Systems 3

Options and Dynamic Memory

Marcel Waldvogel

(Handout)

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

These slides are based on previous lectures held by Alexander Holupirek, Roman Byshko, and especially Stefan Klinger.

Chapter Goals

- How do processes get their information?
- What is the difference between arguments and environment variables?
- Why is one mechanism alone not sufficient?

- Why are processes not happy with static memory allocations?
- How do processes deal with dynamic memory requests?
- How is dynamic memory managed?
- Why are C's pointers and lack of checks necessary to implement dynamic memory management? How is this achieved?

- How to handle strings well?
- How to avoid (security) bugs when doing so?

Arguments and Environment

Command-line arguments

- The function called at program startup is named `main`.
- It shall be defined with a **return** type of `int`, and either zero, or two parameters:

```
int main(void);
int main(int argc, char *argv[]);
```

Terminology (although other names may be used)

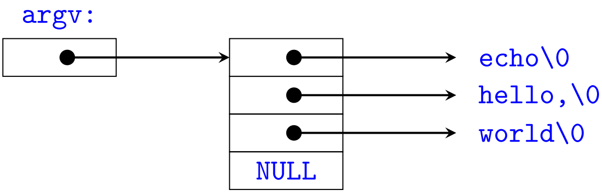
- `argc` stands for *argument count*
- `argv` stands for *argument vector*

If `argc` and `argv` are declared:

- The value of `argc` shall be **nonnegative**
- `argv[0]` represents the **program name** or `argv[0][0]` shall be the null character if the program name is not available.
- `argv[1]` to `argv[argc-1]` represent *program parameters*.
- `argv[argc]` shall be `NULL`, *i.e.*, it may be accessed.

- When a program is executed, the process that starts the new program can pass command-line arguments to it.
- That is the normal operation for UNIX system shells.

```
$ echo hello, world
hello, world
```



Echo command-line arguments

```
#include <stdio.h>

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

      return 0;
   }
```

```
$ ./a.out dsf ' dfsdf\'' ' t
argv[0]: "./a.out"
argv[1]: "dsf"
argv[2]: " dfsdf\ "
argv[3]: "t"
```

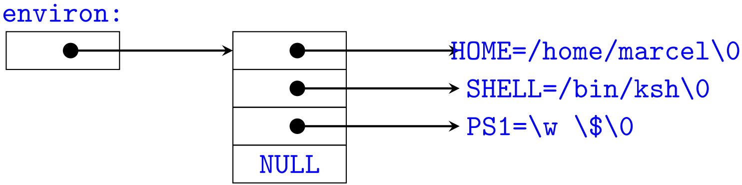
Environment variables

- Each program is also passed an **environment list**.
- Like the argument list, it is an array of character pointers.
- Each pointing to a null-terminated C string, of the form

  `name=value`

- The address of the array is contained in a global variable, the **environment pointer**:

```
extern char **environ;
```



History There once was an optional third argument to `main`, containing the environment.

Print the environment

```c
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

int main(void)
  {
    for (char **env = environ; *env; ++env)
      printf("%s\n", *env);

    char *p = getenv("PATH");  /* see getenv(3) */
    if (p)
      printf("Current path is: %s\n", p);

    return 0;
  }
```

Dynamic memory management

Current situation Until now, we cannot change the amount of space available to store data:

- The **number of variables** in a C program is fixed in the source code.
- **Arrays** cannot grow, nor shrink.

$\Rightarrow$ Use **excessively large** arrays that are guaranteed to be big enough.

That's not nice!

Dynamic memory Get more memory **on demand**, and only if required.

- First figure out how much memory is needed, then request that from the OS (*aka.* **allocating**).
- Or guess how much is needed and allocate that. Adapt as necessary.
- **Return** unused memory to the OS.

Allocating memory

malloc(3) and calloc(3) **allocate** blocks of memory.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t num, size_t size);
```

- size_t, defined in stddef.h is an unsigned integral type.
- malloc allocates a block of size bytes of memory.
  - The memory is **not initialised**.
  - Initialisation can be done using memset(3).
- calloc allocates memory for an array of num elements of size bytes each.
  - The storage is **initialised to zero** (good practice).
- Both fuctions return a pointer to the (start of) the allocated memory, or NULL if the request cannot be satisfied (or the requested size is 0).
- void * is the proper type for a **generic pointer**. No casting needed.

```
int *ip;
ip = calloc(42, sizeof(int)); /* space for 42 ints */
```

Extend or reduce allocated memory

realloc(3) "modifies" the size of a block of memory previously allocated with malloc(3).

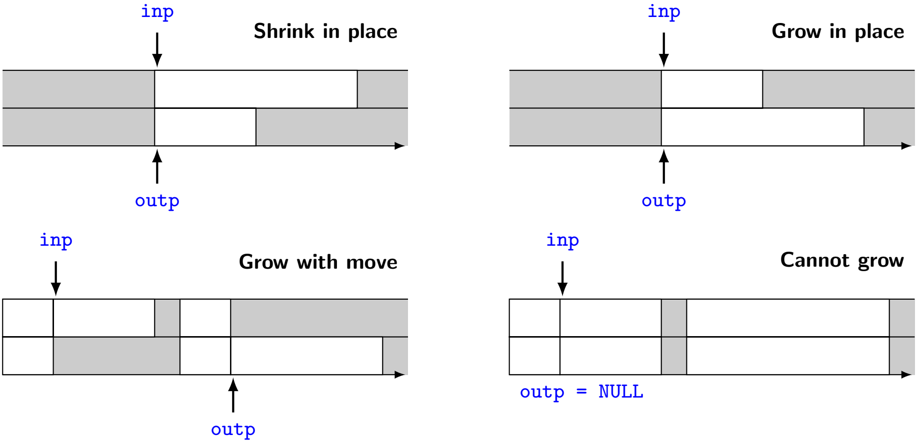
```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

- Changes the size of the object pointed to by ptr to size bytes.
  - Note that it may be necessary to **move** (i.e. copy) all data to a new location!
- realloc returns a **new pointer** to the (possibly moved) object.
  - Do not use the **old pointer**, it is invalid!
- The contents will be **unchanged** in the range from the start of the region up to the minimum of the old and new sizes.
  - Freshly allocated memory is **not initialised**.
- **Note:** ptr must point to memory previously allocated with malloc, *i.e.*, this will not work:

```
int arr[23];
int *p = arr;
p = realloc(p, 42 * sizeof(int));  /* wrong */
```

realloc() scenarios

```
void *inp, *outp;
outp = realloc(inp, newSize);
```



Allocated chunks are white, free chunks are grey. Addresses increase to the right.

Freeing allocated memory

free(3) frees memory previously allocated with malloc(3).

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

- If ptr is a NULL pointer, no action occurs.
- It is an error to dereference something **after it has been freed**.
- It is important to free memory you do not need anymore.
  - In general, this is not an easy task.
  - There is **no garbage collector**.
  - If you do not free, you may **run out of memory**.
- **Note:** ptr must point to memory previously allocated with malloc etc, *i.e.*, this will not work:

```
int arr[23];
free(arr);  /* wrong */
```

Example

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int *p = malloc(8 * sizeof(int));  /* allocate mem for 8 int */

   for (int i = 0; i < 8; i++)                     /* write some data */
   p[i] = i*i;

   p = realloc(p, 16 * sizeof(int));  /* get more space */
   p[15] = 100;

   p = realloc(p, 12 * sizeof(int));  /* free some memory */
   /* p[15] = 7; */       /* invalid */

   for (int i = 0; i < 12; i++)                    /* print whole memory block */
      printf("%2d\t%d\n", i, p[i]);              /* slots 8–11 contain garbage */

   free(p);  /* free all memory used by p */

   return 0;
}
```

Caution

- **Always free allocated memory** when it's no longer used.
- It is a bug not to **check the return values** of `malloc(3)`, `calloc(3)`, or `realloc(3)` for error conditions.
  Review the example on slide 15!
- One **must not access unallocated memory**, or memory after calling free on it.

Ignoring any of these rules **is a bug** that may, or may not, show up during testing. Even if the program behaves as expected, it is still buggy!

Handling strings

- With `#include <string.h>` you'll get access to a plethora of string handling functions, documented in `string`(3).
- Example: Copy string pointed to by src, to buffer pointed to by dest.

```
char *strcpy(char *dest, const char *src);                /* cf. strcpy(3) */
```

Question How can we make a copy of a string?

```
const char *msg = "hello world\n";

char *copy;
strcpy(copy, msg);
```

- What do you think about this approach?

```c
const char *msg = "hello world\n";

char *copy;  /* not initialized, points nowhere */
strcpy(copy, msg);
```

Bad idea: The target pointer does not point to any allocated memory!

$\Rightarrow$ **Undefined behavior**[1]

[1] https://blog.regehr.org/archives/213, https://blog.regehr.org/archives/970

Question String copy: What about this one?

```
char *strcpy(char *dest, const char *src);
const char *msg = "hello world\n";

char *copy = malloc(strlen(msg));
strcpy(copy, msg);
```

```
char *strcpy(char *dest, const char *src);
const char *msg = "hello world\n";

char *copy = malloc(strlen(msg)); /* not enough */
/* return value unchecked */
strcpy(copy, msg);
```

- Unchecked if we got any memory at all.
- Even then, not enough memory is allocated: `strlen` returns length *excluding* `NUL`, but `strcpy` copies that as well!

⇒ **Undefined behavior**

Easy to fix:

```
#include <err.h>

char *copy = malloc(strlen(msg) + 1);
if (!copy)
    err(1, "copy");    /* cf. err(3). Terminates with a message like */
/* a.out: copy: Cannot allocate memory */
```

Question String copy: Not correct. Why?

```
const char *msg = "Old MacDonald Had a Farm";

size_t len = strlen(msg) + 1;
char *cp1 = malloc(len),
*cp2 = malloc(len);

if (!cp1 || !cp2)
   err(1, "cp1 or cp2");

for (size_t i = 0; i < 13; i++)  /* copy only first two words */
   cp1[i] = msg[i];

strcpy(cp2, cp1);  /* copy that to cp2 */
```

```c
strcpy(cp2, cp1);  /* copy cp1 to cp2 */
```

- `strcpy` will copy bytes from `cp1` until the string ends, *i.e.*, until it sees a `'\0'` character.
- The source `cp1` may not be terminated by a `NUL` character!

⇒ `strcpy` may "fall over the edge", and overwrite adjacent memory!
⇒ **Undefined behavior**

Solution to all these cases:

- Use `strncpy(3)` instead, which will not write more than `n` bytes!

```c
char *strncpy(char *dest, const char *src, size_t n);
```

Always be aware of the amount of data to be written!

- *Overflowing fixed-length string buffers leads to security vulnerabilites* `strcpy`(3)

Note that `strncpy` may **not write** the terminating `NUL`!