Pointers and dynamic memory management in C

Jakob Rieck

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

08.05.2014
Agenda

1. Memory layout
2. Pointers
3. Dynamic memory management
4. Literature
Outline

1 Memory layout
   - The stack
   - The heap

2 Pointers

3 Dynamic memory management

4 Literature
Overview

Figure: http://infohost.nmt.edu/~eweiss/222_book/222_book/0201433079/ch07lev1sec6.html
### The stack

- Used for **local variables** in C
- Lightweight LIFO data structure
  - Very fast (de-)allocation
- Automatic (de-)allocation of variables
  - Out of scope, out of reach
- (Severely) space constrained
void bar() {
    int j = 42;
}

void foo() {
    bar();
}

int main() {
    int i = 42;
    foo();
    return 0;
}

High level stack layout

Local variables

internal state
The heap

- Used for working with varying amounts of data → Dynamic memory management
- Manual allocation, deallocation of memory
- Access only through pointers
- Allows access to a lot more memory than stack
Outline

1 Memory layout

2 Pointers
   - What is a pointer?
   - What are they needed for?
   - Declaration
   - Initialization
   - Special pointer (types)
   - Using pointers

3 Dynamic memory management

4 Literature
What is a pointer?

Source code:

```c
int i = 42;
int *ptr = &i;
```

Contents of memory:

- `42`
- `0x0000100C`
- `0x00001008`

Address:

- `0x0000100C`
- `0x00001008`
What are pointers needed for?

- data structures
  - Linked Lists
  - Trees
- Dynamic memory management

Normally in C: call-by-value - called function works on copies of its parameters

```c
void swap(int a, int b) {
    int c = a;
    a = b;
    b = c;
}
```

```c
int main() {
    int a = 42, b = 21;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);
}
```

⇒ call-by-reference - Use pointers (references) as parameters to make swap work!
It is important to understand the syntax of declaring pointers in C. A pointer declaration is defined as follows:

\[
type * \ [cv\text{-}qualifier] \ name \ [= \ expression];
\]

- `cv\text{-}qualifier` refers to type-qualifiers directly related to the pointer type (e.g., `const`).
- `type` can itself be a pointer type.
- `expression` can be `NULL`, address-of variable, ...
Initialization

- expression can be any expression that yields a value of type `type *` or more general type `&`
- `&` is called *address-of* operator
  Given a variable `a` of type `type`, `&a` yields the address of `a`, which is of type `type *`

```c
1 int a = 42;
2 // assign address-of a to b
3 const int * b = &a;
```
NULL

- **NULL** indicates that the pointer does not refer to a valid memory location
- can be assigned to any pointer, regardless of type
- Often used as return value to signal failure
void *

- typeless-pointer
- Implicit conversion between `void *` and any other pointer type (and the other way around)
- Commonly used in the standard library to offer generic functions

```c
void * memset (void * b,
               int c,
               size_t len);

int memcmp (void * s1,
            void * s2,
            size_t n);
```
Referencing & Dereferencing

- **Referencing**: Using the *address-of* operator (&) to assign the address of a variable to a pointer
- **Dereferencing**: Access the contents of memory where the pointer points to
  - Using *asterisk* operator *

```c
// call-by-reference
void swap(int * a, int * b) {
    int c = *a;
    *a = *b;
    *b = c;
}

int main() {
    int a = 42, b = 21;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
}
```
Comparing pointers

- Comparing for equality, inequality using `==` and `!=`
- Operators `>=`, `>`, `<`, `<=` also defined (see next section)
## Pointer arithmetic

```c
1 int arr[3] = {1, 2, 3};
2 int * ptr = &arr[0];
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value in memory</td>
<td>0x1004</td>
</tr>
<tr>
<td>Address</td>
<td>0x0008</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*ptr</td>
<td>arr[0]</td>
<td>*(ptr + 1)</td>
<td>*(ptr + 2)</td>
</tr>
</tbody>
</table>

- **Address:** 0x0008
- **Value in memory:** 0x1004, 1, 2, 3
- **Address:** 0x1004, 0x1008, 0x100c
**Pointer arithmetic**

- \( *\text{ptr} \equiv \text{arr}[0] \)
- \( (\text{ptr} + n) \equiv &\text{arr}[n] \)
  \[ \Rightarrow *(\text{ptr} + n) \equiv \text{arr}[n] \]

If \( \text{ptr} \) points to the \( i \)-th element of an array, \( (\text{ptr} + n) \) points to the \( (i + n) \)-th element of that array.

- \( (\text{ptr}_1 \text{ op } \text{ptr}_2) \) true, iff
  - \( \text{op} \equiv < \), \( \text{ptr}_1 \) points to element with smaller index than \( \text{ptr}_2 \)
  - \( \text{op} \equiv > \), \( \text{ptr}_1 \) points to element with larger index than \( \text{ptr}_2 \)
  - \( \ldots \)
Outline

1. Memory layout

2. Pointers

3. Dynamic memory management
   - When & Why?
   - Memory allocation
   - Resizing memory
   - Deallocating memory
   - Pitfalls

4. Literature
When & Why?

- Dynamic memory management used in functions
  - results should persist after function exits
  - allocate very large blocks of temporary memory

- Adapt to changing needs (the same program can e.g. sort data no matter the size)

- Dynamic data structures need dynamic memory management for
  - Growing
  - Shrinking
malloc()

- Declaration:

  ```c
  void * malloc(size_t size);
  ```

- `malloc()` reserves memory block with at least `size` bytes
  ⇒ returns `NULL` if not enough memory available

- Use `sizeof(type)` to find out size of type in bytes

- `malloc()` does not initialize the memory for you!
**calloc()**

- **Declaration:**

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

- `calloc()` allocates enough memory to hold `count` elements, each occupying `size` bytes in memory.
  ⇒ returns NULL if not enough memory available

- Every byte is set to 0.
realloc()

- Declaration

```
1| void * realloc(void * ptr, size_t size);
```

- `ptr` is a pointer previously returned by `malloc()`, `calloc()` or `realloc()`
- `size` is the new size (in bytes)

`realloc()` tries to change size of `ptr` and returns a new pointer to memory with the requested `size`. 
free()

```c
1| void free(void * ptr);
```

- **ptr** has to be a value previously returned by `malloc()`, `calloc()` or `realloc()`.
- size is part of internal records, so you don't need to specify that.
- General cycle:
  ```
  malloc() → Using memory → free()
  ```
Pitfalls / Problems

- **Check return values**
  ⇒ Dereferencing NULL will (most likely) crash your program!

- **Use-after-free**: Never access a memory block you already `free()`’d.

- **Memory leaks**: Don’t lose track of references to valid memory. You won’t be able to `free()` it if you do so.

- **Buffer overrun / underrun**: No built-in bounds checking in C!

- **Operator precedence**: `(*ptr)++ ≠ *(ptr++)`
Outline

1. Memory layout
2. Pointers
3. Dynamic memory management
4. Literature
Literature


- Prinz, Peter and Tony Crawford: C In a nutshell, 2006.