CS Basics
10) Pointers

Fall Term 2023-24

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

Pointers

- **Basics**
- **Dynamic Memory Allocation**
- **Multidimensional Arrays**
- **Function Pointers**
- **Command Line Arguments**
- **Conclusion**

Basics

Introduction

A pointer “points” to a memory address

- References the *location* where a value is stored (and not the value itself!)
- You may think of it as a label in assembly

Pointers are used frequently in C

- For accessing and iterating over data in memory (e.g. linked lists, trees)
- *Function pointers* (pointers to a function) can be used as function parameters
- Pointers enable functions to modify data *in-place* and to “return” multiple data items at once (e.g. `scanf()`)

Recap: Variables

A variable represents data which resides in memory Automatic variables

- Default type in functions and compound statements
- Only accessible within their scope
- Undetermined initial value
- Memory is released when the variable is not needed anymore

Static and global Variables

- Static variables within functions are defined using the `static` keyword
- Global variables are defined outside of any function
- Both are initialized once at program start and remain allocated for the whole program duration
- They are stored in the `.data` or `.bss` segment

The Operators `&` and `*`

Two operators are used for working with pointers

- `&`: *Address operator*, returns the memory address of an object
- `*`: *Indirection operator* (*dereferencing operator*), used for accessing an object

Basic principle

- Given a variable `v`
- …the address of `v` is `&v`
- …which can in turn be assigned to a variable: `pv = &v;`
- `pv` is called a *pointer to* `v`

Dereferencing

- The value of the variable where `pv` *points to* can be accessed by *dereferencing*: `*pv`
- Note: accessing `*pv` is equivalent to accessing `v`

Declaring Pointers

Variables which hold a pointer to some `type` are declared using an asterisk (`*`)

```
type  *pvar; // pointer to a variable of type "type"
```

Example

```
double my_double = 0.42;
double *my_double_ptr = &my_double;
```

`void` and `NULL` Pointers

`void *`: Pointers to `void`, or *void pointers* for short

- There is no data type `void` in C
- `void *` is used as a pointer to *any* type
- Used e.g for parameters of functions which can handle different data types
- As the type is missing, *void pointers cannot be dereferenced without casting*

`NULL` pointers

- A pointer to the address `0` is not a valid pointer
- Often used for indicating errors, e.g. by functions returning pointers
- `stdio.h`, `stdlib.h` and others define the macro `NULL`, which *should* be used for this purpose

Example: `pointers1.c` I

```c
uint32_t my_int = 42;
uint32_t *my_int_ptr = &my_int;
printf("%d (%p: my_int)\n", my_int, &my_int);
printf("%d (%p: my_int_ptr)\n", *my_int_ptr, &my_int_ptr);

uint32_t copy_of_int = *my_int_ptr;
printf("%d (%p: copy_of_int)\n\n", copy_of_int, &copy_of_int);

/*
  Output:
  42 (0x7ffd56a2062c: my_int)
  42 (0x7ffd56a20620: my_int_ptr)
  42 (0x7ffd56a2061c: copy_of_int)
*/
```

Example: `pointers1.c` II

```c
my_int = (*my_int_ptr / 2) * 3;

printf("%d (%p: my_int)\n", my_int, &my_int);
printf("%d (%p: my_int_ptr)\n", *my_int_ptr, &my_int_ptr);
printf("%d (%p: copy_of_int)\n", copy_of_int, &copy_of_int);

/*
  Output:
  63 (0x7ffd56a2062c: my_int)
  63 (0x7ffd56a20620: my_int_ptr)
  42 (0x7ffd56a2061c: copy_of_int)
*/
```

Pointers as Function Parameters

C does *call-by-value* when invoking functions

- Values of function parameters (arguments) are stored as automatic variables
- Changing a variable does not affect the caller's variable
- Requires additional memory, copying may be inefficient…

Using pointers as parameters enables *call-by-reference*

- Only the address of an object is copied
- Function directly accesses the caller's variable (and may change it!)

Syntax

```
type fname(type1 *par1, type2 *par2) {
  // ...
}
```

Example: pointers2.c

```c
void call_by_val(uint32_t par1, uint32_t par2) { par1 = par2 = 42; }

void call_by_ref(uint32_t *par1, uint32_t *par2) { *par1 = *par2 = 42; }

int main(void) {
  uint32_t x = 0, y = 0;
  printf("x: %d, y: %d (initial values)\n", x, y);

  call_by_val(x, y);
  printf("x: %d, y: %d (after call_by_val)\n", x, y);

  call_by_ref(&x, &y);
  printf("x: %d, y: %d (after call_by_ref)\n", x, y);
}

/*
  Output:
  x: 0, y: 0 (initial values)
  x: 0, y: 0 (after call_by_val)
  x: 42, y: 42 (after call_by_ref)
*/
```

Example: `multiply.c`

```c
void multiply(int32_t *num1, int32_t *num2) { *num1 = *num1 * *num2; }

int main(void) {
  int32_t x = 0, y = 0;

  puts("Please enter two numbers:");
  if (2 != scanf("%d %d", &x, &y)) // Notice: call-by-reference
    return EXIT_FAILURE;

  multiply(&x, &y);
  printf("x: %d, y: %d\n", x, y);
}

/*
  Output:
  Please enter two numbers:
  3 15
  x: 45, y: 15
*/
```

Pointer Arithmetic

Besides assignment and dereferencing, pointers support arithmetic

- Calculations are automatically adapted to object size
- *Beware: Frequent source for programming errors!*

An integer can be added/subtracted to/from a pointer

- Increment/decrement pointer: `++ptr` or `--ptr`
- Address `i`'th object away from pointer: `(ptr + i)` or `(ptr - i)`

A pointer can be subtracted from another pointer

```
ptrdiff_t distance = ptr1 - ptr2;
```

Two pointers can be compared (==, <, ...)

Arrays and Pointers

An array variable is actually a pointer to the first element of the array!

- If `x` is an array: `&x[0]` (or simply `x`) is the address of the first element
- Then, `&x[1]` or `x+1` is the address of the second element; in general: `&x[i]` is `x+i`
- *Note: Arithmetic on array addresses respects element size!*

Accessing an array element

- There is no difference when accessing elements using pointers: `x[i]` and `*(x+i)` are equivalent

But: array *names* are in general not modifiable

```
char my_arr[20] = {0};
++my_arr; // pointer arithmetic does not work here!
```

Array Element Addressing

- **All of the following expressions are equivalent and copy an element from `orig` to `copy`:**[1]

```
copy[i] = orig[i];
copy[i] = *(orig + i);
*(copy + i) = *(orig + i);
*(copy + i) = orig[i];
```

- **A pointer may point to elements *inside* of an array:**
  - Example: substring without copying

```
char line[] = "Hello!";
char *sub1 = line + 2;
char *sub2 = &line[2];

printf("%s %s %s\n", line, sub1, sub2);
// Hello! llo! llo!
```

[1] See also `examples/arrays3.c`

Example: arrays1.c

```c
int main(void) {
  uint32_t array[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
  uint32_t *pointer = array;

  for (int i = 0; i < 10; i++) {
    printf("i = %d, array[%d] = %d, *(pointer + %d) = %d\n", i, i, array[i], i,
           *(pointer + i));
    printf("  &array[%d] = %p, pointer + %d = %p\n", i, &array[i], i,
           pointer + i);
  }
}

/*
  Output:
  i = 0, array[0] = 10, *(pointer + 0) = 10
    &array[0] = 0x7ffe55200a90, pointer + 0 = 0x7ffe55200a90
  i = 1, array[1] = 11, *(pointer + 1) = 11
    &array[1] = 0x7ffe55200a94, pointer + 1 = 0x7ffe55200a94
  ...
*/
```

Pointer Subtraction (arrays2.c)

```c
int main(void) {
  uint32_t array[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
  uint32_t *ptr1 = array, *ptr2 = &array[5];

  printf("ptr1 = %p (%d), ptr2 = %p (%d)\n", ptr1, *ptr1, ptr2, *ptr2);
  printf("ptr2 - ptr1 = %ld\n", ptr2 - ptr1);
}

/*
  Output
  ptr1 = 0x7ffd7a544310 (10), ptr2 = 0x7ffd7a544324 (15)
  ptr1 - ptr2 = 5
*/
```

Dereferencing: Beware of Types!

Dereferencing as incorrect (pointer-) type leads to undefined behavior! Check out `examples/dereferencing.c`

```
/*
  *pi   (0x7ffe5cff868c) =    1 (int* as int)
  *pi   (0x7ffe5cff868c) = 0.30 (int* as float -> NONSENSE)
  *pi   (0x7ffe5cff868c) = 1.00 (int* cast to float)

  *pi-1 (0x7ffe5cff8688) =    0 (float as int via int*)
  *pi-1 (0x7ffe5cff8688) = 1.00 (float as float via int*)
  *pi-1 (0x7ffe5cff8688) = 0.30 (float via int* cast to float*)
*/
```

Dynamic Memory Allocation

Introduction

So far, there are two options for storing arrays

- As *automatic variables*, for the duration of a function or compound statement
- As *global* or *static variables*, stored during the entire program duration
- Problem: Requires a priori knowledge of the *array size*

Often, space required during execution is not known

- May be depending on program input
- Can change dynamically
- May be too large for the stack

Solution: *dynamic memory allocation*

- Request some space in free memory, called the *heap*
- Allocation duration does not depend on any scope
- Can be used from anywhere in the program
- Is addressed using a pointer
- *Must be freed* by the programmer when not used anymore

`malloc()` and `calloc()`

The functions `malloc()` and `calloc()`

- Defined in `stdlib.h`
- Allocate a new block of memory with given size
- Return a `void *` to the block or `NULL` in case of error

Why `calloc()`?[2]

- Prefer using `calloc()` instead of `malloc()`
- Checks for overflows in allocation size
- *Zeroes-out* the allocated memory

Syntax

```
void *malloc(size_t size)
void *calloc(size_t count, size_t size)
```

[2] More details: `https://vorpus.org/blog/why-does-calloc-exist/`

`realloc()` and `free()`

The function `realloc()`

- Can be used to resize a block of memory obtained by `malloc()` or `calloc()`
- Either original block is resized or a new block is allocated
- Contents are preserved (up to the size of the smaller block)
- *Old pointer must not be used after* `realloc()` *(undefined behavior!)*
- Also returns a `void *` to the block or `NULL` in case of error

```
void *realloc(void *ptr, size_t size)
```

The function `free()`

- Releases a dynamically allocated block of memory
- *Unused memory must be freed to prevent memory leaks*

```
void free(void *ptr)
```

Example: `dynamic.c`

```c
  double *ptr1 = calloc(1, sizeof(double));
  if (NULL == ptr1)
    return EXIT_FAILURE;

  *ptr1 = 42.23;
  printf("%.2f\n", *ptr1);

  double *ptr2 = realloc(ptr1, 2 * sizeof(double));
  if (NULL == ptr2)
    return EXIT_FAILURE;

  *(ptr2 + 1) = 0.07;
  printf("%.2f, %.2f\n", ptr2[0], ptr2[1]);

  free(ptr2);

/*
  Output:
  42.23
  42.23, 0.07
*/
```

Example: `grades.c` I

```c
double *grades = calloc(1, sizeof(double));
if (NULL == grades)
  return EXIT_FAILURE;

puts("Enter your grades, end with Ctrl-D:");

uint32_t nb_grades = 0;
while (1 == scanf("%lf", &grades[nb_grades])) {
  nb_grades++;
  grades = realloc(grades, (nb_grades + 1) * sizeof(double));
  if (NULL == grades)
    return EXIT_FAILURE;
}
```

Example: `grades.c` II

```c
if (nb_grades == 0) {
  puts("No grades entered!");
  free(grades);
  return EXIT_SUCCESS;
}

puts("\nYou have entered the following grades:");
double sum = 0;
for (uint32_t i = 0; i < nb_grades; i++) {
  printf("%.1f\n", grades[i]);
  sum += grades[i];
}

free(grades);

printf("Your average grade is: %.1f\n", sum / nb_grades);
```

Array Allocation Using Macros

Why write so much text?

- C has a preprocessor to take care of that!

Example

```
// do this once, ideally in a header; existing
//   C code likely has something like this already
#define new_array(n, t) (t *)calloc(n, sizeof(t))

double *grades;
grades = new_array(LENGTH, double);
if (NULL == grades)
  // error handling
```

Multidimensional Arrays

Introduction

One-dimensional arrays can be represented as pointer with offset

- Element `i` in `array`: `*(array + i)`

Multidimensional arrays can also be represented using pointer notation

- E.g. a two-dimensional array is an array of one-dimensional arrays…
- …or also a pointer to a group of one-dimensional arrays…
- …which in turn are pointers to a group of elements

Syntax

The following statements all return the second element of the second array:

```
array[1][1]
(*(array + 1))[1]
*(array[1] + 1)
*(*(array + 1) + 1)
```

Example: `multidimensional.c` I

```c
// allocate an array, holding an (uint32_t *) for every row
uint32_t **array = calloc(rows, sizeof(uint32_t *));
if (NULL == array)
  return EXIT_FAILURE;

// for every row, allocate an array with an (uint32_t *) for every column
for (uint32_t r = 0; r < rows; r++) {
  array[r] = calloc(cols, sizeof(uint32_t));
  if (NULL == array[r])
    // note: no proper free()'ing here
    return EXIT_FAILURE;
}

puts("\nEnter values:");
for (uint32_t r = 0; r < rows; r++) {
  for (uint32_t c = 0; c < cols; c++) {
    printf("[%d][%d]: ", r, c);
    scanf("%u", *(array + r) + c);
```

Example: `multidimensional.c` II

```
    }
  }

  puts("\nAccess with array notation:");
  for (uint32_t r = 0; r < rows; r++) {
    for (uint32_t c = 0; c < cols; c++) {
      printf("array[%d][%d] = %d\n", r, c, array[r][c]);
    }
  }

  puts("\nAccess using pointers:");
  for (uint32_t r = 0; r < rows; r++) {
    for (uint32_t c = 0; c < cols; c++) {
      printf("*(*(array + %d) + %d) = %d\n", r, c, *(*(array + r) + c));
    }
  }
```

Example: `multidimensional.c` III

```c
// free all rows individually
for (uint32_t r = 0; r < rows; r++) {
  free(array[r]);
}

// free the row array
free(array);
```

On-the-fly Allocation

The full array needs not to be allocated at once

- First, prepare an array to hold later allocations

```
char *array_of_strings[MAX_COUNT];
```

- Later, when required, a new array may be allocated

```
array_of_strings[i] =
    calloc(STRING_LEN, sizeof(char));
if (NULL == array_of_strings[i])
  // error handling
```

“Unlimited” arrays (without `MAX_COUNT`) are also possible

- Using `realloc()`, see Slide 25 for an example.

Function Pointers

Function Pointers

As functions also have addresses, *function pointers* can point to them

- Enables to pass functions to other functions as parameter
- Functions may also be stored in arrays or other data structures (e.g. for a lookup table)

Defining function pointers

```
// without any parameters
type (*fp_name)()

// just with parameter types
type (*fp_name)(type1, type2, ..., typeN)

// with parameter types and names
type (*fp_name)(type1 par1, type2 par2, ..., typeN parN)
```

Invocation: `fp_name(...)` **is equal to** `(*fp_name)(...)`

Example: `funcptr1.c`

```c
void print_even(uint32_t num) { printf("Even: %i\n", num); }

void print_odd(uint32_t num) { printf("Odd: %i\n", num); }

void print_square(int32_t num, void (*printer)(uint32_t)) {
  uint32_t square = num * num;
  printer(square); // <=> (*printer)(square)
}

int main(void) {
  int32_t num = 0;
  printf("Enter a number: ");
  if (1 != scanf("%d", &num))
    return EXIT_FAILURE;

  if ((num % 2) == 0)
    print_square(num, print_even);
  else
    print_square(num, print_odd);
```

Interlude: `typedef`

`typedef` may be used to define new types

Used a lot in (standard-)libraries, e.g. in `stdint.h`

```
typedef unsigned char uint_fast8_t;
```

Useful for defining function pointer types

```
typedef void (*t_printer)(uint32_t)

void print_square(int32_t num, t_printer printer) {
  // ...
}
```

Beware: the suffix "`_t`" should be avoided, as it may conflict with future C versions or POSIX[3]

[3] See also: `https://www.gnu.org/software/libc/manual/html_node/Reserved-Names.html`

Example: `funcptr2.c`

```c
int32_t func(int32_t a, int32_t b) {
  printf("%d+%d=%d\n", a, b, a + b);
  return a + b;
}
int main(void) {
  // normal function call
  func(1, 2);

  // pointer to the function
  int32_t (*ptr1)(int32_t, int32_t) = func;
  ptr1(2, 3);

  // pointer to a function type
  typedef int32_t t_func(int32_t, int32_t);
  t_func *ptr2 = func;
  ptr2(3, 4);

  // function pointer type
  typedef int32_t (*t_func_ptr)(int32_t, int32_t);
  t_func_ptr ptr3 = func;
  ptr3(4, 5);
```

Command Line Arguments

Command Line Arguments

Command line arguments are passed to the program on the stack (as we have seen in the assembly part)

To evaluate the arguments in C, `main()` may define two parameters

- `int argc`: the number (count) of arguments
- `char* argv[]`: the argument values as array of strings

Syntax

```
int main(int argc, char* argv[]) {
  ...
}
```

Example: arguments1.c

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

/*
  Output:
  $ ./arguments1 arg1 arg2
  Command line arguments:
  argv[0] = "./arguments1"
  argv[1] = "arg1"
  argv[2] = "arg2"
*/
```

Numerical Arguments

- `getopt()` **(in** `unistd.h`**) may be used for advanced argument parsing (see** `man 3 getopt`**)**
- **To parse numerical arguments,** `sscanf()` **may be used**

```c
int main(int argc, char *argv[]) {
  if (argc != 3)
    goto error;

  int32_t num1 = 0, num2 = 0;
  if (1 != sscanf(argv[1], "%d", &num1))
    goto error;
  if (1 != sscanf(argv[2], "%d", &num2))
    goto error;

  printf("%d * %d = %d\n", num1, num2, num1 * num2);
  return EXIT_SUCCESS;

error:
  printf("Usage:\n%s <num1> <num2>\n", argv[0]);
  return EXIT_FAILURE;
}
```

Conclusion

Conclusion

Pointers

- Pointers contain addresses
- They can be used as function parameters
- or as function return values

Call-by-reference

- Function can modify the content of a variable
- The content is not copied (saves time / space)
- Variable may be modified unexpectedly

Array = pointer

- An array is a pointer
- Manipulation of pointers for accessing an array

Pointers to functions

- Variables can point to function
- `typedef` can be used to give new types a name