C Programming
Introduction, part 2

Advanced Computer Science Studies in Sweden
Introduction to Studies in Embedded Systems
HT 2017

Pontus Ekberg
Pointer arithmetic

Recap:
• A pointer is a memory address
• Pointers can be stored in variables (of type \texttt{int*}, \texttt{char*} etc.)
• \textit{Referencing}: Given a variable \texttt{a}, you get a pointer to it via \texttt{&a}
• \textit{Dereferencing}: Given a pointer \texttt{ptr}, you get what it points to via \texttt{*ptr}
Pointer arithmetic

Recap:
• A pointer is a memory address
• Pointers can be stored in variables (of type `int*`, `char*` etc.)
• Referencing: Given a variable `a`, you get a pointer to it via `&a`
• Dereferencing: Given a pointer `ptr`, you get what it points to via `*ptr`

Pointer arithmetic:
• Explicitly modifying the value of a pointer (i.e., changing the address) is allowed and often used in C.
If \( \text{ptr} \) is of type \( *\text{int} \), then adding 1 to \( \text{ptr} \) will increment \( \text{ptr} \) by \( \text{sizeof(int)} \) etc.
If `ptr` is of type `*int`, then adding 1 to `ptr` will increment `ptr` by `sizeof(int)` etc.

```c
int arr[] = {10,20,30,40,50,60,70,80,90};

int *ptr = arr;

printf("*ptr = %d\n", *ptr);

ptr += 1;
printf("*ptr = %d\n", *ptr);

ptr += 2;
printf("*ptr = %d\n", *ptr);
```
If `ptr` is of type `*int`, then adding 1 to `ptr` will increment `ptr` by `sizeof(int)` etc.

```c
int arr[] = {10,20,30,40,50,60,70,80,90};
int *ptr = arr;

printf("*ptr = %d\n", *ptr);  // *ptr = 10
ptr += 1;
printf("*ptr = %d\n", *ptr);
ptr += 2;
printf("*ptr = %d\n", *ptr);
```

If \( \text{ptr} \) is of type \( \ast \text{int} \), then adding 1 to \( \text{ptr} \) will increment \( \text{ptr} \) by \text{sizeof(int)} \) etc.

\[
\text{int arr[]} = \{10, 20, 30, 40, 50, 60, 70, 80, 90\};
\]

\[
\text{int } \ast \text{ptr} = \text{arr};
\]

\[
\text{printf("}\ast \text{ptr = %d
","} \ast \text{ptr);}
\]

\[
\text{ptr += 1;}
\]

\[
\text{printf("}\ast \text{ptr = %d
","} \ast \text{ptr);}
\]

\[
\text{ptr += 2;}
\]

\[
\text{printf("}\ast \text{ptr = %d
","} \ast \text{ptr);}
\]
If `ptr` is of type `*int`, then adding 1 to `ptr` will increment `ptr` by `sizeof(int)` etc.

```c
int arr[] = {10,20,30,40,50,60,70,80,90};

int *ptr = arr;

printf("*ptr = %d\n", *ptr);  // *ptr = 10
ptr += 1;
printf("*ptr = %d\n", *ptr);  // *ptr = 20
ptr += 2;
printf("*ptr = %d\n", *ptr);  // *ptr = 40
```
The null pointer

For pointer types, there exists a special “null” value that is guaranteed to not point to any other thing in memory.

It is good practice to explicitly set pointers to null when they are not pointing at anything in particular.
The null pointer

For pointer types, there exists a special “null” value that is guaranteed to not point to any other thing in memory.

It is good practice to explicitly set pointers to null when they are not pointing at anything in particular.

```c
int *ptr1;

int *ptr2 = 0;

int *ptr3 = NULL;
```
The null pointer

For pointer types, there exists a special “null” value that is guaranteed to not point to any other thing in memory.

It is good practice to explicitly set pointers to null when they are not pointing at anything in particular.

```
int *ptr1;     // *ptr1 is uninitialized. Reading it results in undefined behavior!
int *ptr2 = 0;
int *ptr3 = NULL;
```
The null pointer

For pointer types, there exists a special “null” value that is guaranteed to not point to any other thing in memory.

It is good practice to explicitly set pointers to null when they are not pointing at anything in particular.

```c
int *ptr1;
```

*ptr1 is uninitialized. Reading it results in undefined behavior!

```c
int *ptr2 = 0;
```

An unadorned 0 is converted to the null value in a pointer context. Dereferencing it is undefined behavior (though typically triggers an OS trap).

```c
int *ptr3 = NULL;
```
The null pointer

For pointer types, there exists a special “null” value that is guaranteed to not point to any other thing in memory.

It is good practice to explicitly set pointers to null when they are not pointing at anything in particular.

```c
int *ptr1;

int *ptr2 = 0;

int *ptr3 = NULL;
```

*ptr1 is uninitialized. Reading it results in undefined behavior!

An unadorned 0 is converted to the null value in a pointer context. Dereferencing it is undefined behavior (though typically triggers an OS trap).

For extra clarity, the preprocessor macro NULL is defined equal to 0, e.g., in stdio.h and stddef.h.
Pointers and arrays

- Pointers and arrays are *almost* equivalent
- Wherever a pointer is expected it is ok to pass an array instead, it will be converted to a pointer to the array's first element
- This *always happens* when passing an array as an argument to a function
Pointers and arrays

- Pointers and arrays are *almost* equivalent
- Wherever a pointer is expected it is ok to pass an array instead, it will be converted to a pointer to the array's first element
- This *always happens* when passing an array as an argument to a function

```c
int f(char arr[]) {
    ...
}
```
Points and arrays

- Pointers and arrays are *almost* equivalent
- Wherever a pointer is expected it is ok to pass an array instead, it will be converted to a pointer to the array's first element
- This *always happens* when passing an array as an argument to a function

```c
int f(char arr[]) {
    ...
}
```

```c
int f(char *arr) {
    ...
}
```

is exactly the same as
Pointers and arrays, cont.

- One difference is when using `sizeof`
  - For an array, `sizeof` gives the total size of the array
  - For a pointer, `sizeof` gives the size of the pointer itself
One difference is when using `sizeof`

→ For an array, `sizeof` gives the total size of the array
→ For a pointer, `sizeof` gives the size of the pointer itself

```c
void f(int arr[]) {
    printf("sizeof(arr) = %lu\n", sizeof(arr));
}

int main() {
    int arr[10];
    int *ptr = arr;

    printf("sizeof(arr) = %lu\n", sizeof(arr));

    printf("sizeof(ptr) = %lu\n", sizeof(ptr));
    f(arr);
}
```
Pointers and arrays, cont.

• One difference is when using `sizeof`
  → For an array, `sizeof` gives the total size of the array
  → For a pointer, `sizeof` gives the size of the pointer itself

```c
void f(int arr[]) {
    printf("sizeof(arr) = %lu\n", sizeof(arr));
}

int main() {
    int arr[10];
    int *ptr = arr;

    printf("sizeof(arr) = %lu\n", sizeof(arr));
    printf("sizeof(ptr) = %lu\n", sizeof(ptr));
    f(arr);
    memset(arr, 0, sizeof(arr));

    return 0;
}
```

`sizeof(arr) = 40`
Pointers and arrays, cont.

- One difference is when using `sizeof`
  → For an array, `sizeof` gives the total size of the array
  → For a pointer, `sizeof` gives the size of the pointer itself

```c
void f(int arr[]) {
    printf("sizeof(arr) = %lu\n", sizeof(arr));
}

int main() {
    int arr[10];
    int *ptr = arr;

    printf("sizeof(arr) = %lu\n", sizeof(arr));
    printf("sizeof(ptr) = %lu\n", sizeof(ptr));
    f(arr);
    assert(sizeof(arr) == 40);
    assert(sizeof(ptr) == 8);
}```
Pointers and arrays, cont.

• One difference is when using `sizeof`
  → For an array, `sizeof` gives the total size of the array
  → For a pointer, `sizeof` gives the size of the pointer itself

```c
void f(int arr[]) {
    printf("sizeof(arr) = %lu\n", sizeof(arr));
}

int main() {
    int arr[10];
    int *ptr = arr;

    printf("sizeof(arr) = %lu\n", sizeof(arr));
    printf("sizeof(ptr) = %lu\n", sizeof(ptr));
    f(arr);
}
```

```
void f(int arr[]) {  
    printf("sizeof(arr) = %lu\n", sizeof(arr));  
}

int main() {  
    int arr[10];  
    int *ptr = arr;  

    printf("sizeof(arr) = %lu\n", sizeof(arr));  
    printf("sizeof(ptr) = %lu\n", sizeof(ptr));  
    f(arr);  
}
```
Pointers and arrays, cont.

• Array indexing is just syntactic sugar for pointer arithmetic!
• The compiler replaces \texttt{arr[i]} by \texttt{*(arr + i)}
Pointers and arrays, cont.

- Array indexing is just syntactic sugar for pointer arithmetic!
- The compiler replaces \texttt{arr[i]} by \texttt{*(arr + i)}

\begin{verbatim}
int arr[] = {1,2,3,4,5};
printf("%d\n", arr[3]);
printf("%d\n", *(arr + 3));
\end{verbatim}
Pointers and arrays, cont.

- Array indexing is just syntactic sugar for pointer arithmetic!
- The compiler replaces \texttt{arr[i]} by \texttt{*(arr + i)}

```c
int arr[] = {1,2,3,4,5};

printf("%d\n", arr[3]);  \boxed{4}

printf("%d\n", *(arr + 3));  \boxed{4}
```
Pointers and arrays, cont.

- Array indexing is just syntactic sugar for pointer arithmetic!
- The compiler replaces `arr[i]` by `*(arr + i)`

```c
int arr[] = {1,2,3,4,5};

printf("%d\n", arr[3]);    // 4
printf("%d\n", *(arr + 3)); // 4
printf("%d\n", 3[arr]);     // 4
```
Pointers and arrays, cont.

- Array indexing is just syntactic sugar for pointer arithmetic!
- The compiler replaces \texttt{arr[i]} by \texttt{*({arr + i})}

```c
int arr[] = {1,2,3,4,5};

printf("%d\n", arr[3]);  \textcolor{red}{4}

printf("%d\n", *(arr + 3));  \textcolor{red}{4}

printf("%d\n", 3[arr]);  \textcolor{red}{4}
```
Arrays and arrays, cont.

- Array indexing is just syntactic sugar for pointer arithmetic!
- The compiler replaces \texttt{arr[i]} by \texttt{*(arr + i)}

```c
int arr[] = \{1, 2, 3, 4, 5\};

printf("%d\n", arr[3]); \quad 4
printf("%d\n", *(arr + 3)); \quad 4
printf("%d\n", 3[arr]); \quad 4
printf("%d\n", *(3 + arr)); \quad 4
```
More on strings

Recall:
• Strings in C are arrays of char.
• By convention, the null character '\0' is used to mark the end.
More on strings

Recall:
- Strings in C are arrays of **char**.
- By convention, the null character `\0` is used to mark the end.

Therefore:
- String functions have *no way* of knowing the length of a string that isn't null-terminated unless you explicitly pass it along.
More on strings

Recall:
• Strings in C are arrays of char.
• By convention, the null character '\0' is used to mark the end.

Therefore:
• String functions have no way of knowing the length of a string that isn't null-terminated unless you explicitly pass it along.

There are several useful string functions in string.h.

Check the documentation: man string.h in a terminal!
Structs

• In C you can define complex data types in the form of structs
• A struct is a fixed collection of members of varying types
• In C you can define complex data types in the form of structs
• A struct is a fixed collection of members of varying types

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

struct person {
    char name[20];
    unsigned int age;
};
```
• In C you can define complex data types in the form of structs
• A struct is a fixed collection of members of varying types

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

struct person {
    char name[20];
    unsigned int age;
};

int main() {
    struct person p1;
    struct person p2 = {"Selma", 5};

    strncpy(p1.name, "Hedvig", 20);
    p1.age = 1;

    printf("%s is %d years old\n", p1.name, p1.age);
    return 0;
}
```
Structs, cont.

• To get the size of a struct, use `sizeof(struct person)` etc.
  → The size might be larger than the sum of the components
• Struct definitions can't be recursive, but can include pointers to structs of the same type
Structs, cont.

• To get the size of a struct, use `sizeof(struct person)` etc.
  → The size might be larger than the sum of the components
• Struct definitions can't be recursive, but can include pointers to structs of the same type

```c
struct person {
    char name[20];
    unsigned int age;
    struct person mother;
};
```
• To get the size of a struct, use `sizeof(struct person)` etc. → The size might be larger than the sum of the components
• Struct definitions can't be recursive, but can include pointers to structs of the same type

```c
struct person {
    char name[20];
    unsigned int age;
    struct person mother;
};
```

```c
struct person {
    char name[20];
    unsigned int age;
    struct person *mother;
};
```
• To get the size of a struct, use `sizeof(struct person)` etc. → The size might be larger than the sum of the components
• Struct definitions can't be recursive, but can include pointers to structs of the same type

```c
struct person {
    char name[20];
    unsigned int age;
    struct person mother;
};
```

```c
struct node {
    int value;
    struct node *left;
    struct node *right;
};
```

```c
struct person {
    char name[20];
    unsigned int age;
    struct person *mother;
};
```
Structs, cont.

- Use . to access the individual fields in a struct
- If you have a pointer ptr to a struct, use -> as shorthand
  \( \rightarrow \) ptr->field is the same as (*ptr).field
• Use . to access the individual fields in a struct
• If you have a pointer ptr to a struct, use -> as shorthand
  → ptr->field is the same as (*ptr).field

```
struct node {
    int value;
    struct node *left;
    struct node *right;
};

int main() {
    struct node leaf1 = {5}; /* The remaining fields are zeroed */
    struct node leaf2 = {12};
    struct node root = {7, &leaf1, &leaf2};

    printf("Value of root = %d\n", root.value);
    printf("Value of leaf1 = %d\n", root.left->value);
    printf("Value of leaf2 = %d\n", root.right->value);

    return 0;
}
```
Memory

There are three important types of memory allocation:
Memory

There are three important types of memory allocation:

- *Static allocation* happens when you define a *static* or *global* variable. The lifetime of these variables is the entire duration of the program.
Memory

There are three important types of memory allocation:

- **Static allocation** happens when you define a static or global variable. The lifetime of these variables is the entire duration of the program.

- **Automatic allocation** happens when you define an automatic variable, such as a local variable or function parameter. The lifetime of these is the block in which they are defined.
There are three important types of memory allocation:

- **Static allocation** happens when you define a *static* or *global* variable. The lifetime of these variables is the entire duration of the program.

- **Automatic allocation** happens when you define an automatic variable, such as a *local variable* or *function parameter*. The lifetime of these is the block in which they are defined.

- **Dynamic allocation** happens when you *explicitly ask for* some dynamic memory to be allocated. The lifetime of this allocated memory is until you *explicitly free it*.
Typical memory layout

High address (e.g., $2^{64} - 1$)

Low address (e.g., 0)
Typical memory layout

High address (e.g., $2^{64} - 1$)

Low address (e.g., 0)

unmapped
Typical memory layout

High address (e.g., $2^{64} - 1$)

Program code

Low address (e.g., 0)

- text
- unmapped
Typical memory layout

High address (e.g., $2^{64} - 1$)

Program code

- initialized data
- text
- unmapped

Low address (e.g., 0)
Typical memory layout

High address (e.g., $2^{64} - 1$)

Program code

Low address (e.g., 0)

initialized data

Read from the binary file

unmapped
text
Typical memory layout

High address (e.g., $2^{64} - 1$)

Low address (e.g., 0)

Program code

- Uninitialized data
- Initialized data
- Text
- Unmapped

Read from the binary file
Typical memory layout

High address (e.g., $2^{64} - 1$)

Statically allocated memory (Static/global variables etc.)

- Program code
- Uninitialized data
- Initialized data
- Text
- Unmapped

Read from the binary file
Typical memory layout

High address (e.g., \(2^{64} - 1\))

- Statically allocated memory (Static/global variables etc.)
  - Program code
  - Low address (e.g., 0)

- command-line arguments, environment variables etc.
  - uninitialized data
  - initialized data
  - text
  - unmapped

Read from the binary file
Typical memory layout

High address (e.g., $2^{64} - 1$)

- Statically allocated memory (Static/global variables etc.)
  - Program code
  - command-line arguments, environment variables etc.
    - stack
      - uninitialized data
      - initialized data
      - text
      - unmapped

Read from the binary file

Low address (e.g., 0)
Typical memory layout

High address (e.g., $2^{64} - 1$)

- **Automatically allocated memory**
  - (Local variables etc.)

- **Statically allocated memory**
  - (Static/global variables etc.)

- Program code

- Low address (e.g., 0)

- command-line arguments, environment variables etc.

  - stack

- Read from the binary file

  - uninitialized data
  - initialized data
  - text
  - unmapped
Typical memory layout

High address (e.g., $2^{64} - 1$)

- Statically allocated memory (Static/global variables etc.)
  - Program code
  - Uninitialized data
  - Initialized data
  - Text
  - Unmapped

- Automatically allocated memory (Local variables etc.)
  - Stack
  - Heap

Read from the binary file
# Typical memory layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low address</td>
<td>(e.g., 0)</td>
</tr>
<tr>
<td>High address</td>
<td>(e.g., $2^{64} - 1$)</td>
</tr>
<tr>
<td><strong>Automatically allocated memory</strong></td>
<td>(Local variables etc.)</td>
</tr>
<tr>
<td><strong>Dynamically allocated memory</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Statically allocated memory</strong> (Static/global variables etc.)</td>
<td></td>
</tr>
<tr>
<td>Program code</td>
<td></td>
</tr>
<tr>
<td>Low address</td>
<td></td>
</tr>
</tbody>
</table>

- **Stack**
- **Heap**
- **Uninitialized data**
- **Initialized data**
- **Text**
- **Unmapped**

Read from the binary file
Typical memory layout

<table>
<thead>
<tr>
<th>High address (e.g., $2^{64} - 1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Automatically allocated memory</strong></td>
</tr>
<tr>
<td>(Local variables etc.)</td>
</tr>
<tr>
<td><strong>Dynamically allocated memory</strong></td>
</tr>
<tr>
<td><strong>Statically allocated memory</strong></td>
</tr>
<tr>
<td>(Static/global variables etc.)</td>
</tr>
<tr>
<td><strong>Program code</strong></td>
</tr>
<tr>
<td><strong>Low address (e.g., 0)</strong></td>
</tr>
</tbody>
</table>

| command-line arguments,          |
| environment variables etc.       |
| **stack**                         |
| **heap**                          |
| **uninitialized data**            |
| **initialized data**              |
| **text**                          |
| **unmapped**                      |

```c
void func1(int n) {
    float arr[5];
    ...
    if (n > 0)
        func1(n-1);
}
char func2() {
    int x, y, z;
    ...
}
int main() {
    char c;
    func1(3);
    c = func2();
    ...
}
```
Typical memory layout

High address (e.g., $2^{64}-1$)

`command-line arguments, environment variables etc.`

`main`

`heap`

`uninitialized data`

`initialized data`

`text`

`unmapped`

`Automatically allocated memory`

(Local variables etc.)

`Dynamically allocated memory`

`Statically allocated memory` (Static/global variables etc.)

Program code

Low address (e.g., 0)

void func1(int n) {
    float arr[5];
    ...
    if (n > 0)
        func1(n-1);
}

char func2() {
    int x, y, z;
    ...
}

int main() {
    char c;
    func1(3);
    c = func2();
    ...
}
Typical memory layout

- **Low address (e.g., 0)**
- **High address (e.g., $2^{64} - 1$)**

**Automatically allocated memory**
(Local variables etc.)

**Dynamically allocated memory**

**Statically allocated memory** (Static/global variables etc.)

- **Program code**
- **Uninitialized data**
- **Initialized data**
- **Text**
- **Unmapped**

**Command-line arguments, environment variables etc.**

```c
void func1(int n) {
    float arr[5];
    ...
    if (n > 0)
        func1(n-1);
}

char func2() {
    int x, y, z;
    ...
}

int main() {
    char c;
    func1(3);
    c = func2();
    ...
}
```
Typical memory layout

- **Low address (e.g., 0)**
- **High address (e.g., $2^{64}-1$)**

**Automatically allocated memory**
(Local variables etc.)

**Dynamically allocated memory**

**Statically allocated memory** (Static/global variables etc.)

Program code

- Unmapped
- Uninitialized data
- Initialized data
- Text
- Command-line arguments, environment variables etc.

![Memory diagram]

```c
void func1(int n) {
    float arr[5];
    ... 
    if (n > 0)
        func1(n-1);
}

char func2() {
    int x, y, z;
    ... 
}

int main() {
    char c;
    func1(3);
    c = func2();
    ... 
}
```
Typical memory layout

High address (e.g., $2^{64} - 1$)

Automatically allocated memory
(Local variables etc.)

Dynamically allocated memory

Stack

Statically allocated memory (Static/global variables etc.)

Program code

Low address (e.g., 0)

command-line arguments, environment variables etc.

main

func1

func1

func1

heap

uninitialized data

initialized data

text

unmapped

void func1(int n) {
    float arr[5];
    ...
    if (n > 0)
        func1(n-1);
}

dchar func2() {
    int x, y, z;
    ...
}

int main() {
    char c;
    func1(3);
    c = func2();
    ...
}
## Typical memory layout

<table>
<thead>
<tr>
<th>Low address (e.g., 0)</th>
<th>High address (e.g., $2^{64} - 1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statically allocated memory</strong> (Static/global variables etc.)</td>
<td><strong>Automatically allocated memory</strong> (Local variables etc.)</td>
</tr>
<tr>
<td>Program code</td>
<td>command-line arguments, environment variables etc.</td>
</tr>
<tr>
<td></td>
<td>main</td>
</tr>
<tr>
<td></td>
<td>func1</td>
</tr>
<tr>
<td></td>
<td>func1</td>
</tr>
<tr>
<td></td>
<td>func1</td>
</tr>
<tr>
<td></td>
<td>func1</td>
</tr>
<tr>
<td></td>
<td>func1</td>
</tr>
<tr>
<td></td>
<td>heap</td>
</tr>
<tr>
<td></td>
<td>uninitialized data</td>
</tr>
<tr>
<td></td>
<td>initialized data</td>
</tr>
<tr>
<td></td>
<td>text</td>
</tr>
<tr>
<td></td>
<td>unmapped</td>
</tr>
</tbody>
</table>

### Code Example

```c
void func1(int n) {
    float arr[5];
    ...
    if (n > 0)
        func1(n-1);
}

char func2() {
    int x, y, z;
    ...
}

int main() {
    char c;
    func1(3);
    c = func2();
    ...
}
```
Typical memory layout

High address (e.g., $2^{64} - 1$)

- Automatically allocated memory
  (Local variables etc.)

- Dynamically allocated memory

- Statically allocated memory
  (Static/global variables etc.)

- Program code

- Low address (e.g., 0)

<table>
<thead>
<tr>
<th>command-line arguments, environment variables etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
</tr>
<tr>
<td>func1</td>
</tr>
<tr>
<td>func1</td>
</tr>
<tr>
<td>func1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>heap</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>uninitialized data</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialized data</td>
</tr>
<tr>
<td>text</td>
</tr>
<tr>
<td>unmapped</td>
</tr>
</tbody>
</table>

void func1(int n) {
  float arr[5];
  ...
  if (n > 0)
    func1(n-1);
}

char func2() {
  int x, y, z;
  ...
}

int main() {
  char c;
  func1(3);
  c = func2();
  ...
}
Typical memory layout

High address (e.g., $2^{64} - 1$)

- **Automatically allocated memory** (Local variables etc.)
- **Dynamically allocated memory**
- **Statically allocated memory** (Static/global variables etc.)

Program code

Low address (e.g., 0)

- text
- initialized data
- uninitialized data
- heap
- command-line arguments, environment variables etc.

void func1(int n) {
    float arr[5];
    ... 
    if (n > 0)
        func1(n-1);
}

char func2() {
    int x, y, z;
    ...
}

int main() {
    char c;
    func1(3);
    c = func2();
    ...
}
Typical memory layout

High address (e.g., $2^{64} - 1$)

- **Automatically allocated memory** (Local variables etc.)
- **Dynamically allocated memory**
- **Statically allocated memory** (Static/global variables etc.)
- **Program code**
- **Low address (e.g., 0)**

<table>
<thead>
<tr>
<th>command-line arguments, environment variables etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
</tr>
<tr>
<td>func1</td>
</tr>
</tbody>
</table>

heap

<table>
<thead>
<tr>
<th>uninitialized data</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialized data</td>
</tr>
<tr>
<td>text</td>
</tr>
<tr>
<td>unmapped</td>
</tr>
</tbody>
</table>

void func1(int n) {
    float arr[5];
    ...
    if (n > 0)
        func1(n-1);
}

char func2() {
    int x, y, z;
    ...
}

int main() {
    char c;
    func1(3);
    c = func2();
    ...
}
Typical memory layout

High address (e.g., $2^{64} - 1$)

- Automatically allocated memory
  (Local variables etc.)

- Dynamically allocated memory

- Statically allocated memory
  (Static/global variables etc.)

- Program code

Low address (e.g., 0)

command-line arguments, environment variables etc.

main

heap

uninitialized data

initialized data

text

unmapped

void func1(int n) {
    float arr[5];
    ... 
    if (n > 0)
        func1(n-1);
}

char func2() {
    int x, y, z;
    ...
}

int main() {
    char c;
    func1(3);
    c = func2();
    ...
}
### Typical memory layout

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low address (e.g., 0)</td>
<td>Program code, initialized data, text, unmapped</td>
</tr>
<tr>
<td>High address (e.g., (2^{64} - 1))</td>
<td>Statically allocated memory (Static/global variables etc.)</td>
</tr>
<tr>
<td></td>
<td>Dynamically allocated memory (Local variables etc.)</td>
</tr>
<tr>
<td></td>
<td>Automatically allocated memory</td>
</tr>
</tbody>
</table>

#### Code Snippets

```c
void func1(int n) {
    float arr[5];
    ...  
    if (n > 0)  
        func1(n-1);
}

char func2() {
    int x, y, z;
    ...  
}

int main() {
    char c;
    func1(3);
    c = func2();
    ...  
}
```
Typical memory layout

High address (e.g., \(2^{64} - 1\))

- Automatically allocated memory
  - (Local variables etc.)

- Dynamically allocated memory
- Statically allocated memory
  - (Static/global variables etc.)

Low address (e.g., 0)

Program code

- Uninitialized data
- Initialized data
- Text
- Unmapped

command-line arguments, environment variables etc.

main

heap

void func1(int n) {
    float arr[5];
    ...
    if (n > 0)
        func1(n-1);
}

char func2() {
    int x, y, z;
    ...
}

int main() {
    char c;
    func1(3);
    c = func2();
    ...
}
Dynamically allocated memory

- Dynamic memory (on the heap) is allocated when you ask for it!
- Use functions `malloc`, `calloc` or `realloc` in `stdlib.h`.
- This memory is allocated until you hand it back with `free`. 
Dynamically allocated memory

- Dynamic memory (on the heap) is allocated when you ask for it!
- Use functions `malloc`, `calloc` or `realloc` in `stdlib.h`.
- This memory is allocated until you hand it back with `free`.
  (Beware of memory leaks: There is no garbage collection!)
Dynamically allocated memory

- Dynamic memory (on the heap) is allocated when you ask for it!
- Use functions `malloc`, `calloc` or `realloc` in `stdlib.h`.
- This memory is allocated until you hand it back with `free`.
  (Beware of memory leaks: There is no garbage collection!)

```c
void *malloc(size_t size);
void free(void *ptr);
```
Dynamically allocated memory

• Dynamic memory (on the heap) is allocated when you ask for it!
• Use functions `malloc`, `calloc` or `realloc` in `stdlib.h`.
• This memory is allocated until you hand it back with `free`. (Beware of memory leaks: There is no garbage collection!)

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

```c
int *buf = malloc(10 * sizeof(int)); /* Allocate space */

if (NULL == buf) {
    /* Do error handling */
}

/* Do stuff with buf... */

free(buf); /* Free the space when done */
```
Typical memory layout

```
void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
```
Typical memory layout

<table>
<thead>
<tr>
<th>command-line arguments, environment variables etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
</tr>
<tr>
<td>heap</td>
</tr>
<tr>
<td>uninitialized data</td>
</tr>
<tr>
<td>initialized data</td>
</tr>
<tr>
<td>text</td>
</tr>
<tr>
<td>unmapped</td>
</tr>
</tbody>
</table>

```c
void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
    ...
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
```
Typical memory layout

<table>
<thead>
<tr>
<th>command-line arguments, environment variables etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n * sizeof(int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uninitialized data</td>
</tr>
<tr>
<td>initialized data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>unmapped</td>
</tr>
</tbody>
</table>

```c
void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
    ...
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
```
Typical memory layout

<table>
<thead>
<tr>
<th>command-line arguments, environment variables etc.</th>
<th>main</th>
<th>ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>n * sizeof(int)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uninitialized data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>initialized data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unmapped</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
    ...
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
```
Typical memory layout

<table>
<thead>
<tr>
<th>command-line arguments, environment variables etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
</tr>
<tr>
<td>n * sizeof(int)</td>
</tr>
<tr>
<td>uninitialized data</td>
</tr>
<tr>
<td>initialized data</td>
</tr>
<tr>
<td>text</td>
</tr>
<tr>
<td>unmapped</td>
</tr>
</tbody>
</table>

```c
void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
    ...
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
```
void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
    ...
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
Typical memory layout

```c
void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
    ...
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
```
Typical memory layout

void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
    ...
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
    ...
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
Typical memory layout

void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
    ...
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
void func1(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        ptr[i] = f(i);
    }
    ...
}

void func2(int *ptr, int n) {
    /* Do stuff with ptr */
}

int main() {
    int n;
    ...
    int *ptr = malloc(n * sizeof(int));
    func1(ptr, n);
    func2(ptr, n);
    free(ptr);
    ...
}
Typical memory layout

To avoid memory leaks and undefined behavior:

- A pointer to each dynamically allocated memory region should be passed to `free` exactly once!
- Never follow a pointer that has been passed to `free`.
Other useful things

- Bitwise operations (and, or, xor etc.)
- Enums
- Union types
- #defines
- Parallelism with pthreads, OpenMP etc.
- Inline assembly code
- ...