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

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

informatik die zukunft

Agenda

Outline

Overview

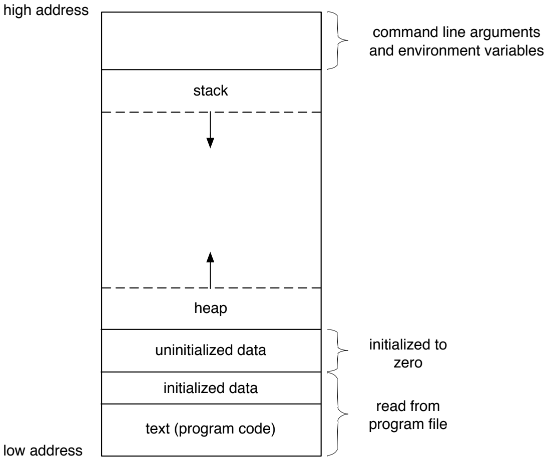


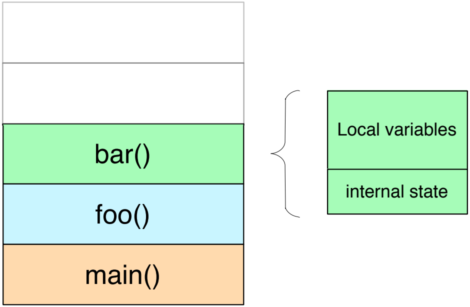
Figure : `http://infohost.nmt.edu/~eweiss/222_book/222_book/0201433079/ch07lev1sec6.html`

The stack

- Used for **local variables** in C
- Lightweight LIFO data structure
  $\Rightarrow$ Very fast (de-)allocation
- Automatic (de-)allocation of variables
  $\Rightarrow$ Out of scope, out of reach
- (Severely) space constrained

High level stack layout

```c
void bar() {
     int j = 42;
}

void foo() {
     bar();
}

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

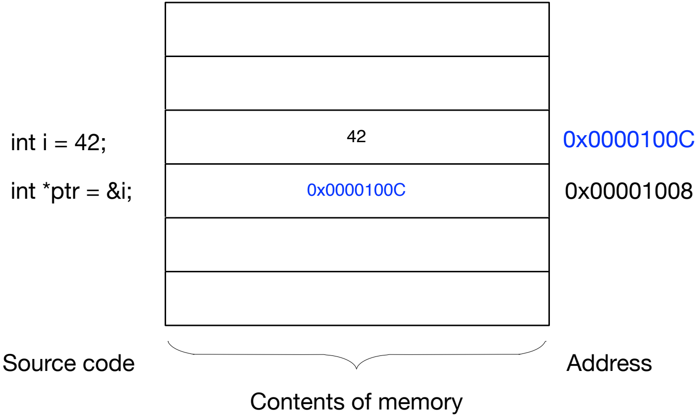


The heap

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

Outline

What is a pointer?



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;
}

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!

Declaration

```
type * [cv-qualifier] name [= expression];
```

- **cv-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 *`

```
int a = 42;
// assign address-of a to b
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

```
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 *

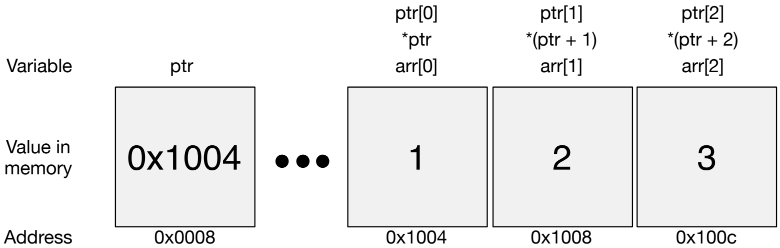
```
// call-by-reference
void swap(int * a, int * b) {
    int c = *a;
    *a = *b;
    *b = c;
}

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



Pointer arithmetic

- `*ptr` $\equiv$ `arr[0]`
- `(ptr + n)` $\equiv$ `&arr[n]`
  $\Rightarrow$ `*(ptr + n)` $\equiv$ `arr[n]`

If `ptr` points to the i-th element of an array, `(ptr + n)` points to the $(i + n)$-th element of that array.

- (`ptr1` op `ptr2`) true, iff
  - `op` $\equiv$ `<`, `ptr1` points to element with smaller index than `ptr2`
  - `op` $\equiv$ `>`, `ptr1` points to element with larger index than `ptr2`
  - $\cdots$

Outline

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:

```
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

```c
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()

```
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 loose 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)++` $\not\equiv$ `*(ptr++)`

Outline

Literature

- Duarte, Gustavo: Anatomy of a Program in Memory, 2009, URL: `http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/` (visited on Apr. 29, 2014).
- Kerninghan, Brian W. and Dennis M. Ritchie: The C Programming Language, 1988.
- Prinz, Peter and Tony Crawford: C In a nutshell, 2006.
- Memory Layout of a C Program, URL: `http://infohost.nmt.edu/~eweiss/222_book/222_book/0201433079/ch07lev1sec6.html` (excerpt from Stevens, Richard and Stephen Rago: Advanced Programming In The UNIX Environment, Second Edition)