Hacking in C

Memory layout

Radboud University, Nijmegen, The Netherlands

Spring 2018

A short recap

- The `&` operator gives us the address of data
- Inverse of `&` is the `*` operator (dereferencing)

A short recap

- The `&` operator gives us the address of data
- Inverse of `&` is the `*` operator (dereferencing)
- *Aligning* data to word (or larger) limits makes access more efficient
- Compilers may introduce padding to align data

A short recap

- The & operator gives us the address of data
- Inverse of & is the * operator (dereferencing)
- *Aligning* data to word (or larger) limits makes access more efficient
- Compilers may introduce padding to align data
- Arrays are passed by reference (decay to pointer to the first element)

A short recap

- The `&` operator gives us the address of data
- Inverse of `&` is the `*` operator (dereferencing)
- *Aligning* data to word (or larger) limits makes access more efficient
- Compilers may introduce padding to align data
- Arrays are passed by reference (decay to pointer to the first element)
- Can do "pointer arithmetic", i.e., increase and decrease pointers
- `x++` for `type *x` increases *address* by `sizeof(type)`

A short recap

- The & operator gives us the address of data
- Inverse of & is the * operator (dereferencing)
- *Aligning* data to word (or larger) limits makes access more efficient
- Compilers may introduce padding to align data
- Arrays are passed by reference (decay to pointer to the first element)
- Can do "pointer arithmetic", i.e., increase and decrease pointers
- `x++` for `type *x` increases *address* by `sizeof(type)`
- Strings are null-terminated arrays of bytes
- Array access can be expressed as pointers: `a[i]` is the same as `*(a+i)`

A short recap

- The & operator gives us the address of data
- Inverse of & is the * operator (dereferencing)
- *Aligning* data to word (or larger) limits makes access more efficient
- Compilers may introduce padding to align data
- Arrays are passed by reference (decay to pointer to the first element)
- Can do "pointer arithmetic", i.e., increase and decrease pointers
- `x++` for `type *x` increases *address* by `sizeof(type)`
- Strings are null-terminated arrays of bytes
- Array access can be expressed as pointers: `a[i]` is the same as `*(a+i)`
- ... is the same as `i[a]`! (try it out ;-))

A short recap

- The & operator gives us the address of data
- Inverse of & is the * operator (dereferencing)
- *Aligning* data to word (or larger) limits makes access more efficient
- Compilers may introduce padding to align data
- Arrays are passed by reference (decay to pointer to the first element)
- Can do "pointer arithmetic", i.e., increase and decrease pointers
- `x++` for `type *x` increases *address* by `sizeof(type)`
- Strings are null-terminated arrays of bytes
- Array access can be expressed as pointers: `a[i]` is the same as `*(a+i)`
- ... is the same as `i[a]`! (try it out ;-))
- Can use pointers ot inspect raw memory content

A short recap

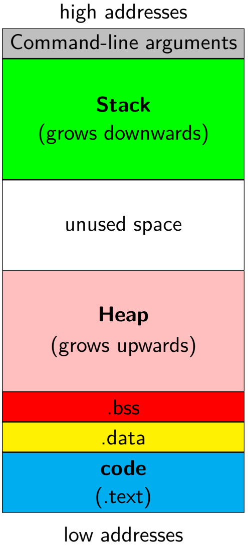
- The & operator gives us the address of data
- Inverse of & is the * operator (dereferencing)
- *Aligning* data to word (or larger) limits makes access more efficient
- Compilers may introduce padding to align data
- Arrays are passed by reference (decay to pointer to the first element)
- Can do "pointer arithmetic", i.e., increase and decrease pointers
- `x++` for `type *x` increases *address* by `sizeof(type)`
- Strings are null-terminated arrays of bytes
- Array access can be expressed as pointers: `a[i]` is the same as `*(a+i)`
- ... is the same as `i[a]`! (try it out ;-))
- Can use pointers ot inspect raw memory content

This lecture: look at the systematics of what is stored where

Memory segments

The OS allocates memory for data and code of each running process

- stack: for local variables (including command-line arguments)
- heap: For *dynamic* memory
- data segment:
  - global and static uninitialized variables (.bss)
  - global and static initialized variables (.data)
- code segment: code (and possibly constants)



/proc/<pid>/maps, ps, and size

- Find information about memory allocation for process with PID <pid> in

  /proc/<pid>/maps

- For example:

```
008e6000-00b11000         rw-p 00000000 00:00 0   [heap]
7ffd739cb000-7ffd739ec000 rw-p 00000000 00:00 0   [stack]
```

- Also information about dynamic libraries used by process

/proc/<pid>/maps, ps, and size

- Find information about memory allocation for process with PID `<pid>` in

  `/proc/<pid>/maps`

- For example:

```
008e6000-00b11000         rw-p 00000000 00:00 0   [heap]
7ffd739cb000-7ffd739ec000 rw-p 00000000 00:00 0   [stack]
```

- Also information about dynamic libraries used by process
- List all processes with PID: `ps`

`/proc/<pid>/maps`, `ps`, and `size`

- Find information about memory allocation for process with PID `<pid>` in

  `/proc/<pid>/maps`

- For example:

```
008e6000-00b11000         rw-p 00000000 00:00 0   [heap]
7ffd739cb000-7ffd739ec000 rw-p 00000000 00:00 0   [stack]
```

- Also information about dynamic libraries used by process
- List all processes with PID: `ps`
- Find information about memory segment sizes using `size`
- Use `size` on binary (`.o` file or executable)
- For more verbatim output can use `size -A`

Virtual memory

- Central idea:
  - Don't let processes use addresses in physical memory
  - Instead, use *virtual addresses*
  - For each access to a virtual address, map to actual physical address

Virtual memory

- Central idea:
  - Don't let processes use addresses in physical memory
  - Instead, use *virtual addresses*
  - For each access to a virtual address, map to actual physical address
- Obviously, don't want to map byte-by-byte
- Chop the memory into *pages* of fixed size (typically 4KB)

Virtual memory

- Central idea:
  - Don't let processes use addresses in physical memory
  - Instead, use *virtual addresses*
  - For each access to a virtual address, map to actual physical address
- Obviously, don't want to map byte-by-byte
- Chop the memory into *pages* of fixed size (typically 4KB)
- Use a *page table* to establish the mapping

Virtual memory

- Central idea:
  - Don't let processes use addresses in physical memory
  - Instead, use *virtual addresses*
  - For each access to a virtual address, map to actual physical address
- Obviously, don't want to map byte-by-byte
- Chop the memory into *pages* of fixed size (typically 4KB)
- Use a *page table* to establish the mapping
- Essentially, use a different page table for each process
- If there is no entry for a virtual address in a processes' page table: exit with segmentation fault

Advantages of virtual memory

- Processes can use (seemingly) contiguous memory locations
- Those addresses don't have to be contiguous in *physical* memory

Advantages of virtual memory

- Processes can use (seemingly) contiguous memory locations
- Those addresses don't have to be contiguous in *physical* memory
- Can even assign more memory than is physically available
- Need to swap memory content to and from hard drive

Advantages of virtual memory

- Processes can use (seemingly) contiguous memory locations
- Those addresses don't have to be contiguous in *physical* memory
- Can even assign more memory than is physically available
- Need to swap memory content to and from hard drive
- Can **separate address spaces** of different programs!
- OS can now ensure that one process cannot read/write another processes' memory

Bare-metal “memory management”

- C is also used to program small embedded microcontrollers
- Sometimes run code *bare metal*, i.e., without OS
- No virtual memory, no segfaults

Bare-metal "memory management"

- C is also used to program small embedded microcontrollers
- Sometimes run code *bare metal*, i.e., without OS
- No virtual memory, no segfaults
- Stack can happily grow into heap or data segment
- Typically rather little RAM, so this happens easily
- Nasty to debug behavior

Global variables

- Global variables are declared outside of all functions
- Example:

```
#include <stdio.h>
long n = 12345678;
char *s = "hello world!\n";
int a[256];
...
```

- The initialized variables `n` and `s` will be in .data
- The uninialized variable `a` will be in .bss

Global variables

- Global variables are declared outside of all functions
- Example:

```
#include <stdio.h>
long n = 12345678;
char *s = "hello world!\n";
int a[256];
...
```

- The initialized variables `n` and `s` will be in .data
- The uninialized variable `a` will be in .bss
- The .bss section is typically initialized to zero
- An OS can do this “on-demand”, i.e., when reading a variable for the first time

Global variables

- Global variables are declared outside of all functions
- Example:

```
#include <stdio.h>
long n = 12345678;
char *s = "hello world!\n";
int a[256];
...
```

- The initialized variables `n` and `s` will be in .data
- The uninialized variable `a` will be in .bss
- The .bss section is typically initialized to zero
- An OS can do this “on-demand”, i.e., when reading a variable for the first time
- Some platforms have a special non-initialized .bss subsection
- Example: AVR microcontrollers with a .noinit section

Static variables

- A static variable is local, but keeps its value across calls
- Example:

```
void f()
{
  static int x = 0;
  printf("%d\n", x++);
}
```

- If `x` was not declared static, this function would always print 0

Static variables

- A static variable is local, but keeps its value across calls
- Example:

```
void f()
{
  static int x = 0;
  printf("%d\n", x++);
}
```

- If `x` was not declared static, this function would always print 0

Static variables

- A static variable is local, but keeps its value across calls
- Example:

```
void f()
{
  static int x = 0;
  printf("%d\n", x++);
}
```

- If `x` was not declared static, this function would always print 0
- Different for static `x`; output increases by one for every call

Static variables

- A static variable is local, but keeps its value across calls
- Example:

```
void f()
{
  static int x = 0;
  printf("%d\n", x++);
}
```

- If `x` was not declared static, this function would always print 0
- Different for static `x`; output increases by one for every call
- Would get the same behavior if `x` was global
- ... but a global `x` could be modified also by other functions

The stack – a simple datastructure

- A stack is essentially a LIFO queue; two operations
  - `PUSH(x)`
  - `x = POP()`

The stack – a simple datastructure

- A stack is essentially a LIFO queue; two operations
  - `PUSH(x)`
  - `x = POP()`
- The memory stack, very much simplified:
  - Function calls push local data on the stack
  - Returns from functions pop that data again

The stack – a simple datastructure

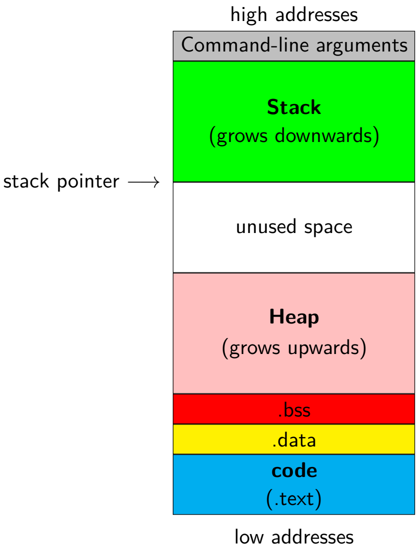
- A stack is essentially a LIFO queue; two operations
  - `PUSH(x)`
  - `x = POP()`
- The memory stack, very much simplified:
  - Function calls push local data on the stack
  - Returns from functions pop that data again
- Often also possible: access data relative to the top
- Required for all these operations: pointer to the top

The stack – a simple datastructure

- A stack is essentially a LIFO queue; two operations
  - `PUSH(x)`
  - `x = POP()`
- The memory stack, very much simplified:
  - Function calls push local data on the stack
  - Returns from functions pop that data again
- Often also possible: access data relative to the top
- Required for all these operations: pointer to the top
- Pointer can be
  - "hidden" (only modified by `PUSH` or `POP`)
  - "exposed" (allowing relative data access)
- On AVR: extra instructions to expose the stack pointer

Stack frames and the stack pointer

- Stack consists of *stack frames*
- Each function on the current *call stack* has its own frame
- Active frame is on top of the stack
- "Top of the stack": at low addresses
- Stack pointer points to end (low address) of active frame
- Stack pointer is typically in special register (`rsp` on AMD64)

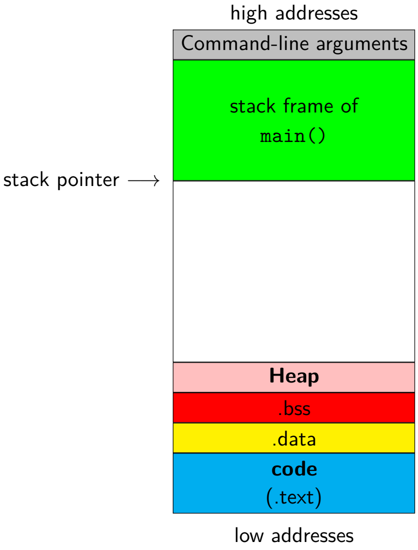


Stack frames and the stack pointer

Example:

```
int func(int a, int b)
{
  ...
  return 10001;
}

int main(void)
{
  ...
  int x = func(42, 23)
  ...
}
```

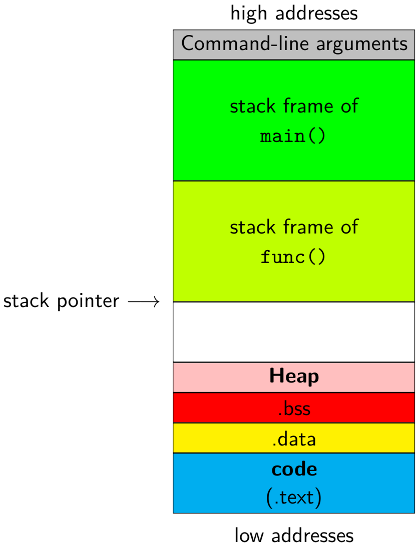


Stack frames and the stack pointer

Example:

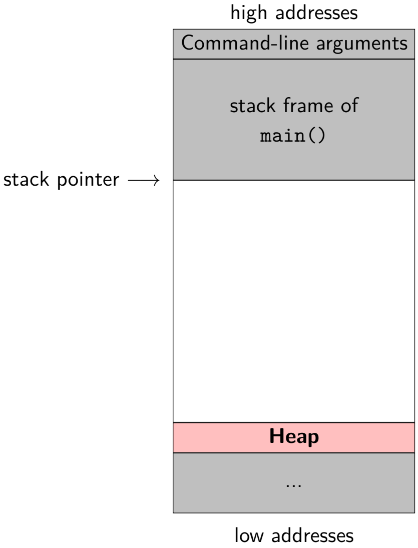
```
int func(int a, int b)
{
  ...
  return 10001;
}

int main(void)
{
  ...
  int x = func(42, 23)
  ...
}
```



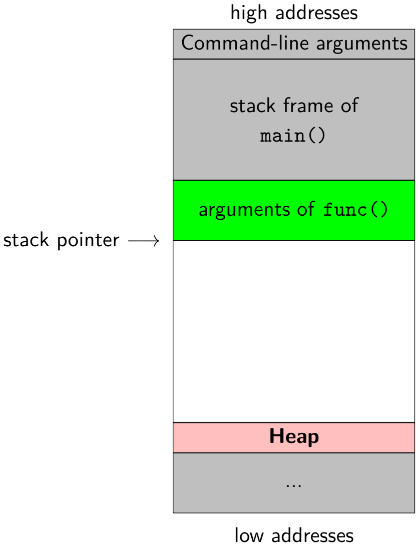
A zoom into the stack frame

- Stack before the function call



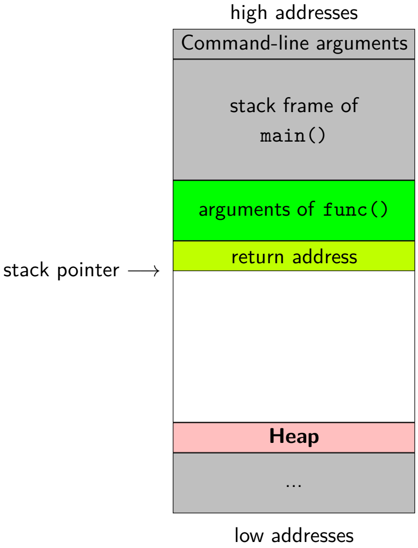
A zoom into the stack frame

- Stack before the function call
- Caller (`main`) first puts arguments for `func` on the stack



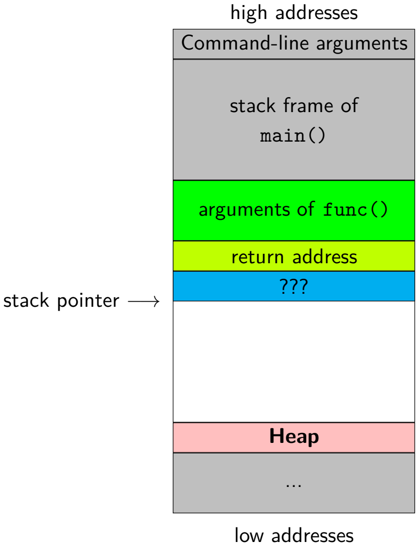
A zoom into the stack frame

- Stack before the function call
- Caller (`main`) first puts arguments for `func` on the stack
- Caller pushes the **return address** onto the stack



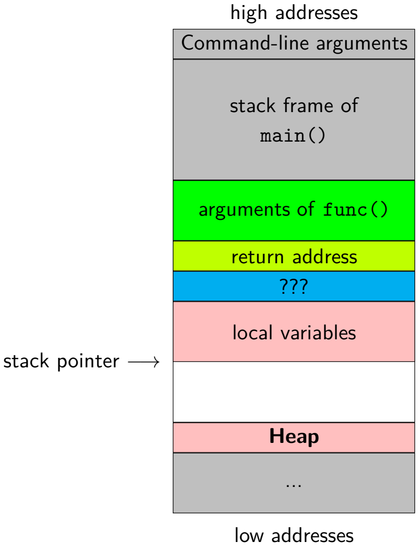
A zoom into the stack frame

- Stack before the function call
- Caller (`main`) first puts arguments for `func` on the stack
- Caller pushes the **return address** onto the stack
- ???



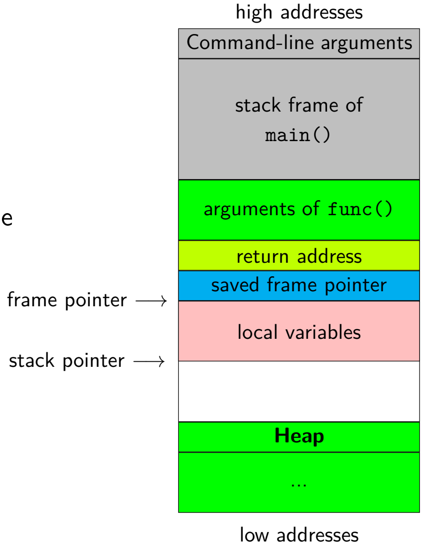
A zoom into the stack frame

- Stack before the function call
- Caller (`main`) first puts arguments for `func` on the stack
- Caller pushes the **return address** onto the stack
- ???
- Callee pushes local variables onto the stack



The frame pointer

- So what's with the ???...?
- Traditionally also have an *frame pointer*
- Pointing to the end (high address) of the active stack frame
- On x86 in `ebp` register (AMD64: `rbp`)
- Function call also saves previous frame pointer on the stack
- On AMD64 commonly omitted:
  - Faster function calls
  - One additional register available



Size of the stack

- C does not limit the size of the stack *in the language*
- In practice, of course stack space is limited
- In bare-metal environments, limited by hardware
- Otherwise limited by OS

Size of the stack

- C does not limit the size of the stack *in the language*
- In practice, of course stack space is limited
- In bare-metal environments, limited by hardware
- Otherwise limited by OS
- Under Linux, use `ulimit -s` to see stack size (in KB)
- Inside a C program, can use `getrlimit`
- Can also use `setrlimit` to request larger (or smaller) stack

Things that may go wrong on the stack

- Obviously, we may exhaust stack space
- Simple example: infinite recursion (`exhauststack.c`)
- This is known as **stack overflow**
- In safety critical environments need to avoid this!
- Generally, don't put "big data" on the stack

Things that may go wrong on the stack

- Obviously, we may exhaust stack space
- Simple example: infinite recursion (`exhauststack.c`)
- This is known as **stack overflow**
- In safety critical environments need to avoid this!
- Generally, don't put "big data" on the stack
- Variables on the stack are not auto-initialized
- Reading uninitalized local variables allows to read local data from previous functions

Things that may go wrong on the stack

- Obviously, we may exhaust stack space
- Simple example: infinite recursion (`exhauststack.c`)
- This is known as **stack overflow**
- In safety critical environments need to avoid this!
- Generally, don't put "big data" on the stack
- Variables on the stack are not auto-initialized
- Reading uninitalized local variables allows to read local data from previous functions
- The stack mixes program and control data
- Writing beyond buffers may overwrite return addresses
- Main attack vector for "targeted undefined behavior"

. . . how bad is “wrong” exactly?

. . . how bad is “wrong” exactly?

“On Thursday October 24, 2013, an Oklahoma court ruled against Toyota in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module’s (ECM) firmware.”

What went wrong?

- Critical variables were not mirrored (stored twice)
- Most importantly, result value `TargetThrottleAngle` wasn't mirrored
- Also critical data structes of the real-time OS weren't mirrored

What went wrong?

- Critical variables were not mirrored (stored twice)
- Most importantly, result value `TargetThrottleAngle` wasn't mirrored
- Also critical data structes of the real-time OS weren't mirrored
- Stack overflow
  - Toyota claimed stack upper bound of 41% of total memory
  - Stack was actually using 94% of total memory
  - Analysis ignored stack used by some 350 assembly functions

What went wrong?

- Critical variables were not mirrored (stored twice)
- Most importantly, result value `TargetThrottleAngle` wasn't mirrored
- Also critical data structes of the real-time OS weren't mirrored
- Stack overflow
  - Toyota claimed stack upper bound of 41% of total memory
  - Stack was actually using 94% of total memory
  - Analysis ignored stack used by some 350 assembly functions
- Code used recursion (forbidden by MISRA-C guidelines)
- MISRA-C: guidelines by the Motor Industry Software Reliability Association

What went wrong?

- Critical variables were not mirrored (stored twice)
- Most importantly, result value `TargetThrottleAngle` wasn't mirrored
- Also critical data structes of the real-time OS weren't mirrored
- Stack overflow
  - Toyota claimed stack upper bound of 41% of total memory
  - Stack was actually using 94% of total memory
  - Analysis ignored stack used by some 350 assembly functions
- Code used recursion (forbidden by MISRA-C guidelines)
- MISRA-C: guidelines by the Motor Industry Software Reliability Association

"A litany of other faults were found in the code, including buffer overflow, unsafe casting, and race conditions between tasks."

Hardware specifics

- Stack layout shown so far is typical
- Many details look different on different architectures:
  - Memory-segment layout may be different
  - (Some) function arguments may be passed through registers
  - Return values often passed through registers (sometimes also over the stack)
  - Frame pointer may be omitted

Hardware specifics

- Stack layout shown so far is typical
- Many details look different on different architectures:
  - Memory-segment layout may be different
  - (Some) function arguments may be passed through registers
  - Return values often passed through registers (sometimes also over the stack)
  - Frame pointer may be omitted
- Example: AMD64
  - Integer and pointer arguments are passed through `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
  - Return value in `rax`
  - ... at least for Linux, Windows is subtly different

Limitations of the stack

```
int * table_of(int num, int len) {
  int table[len];
  for ( int i =0; i <= len ; i ++) {
    table[ i ] = i *num;
  }
  return table; /* an int [] can be used as an int * */
}
```

What happens if we call this function as follows?:

```
int *table3 = table_of(3,10);
printf("5 times 3 is %d \n", table3[5]);
```

Limitations of the stack

```
int * table_of(int num, int len) {
  int table[len];
  for ( int i =0; i <= len ; i ++) {
    table[ i ] = i *num;
  }
  return table; /* an int [] can be used as an int * */
}
```

What happens if we call this function as follows?:

```
int *table3 = table_of(3,10);
printf("5 times 3 is %d \n", table3[5]);
```

- The stack cannot preserve data beyond `return` of a function.
- Except of course of returned *data* (not pointers!)

Limitations of the stack

```
int * table_of(int num, int len) {
  int table[len];
  for ( int i =0; i <= len ; i ++) {
    table[ i ] = i *num;
  }
  return table; /* an int [] can be used as an int * */
}
```

What happens if we call this function as follows?:

```
int *table3 = table_of(3,10);
printf("5 times 3 is %d \n", table3[5]);
```

- The stack cannot preserve data beyond `return` of a function.
- Except of course of returned *data* (not pointers!)
- Obvious other limitation: size!

The heap

- Think about the heap as a large piece of scrap paper
- We can request (large) continuous space on the piece of paper
- Note that “continuous” is easily ensured by virtual memory

The heap

- Think about the heap as a large piece of scrap paper
- We can request (large) continuous space on the piece of paper
- Note that “continuous” is easily ensured by virtual memory
- This space is accessible through a pointer (what else ;-))
- Space remains valid across function calls
- Every function that “knows” a pointer to the space can use it

malloc

- Function to request space: void *malloc(size_t nbytes)
- Need to #include <stdlib.h> to use malloc
- size_t is an unsigned integer type

malloc

- Function to request space: `void *malloc(size_t nbytes)`
- Need to `#include <stdlib.h>` to use `malloc`
- `size_t` is an unsigned integer type
- Returns a `void` pointer to `nbytes` of memory
- Can also fail, in that case, it returns `NULL`

malloc

- Function to request space: `void *malloc(size_t nbytes)`
- Need to `#include <stdlib.h>` to use `malloc`
- `size_t` is an unsigned integer type
- Returns a `void` pointer to `nbytes` of memory
- Can also fail, in that case, it returns `NULL`
- Usually pointers in C are typed, `void *x` is an "untyped" pointer
- A `void *` implicitly casts to and from any other pointer type
- Remember that this is *not* the case in C++!

malloc

- Function to request space: `void *malloc(size_t nbytes)`
- Need to `#include <stdlib.h>` to use `malloc`
- `size_t` is an unsigned integer type
- Returns a `void` pointer to `nbytes` of memory
- Can also fail, in that case, it returns `NULL`
- Usually pointers in C are typed, `void *x` is an "untyped" pointer
- A `void *` implicitly casts to and from any other pointer type
- Remember that this is *not* the case in C++!
- Example of `malloc` usage:

```
int *x = malloc(10000 * sizeof(int));
```

- Request for space for $10\,000$ integers on the heap

NULL

- The value NULL is guaranteed to not point to a valid address
- The following code produces **undefined behavior**:

```
int *x = NULL;
int i = *x;
```

NULL

- The value NULL is guaranteed to not point to a valid address
- The following code produces **undefined behavior**:

```
int *x = NULL;
int i = *x;
```

- Important to note: NULL is not the same as 0

NULL

- The value NULL is guaranteed to not point to a valid address
- The following code produces **undefined behavior**:

```
int *x = NULL;
int i = *x;
```

- Important to note: NULL is not the same as 0
- In boolean expressions, NULL evaluates to false
- These two lines have the same semantics:

```
if(x == NULL) printf("NULL\n");
if(!x) printf("NULL\n");
```

NULL

- The value NULL is guaranteed to not point to a valid address
- The following code produces **undefined behavior**:

```
int *x = NULL;
int i = *x;
```

- Important to note: NULL is not the same as 0
- In boolean expressions, NULL evaluates to false
- These two lines have the same semantics:

```
if(x == NULL) printf("NULL\n");
if(!x) printf("NULL\n");
```

- Not true in all programming languages, e.g., not in C#

ALWAYS check for malloc failure!

- The following code is terribly unsafe:

```
int *table = malloc(TABLESIZE * sizeof(int));
for(size_t i=0;i<TABLESIZE;i++)
  table[i] = 42;
```

ALWAYS check for malloc failure!

- The following code is terribly unsafe:

```
int *table = malloc(TABLESIZE * sizeof(int));
for(size_t i=0;i<TABLESIZE;i++)
  table[i] = 42;
```

ALWAYS check for malloc failure!

- The following code is terribly unsafe:

```
int *table = malloc(TABLESIZE * sizeof(int));
for(size_t i=0;i<TABLESIZE;i++)
  table[i] = 42;
```

- `malloc` might return `NULL`
- `table[i]` dereferences the pointer `table`
- Consequence: **undefined behavior!**

ALWAYS check for malloc failure!

- The following code is terribly unsafe:

```
int *table = malloc(TABLESIZE * sizeof(int));
for(size_t i=0;i<TABLESIZE;i++)
  table[i] = 42;
```

- `malloc` might return `NULL`
- `table[i]` dereferences the pointer `table`
- Consequence: **undefined behavior!**
- Correct version:

```
int *table = malloc(TABLESIZE * sizeof(int));
if(table == NULL) exit(-1);
for(size_t i=0;i<TABLESIZE;i++)
  table[i] = 42;
```

ALWAYS check for malloc failure!

- The following code is terribly unsafe:

```
int *table = malloc(TABLESIZE * sizeof(int));
for(size_t i=0;i<TABLESIZE;i++)
  table[i] = 42;
```

- `malloc` might return `NULL`
- `table[i]` dereferences the pointer `table`
- Consequence: **undefined behavior!**
- Correct version:

```
int *table = malloc(TABLESIZE * sizeof(int));
if(table == NULL) exit(-1);
for(size_t i=0;i<TABLESIZE;i++)
  table[i] = 42;
```

- Could alternatively use boolean behavior of `NULL`:

```
if(!table) exit(-1);
```

`free`

- You, the programmer, are in charge of *releasing* memory!
- When you don't need some allocated memory anymore, use

```
free(x);
```

- Here, `x` is a pointer to previously `malloc`'ed memory

free

- You, the programmer, are in charge of *releasing* memory!
- When you don't need some allocated memory anymore, use

```
free(x);
```

- Here, x is a pointer to previously malloc'ed memory
- Typical usage patters:

```
int *x = malloc(NUMX * sizeof(int));
if(x == NULL) exit(-1);
...
free(x);
```

- The calls to malloc and free can be in different functions

free

- You, the programmer, are in charge of *releasing* memory!
- When you don't need some allocated memory anymore, use

```
free(x);
```

- Here, `x` is a pointer to previously `malloc`'ed memory
- Typical usage patters:

```
int *x = malloc(NUMX * sizeof(int));
if(x == NULL) exit(-1);
...
free(x);
```

- The calls to `malloc` and `free` can be in different functions
- Not freeing `malloc`'ed memory is known as a *memory leak*

`realloc`

- Sometimes want to *expand* or *shrink* `malloc`'ed space
- Do this by using

```
void *realloc(void *ptr, size_t new_size);
```

- Content in the allocated area is preserved
- New space is created (or cut away) "at the end"

`realloc`

- Sometimes want to *expand* or *shrink* `malloc`'ed space
- Do this by using

```
void *realloc(void *ptr, size_t new_size);
```

- Content in the allocated area is preserved
- New space is created (or cut away) "at the end"
- This call may also return `NULL`
- If return value is `NULL`, previously allocated memory is not freed!

`realloc`

- Sometimes want to *expand* or *shrink* `malloc`'ed space
- Do this by using

```
void *realloc(void *ptr, size_t new_size);
```

- Content in the allocated area is preserved
- New space is created (or cut away) “at the end”
- This call may also return `NULL`
- If return value is `NULL`, previously allocated memory is not freed!
- Usage pattern:

```
xnew = realloc(x, NEWSIZE);
if(xnew == NULL)
{
  free(x);
  exit(-1); // or continue with old size of x
}
else
{
  x = xnew;
}
```

Dangling pointers, double-free, . . .

- **Never** use a pointer after it has been freed, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
...
printf("Let's see what the value of x is now: %p\n", x);
```

- This is **undefined behaviour**

Dangling pointers, double-free, . . .

- **Never** use a pointer after it has been freed, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
...
printf("Let's see what the value of x is now: %p\n", x);
```

- This is **undefined behaviour**
- Also, never double-free a pointer, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
free(x);
```

Dangling pointers, double-free, . . .

- **Never** use a pointer after it has been freed, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
...
printf("Let's see what the value of x is now: %p\n", x);
```

- This is **undefined behaviour**
- Also, never double-free a pointer, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
free(x);
```

- Not always that obvious, you may have *pointer aliases*
- Pointer alias: multiple pointers to the same location

Dangling pointers, double-free, ...

- **Never** use a pointer after it has been freed, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
...
printf("Let's see what the value of x is now: %p\n", x);
```

- This is **undefined behaviour**
- Also, never double-free a pointer, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
free(x);
```

- Not always that obvious, you may have *pointer aliases*
- Pointer alias: multiple pointers to the same location
- Never "lose" the last pointer to a location
- This inevitable creates a memory leak: you *cannot* `free` anymore!

Stack vs. heap vs. data segment

Data segment

- Data in the data segment exists throughout the whole execution of the program
  - global variables accessible to every function
  - static local variables only accessible to the respective function

Stack vs. heap vs. data segment

Data segment

- Data in the data segment exists throughout the whole execution of the program
  - global variables accessible to every function
  - static local variables only accessible to the respective function

Stack

- Space on the stack *allocated automatically*
- Stack space automatically removed when returning from a function
- Certain risk of overflowing the stack

Stack vs. heap vs. data segment

Data segment

- Data in the data segment exists throughout the whole execution of the program
  - global variables accessible to every function
  - static local variables only accessible to the respective function

Stack

- Space on the stack *allocated automatically*
- Stack space automatically removed when returning from a function
- Certain risk of overflowing the stack

Heap

- Space on the heap needs to be *requested manually* (`malloc`)
- Request may be denied (`NULL` return) and this must be handled
- Space on the heap needs to be *freed manually* (`free`)
- Risk of memory leaks, double frees, etc.

What's wrong with this code (part 1)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  if(a == NULL) return -1;
  char *x = (char *)a;
  ...
  free(x);
  free(a);
}
```

What's wrong with this code (part 1)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  if(a == NULL) return -1;
  char *x = (char *)a;
  ...
  free(x);
  free(a);
}
```

What's wrong with this code (part 1)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  if(a == NULL) return -1;
  char *x = (char *)a;
  ...
  free(x);
  free(a);
}
```

- Fairly simple: double-free.

What's wrong with this code (part 2)?

```
int *f()
{
  int a[100];
  for(i=0;i<100;i++)
    a[i] = i;
  return a;
}
```

What's wrong with this code (part 2)?

```
int *f()
{
  int a[100];
  for(i=0;i<100;i++)
    a[i] = i;
  return a;
}
```

What's wrong with this code (part 2)?

```
int *f()
{
  int a[100];
  for(i=0;i<100;i++)
    a[i] = i;
  return a;
}
```

- Return type is `int *`, returning `a` is not a *type* problem
- Remember that an array can "decay" to a pointer to its first element

What's wrong with this code (part 2)?

```
int *f()
{
  int a[100];
  for(i=0;i<100;i++)
    a[i] = i;
  return a;
}
```

- Return type is `int *`, returning `a` is not a *type* problem
- Remember that an array can "decay" to a pointer to its first element
- Code is syntactically completely correct C
- Returning pointer to a local variable is **undefined behavior**
- Never do this, not even for debugging purposes

What's wrong with this code (part 2)?

```
int *f()
{
  int a[100];
  for(i=0;i<100;i++)
    a[i] = i;
  return a;
}
```

- Return type is `int *`, returning a is not a *type* problem
- Remember that an array can "decay" to a pointer to its first element
- Code is syntactically completely correct C
- Returning pointer to a local variable is **undefined behavior**
- Never do this, not even for debugging purposes
- Any decent compiler will put out warnings

What's wrong with this code (part 3)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  int x = 5;
  int *y = a;
  a = &x;
  free(a);
  return x;
}
```

What's wrong with this code (part 3)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  int x = 5;
  int *y = a;
  a = &x;
  free(a);
  return x;
}
```

What's wrong with this code (part 3)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  int x = 5;
  int *y = a;
  a = &x;
  free(a);
  return x;
}
```

- No check whether `malloc` returned `NULL`
- The function is *so* wrong, that this isn't even really a problem

What's wrong with this code (part 3)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  int x = 5;
  int *y = a;
  a = &x;
  free(a);
  return x;
}
```

- No check whether `malloc` returned `NULL`
- The function is *so* wrong, that this isn't even really a problem
- The `free` is used on a *stack* address

What's wrong with this code (part 3)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  int x = 5;
  int *y = a;
  a = &x;
  free(a);
  return x;
}
```

- No check whether `malloc` returned `NULL`
- The function is *so* wrong, that this isn't even really a problem
- The `free` is used on a *stack* address
- The value of `y` is lost after `return`
- Cannot `free` the allocated memory anymore

`valgrind`

- Memory bugs are hard to find manually
- They are one of the biggest problems in C code
- Luckily there is tool assistance: `valgrind`

valgrind

- Memory bugs are hard to find manually
- They are one of the biggest problems in C code
- Luckily there is tool assistance: `valgrind`
- Run code is a sort of virtual machine, include memory checks
- Muuuuuuch slower than actually running the code, but:
  - Find memory leaks (`malloc` without `free`)
  - Find access to freed memory
  - Find double-free
  - Find branches and memory access depending on uninitialized data

`valgrind`

- Memory bugs are hard to find manually
- They are one of the biggest problems in C code
- Luckily there is tool assistance: `valgrind`
- Run code is a sort of virtual machine, include memory checks
- Muuuuuuch slower than actually running the code, but:
  - Find memory leaks (`malloc` without `free`)
  - Find access to freed memory
  - Find double-free
  - Find branches and memory access depending on uninitialized data
- Many more tools beyond the memory checker in valgrind, e.g.,
  - `cachgrind`, a cache profiler
  - `callgrind`, generating call graphs

`valgrind`

- Memory bugs are hard to find manually
- They are one of the biggest problems in C code
- Luckily there is tool assistance: `valgrind`
- Run code is a sort of virtual machine, include memory checks
- Muuuuuuch slower than actually running the code, but:
  - Find memory leaks (`malloc` without `free`)
  - Find access to freed memory
  - Find double-free
  - Find branches and memory access depending on uninitialized data
- Many more tools beyond the memory checker in valgrind, e.g.,
  - `cachgrind`, a cache profiler
  - `callgrind`, generating call graphs
- `valgrind` is a dynamic analyzer, not static
- For example, no guarantees of branch coverage

valgrind

- Memory bugs are hard to find manually
- They are one of the biggest problems in C code
- Luckily there is tool assistance: `valgrind`
- Run code is a sort of virtual machine, include memory checks
- Muuuuuuch slower than actually running the code, but:
  - Find memory leaks (`malloc` without `free`)
  - Find access to freed memory
  - Find double-free
  - Find branches and memory access depending on uninitialized data
- Many more tools beyond the memory checker in valgrind, e.g.,
  - `cachgrind`, a cache profiler
  - `callgrind`, generating call graphs
- `valgrind` is a dynamic analyzer, not static
- For example, no guarantees of branch coverage
- Generally good practice:
  - run your code in valgrind before submitting/publishing
  - make sure that valgrind reports to errors