Pointers, in practice

- Mind the lifespan of what you find the `&` of– if it lives on the stack, it will disappear with the block it lives in.
- In particular, don't have functions return pointers to their own local values— those disappear at return

Dynamic memory allocation

- The library function `malloc` allocates a chunk of memory at run-time and returns the address.
- The library function `free` can be used to release the chunk of memory allocated by `malloc`.

```
// Declare an int pointer
int *p;

// Allocate memory for n ints
p = malloc ( n * sizeof(int) );

// Check if the memory allocation was successful
if ( !p ) {
    // Error
}

// Release the allocated memory
free ( p );
```

Segmentation faults

- Error that occurs if (when) you try to **access memory you did not allocate**, i.e., memory you do not 'own'.
- Causes the program to crash.
- Solution
  - Quick-fix: print statements
  - Robust fix: **debugger**

What about the lifetime and scope of dynamic heap variables?

- Local variables allocated on stack disappear when we are outside their local scope.
- Variables dynamically allocated in the heap using `malloc` do not.

What about the lifetime and scope of dynamic heap variables?

```
int* make_variable ( void ) {
    int local = 3;
    int* ret =
        (int*) malloc ( sizeof(int) );
    *ret = 42;
    return ret;
}

int main ( int argc, char** argv ) {
    int local = 2;
    int* ptr = make_variable();

    // prints "values: 2, 42"
    printf ( "values: %d, %d\n",
        local, *ptr );
    return 0
}
```

Arrays

Arrays...

- are fixed size, where the size is indicated on construction.
- have no knowledge of their size when they are out of scope.
- have no built in check to see if indices are within bounds.

```
// Create array of length 3
int m[] = {5, 8, 10}
// Access the array (n = 5)
int n = m[0];
// Error
int o = m[4];
```

Arrays- are they simply pointers?

- Yes. Almost.
- There is a close relationship between arrays and pointers.
- Pointers are also commonly used to pass arrays between functions.

```
// Construct array of 15 ints
int my_arr[15];

// Access value
my_arr[4] = 42;

// Square brackets is simply
// syntactic sugar for
// pointer arithmetic.
// We could do this instead:
*(my_arr + 4) = 42;
```

Type definitions

Define names for user-defined or built-in types.

```
typedef <type> <name>;
```

- Convenient for hiding complexity and for configuration.

```
typedef double real_t;
typedef int64_t int_t;
```

Structs and user-defined types

Structs are **collections of variables**.

- Convenient for organizing data.

```
struct options_struct {
    int_t N;
    int_t max_step;
    int_t snapshot_frequency;
};

options_struct opts;
options_struct* opts_ptr = &opts;

// Access struct member
opts.N = 10;
// Dereference pointer
// and access struct member
int n = opts_ptr->N;
```

Structs and user-defined types

Structs are **collections of variables**.

- Convenient for organizing data.

```
typedef struct options_struct {
    int_t N;
    int_t max_step;
    int_t snapshot_frequency;
} OPTIONS;

OPTIONS opts;
OPTIONS* opts_ptr = &opts;

// Access struct member
opts.N = 10;
// Dereference pointer
// and access struct member
int_t n = opts_ptr->N;
```

Macros

Macros are initialised with the directive `#define`, and tells **the preprocessor** to replace any occurance of the macro with the value of the macro. This happens **at compile-time**.

- Convenient for clairity.

```
#define PI 3.14
double circle_area ( double r ) {
    return r*r*PI;
}
```

- Convenient for configuration.

```
#define THRESHOLD 0.001
void func ( double x ) {
    if ( x < THRESHOLD ) { ... }
    else if ( x > THRESHOLD ) { ... }
    else { ... }
}
```

Macros

- Convenient for hiding complexity.

```
#define SQUARE(x) ((x)*(x))
int circle_area = SQUARE(3) * 3.14;
```

- Why would this not work?

```
#define BAD_SQUARE(x) x*x
int circle_area = BAD_SQUARE(2+5) * 3.14;
```

Macros

- Convenient for hiding complexity.

```
#define SQUARE(x) ((x)*(x))
int circle_area = SQUARE(3) * 3.14;
//int circle_area = ((3)*(3)) * 3.14;
```

- Why would this not work?

```
#define BAD_SQUARE(x) x*x
int circle_area = BAD_SQUARE(2+5) * 3.14;
// int value = 2+5*2+5 * 3.14;
```

Extra material

- General walkthrough of C:
  `https://www.youtube.com/watch?v=3lQEunpmtRA`
- Video on understanding pointers:
  `https://www.youtube.com/watch?v=2ybLD6_2gKM`