Signals, Instruments, and Systems – W3

C Programming & Memory Management in C
Any other questions about the lab or last lecture?
Outline

- Pointers
- Parameter passing
- Dynamic allocation of memory
- Debugging with gdb
Argument passing in C

- Arguments are always passed \textit{by value} in C function calls! This means that \textbf{local copies} of the values of the arguments are passed to the routines!

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;

    exchange(a, b);

    printf("Main: a = %d, b = %d\n", a, b);

    return 0;
}

computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 5, b = 7
```
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a,b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}
```

Output:
```
computer:~> ./exchange
```

#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a, b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

Output:
computer:~> ./exchange

Output:
Exchange: a = 5, b = 7
Main: a = 7, b = 5
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a, b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

Output:
computer:~> ./exchange

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Computer memory

```
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a, b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}
```

Output:
```
computer:~> ./exchange
```

```sh
5
7
```
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a, b);

  printf("Main: a = %d, b = %d\n", a, b);
  return 0;
}
```

Output:

```
computer:~/ ~> ./exchange
```

```
Output:
```
```
```
```
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a,b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}
```

Output:
```
computer:~> ./exchange
```

```
| a = 5 |
| b = 7 |
| tmp = 5 |
```

Computer memory: copied arguments

Computer memory area: exchange

#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a, b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

Output:
computer:~> ./exchange

```
Exchange: a = 5, b = 7
Main: a = 7, b = 7
```
What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a, b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}
```

Output:
```
computer:~> ./exchange
```

- **Computer memory**
  - `a = 5`
  - `b = 7`
  - `tmp = 5`
  - `a = 7`
  - `b = 5`

- **copied arguments**
  - `b = 7`
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a, b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

Output:

computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a, b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

Output:

computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
What happens?

```
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a, b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}
```

Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 5, b = 7
```
Pointers

```c
int i;

int* pi;

int *pi;
```
Pointers

float f;

float* pf;

||

float *pf;
Pointers

float f;

float** pf;

II

float ***pf;
## Pointers

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td></td>
</tr>
<tr>
<td>5464</td>
<td></td>
</tr>
<tr>
<td>5468</td>
<td></td>
</tr>
</tbody>
</table>
Pointers

```c
int a = 5;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td></td>
</tr>
<tr>
<td>5464</td>
<td></td>
</tr>
<tr>
<td>5468</td>
<td></td>
</tr>
</tbody>
</table>
Pointers

int a = 5;

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td></td>
</tr>
<tr>
<td>5468</td>
<td></td>
</tr>
</tbody>
</table>
Pointers

```c
int a = 5;
int b = 7;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td></td>
</tr>
<tr>
<td>5468</td>
<td></td>
</tr>
</tbody>
</table>
Pointers

```
int a = 5;
int b = 7;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td>b = 7</td>
</tr>
<tr>
<td>5468</td>
<td></td>
</tr>
</tbody>
</table>
Pointers

```c
int a = 5;
int b = 7;
int* pa;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td>b = 7</td>
</tr>
<tr>
<td>5468</td>
<td></td>
</tr>
</tbody>
</table>
Pointers

int a = 5;
int b = 7;
int* pa = &a;

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td>b = 7</td>
</tr>
<tr>
<td>5468</td>
<td></td>
</tr>
</tbody>
</table>
Pointers

```c
int a = 5;
int b = 7;
int* pa = &a;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td>b = 7</td>
</tr>
<tr>
<td>5468</td>
<td></td>
</tr>
</tbody>
</table>
Pointers

```c
int a = 5;
int b = 7;
int* pa = &a;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td>b = 7</td>
</tr>
<tr>
<td>5468</td>
<td>pa = 5460</td>
</tr>
</tbody>
</table>

address-of operator
## Pointers

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>0x5464</td>
<td>b = 7</td>
</tr>
<tr>
<td>0x5468</td>
<td>pa = 0x5460</td>
</tr>
</tbody>
</table>

- **pa** points to the address of **a**.
- **b** is stored at address 0x5464.

The addresses are in hexadecimal format.
Pointers

\[ \text{pa} = \&\text{b}; \]

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>( a = 5 )</td>
</tr>
<tr>
<td>5464</td>
<td>( b = 7 )</td>
</tr>
<tr>
<td>5468</td>
<td>( \text{pa} = 5460 )</td>
</tr>
</tbody>
</table>
Pointers

\[ \text{pa} = \&b; \]

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>(a = 5)</td>
</tr>
<tr>
<td>5464</td>
<td>(b = 7)</td>
</tr>
<tr>
<td>5468</td>
<td>(\text{pa} = 5464)</td>
</tr>
</tbody>
</table>
Pointers

```
pa = &b;
*pa = 42;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td>b = 7</td>
</tr>
<tr>
<td>5468</td>
<td>pa = 5464</td>
</tr>
</tbody>
</table>

```
Pointers

pa = &b;

*pa = 42;

indirection operator

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td>b = 7</td>
</tr>
<tr>
<td>5468</td>
<td>pa = 5464</td>
</tr>
</tbody>
</table>
Pointers

\[ \text{pa} = \&b; \]
\[ *\text{pa} = 42; \]

indirection operator

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td>b = 42</td>
</tr>
<tr>
<td>5468</td>
<td>pa = 5464</td>
</tr>
</tbody>
</table>
Pointers

```c
pa = &b;
*pa = 42;
a = *pa;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 5</td>
</tr>
<tr>
<td>5464</td>
<td>b = 42</td>
</tr>
<tr>
<td>5468</td>
<td>pa = 5464</td>
</tr>
</tbody>
</table>
Pointers

\[ \text{pa} = \&\text{b}; \]
\[ \text{*pa} = 42; \]
\[ \text{a} = \text{*pa}; \]

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>a = 42</td>
</tr>
<tr>
<td>5464</td>
<td>b = 42</td>
</tr>
<tr>
<td>5468</td>
<td>pa = 5464</td>
</tr>
</tbody>
</table>
Argument passing in C

- Arguments are always passed by value in C function calls! This means that local copies of the values of the arguments are passed to the routines!

```c
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
    printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(a, b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}
```

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 5, b = 7
```
How to solve the problem?

- By using **pointers**, i.e. variables that contain the address of another variable!

```c
#include <stdio.h>

void exchange(int* pa, int* pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;

    exchange(&a, &b);

    printf("Main: a = %d, b = %d\n", a, b);

    return 0;
}
```

**Output:**

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 7, b = 5
```

**int* pa and int* pb** are pointers!
What happens now?

```c
#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(&a,&b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}
```

Output:
```
computer:~> ./exchange
```
What happens now?

#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(&a,&b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

**Output:**

computer:~> ./exchange
```c
#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(&a, &b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}
```

Output:
```
computer:~> ./exchange
Exchange: a = 5, b = 7
Main: a = 7, b = 5
```
#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(&a,&b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

Output:

computer:~> ./exchange

What happens now?
What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(&a, &b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

Output:
computer:~> ./exchange
```
#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(&a, &b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

Output:

computer:~> ./exchange
#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;

    exchange(&a, &b);

    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

Output:
computer:~> ./exchange
What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;
    
    exchange(&a, &b);
    
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}
```

Output:
```
computer:~> ./exchange
```

#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;

    exchange(&a, &b);

    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

**Output:**

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
```
#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;
    exchange(&a, &b);
    printf("Main: a = %d, b = %d\n", a, b);
    return 0;
}

Output:

computer:~> ./exchange

computer:~> Exchange: a = 7, b = 5
What happens now?

```c
#include <stdio.h>

void exchange(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
    int a = 5;
    int b = 7;

    exchange(&a,&b);

    printf("Main: a = %d, b = %d\n", a, b);

    return 0;
}
```

Output:
```
computer:~> ./exchange
Exchange: a = 7, b = 5
computer:~> Main: a = 7, b = 5
```
Arrays

- Arrays and pointers are closely related.

```c
float v[3];
v[0] = 1.3;
v[1] = 4.5;
v[2] = 5.2;
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>float</td>
<td>None</td>
</tr>
<tr>
<td>v[1]</td>
<td>float</td>
<td>v+1=12</td>
</tr>
</tbody>
</table>

- `v == & (v[0])`
- The expression `v[0]` is the same as `*(v+0)`
- The expression `v[1]` is the same as `*(v+1)`
Passing an array to a function

```
#include <stdio.h>

#define SIZE 3

void g(int* array_p, int const size) {
    int i;

    for (i = 0; i < size; ++i) {
        array_p[i] = 2 * (i+1);
    }
}

int main(void) {
    int i;
    int array[SIZE] = {0, 0, 0};

    g(array, SIZE);

    for (i = 0; i < SIZE; ++i) {
        printf("%d:%d ", i, array[i]);
    }

    return 0;
}
```

- The two variables `array_p` and `array` are not the same (`array_p` is a pointer to the first element of `array`!)
- For the purpose of modifying the array from the function `g()`, `array_p` acts the same as `array`
- Here is the output of the program:

```
computer:~> gcc -o array2fun array2fun.c
computer:~> ./array2fun
computer:~> 0:2 1:4 2:6
```
Strings

- There is no string type in C. Instead, we use arrays of `char`, i.e. the type `char*`.

  ```c
  char str[] = "hello";
  ```

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>char*</td>
<td>none</td>
<td>2</td>
</tr>
<tr>
<td>str[4]</td>
<td>char</td>
<td>str+4 (0x6)</td>
<td>‘o’</td>
</tr>
<tr>
<td>str[2]</td>
<td>char</td>
<td>str+2 (0x4)</td>
<td>‘l’</td>
</tr>
</tbody>
</table>

- You can use the `printf` to print out chains of characters. It will read up to the character ‘\0’.

  ```c
  printf("\%s",str);      \rightarrow\ computer:~>\ hello
  printf("\%s",str+3);    \rightarrow\ computer:~>\ lo
  ```
Memory: a more realistic approach

- In a real computer, memory is organized into blocks of 8 bits, called **bytes**.

- On most modern computers, each byte has its own address.

- Memory is **limited**, not only in terms of the number of RAM modules that are installed, but also in terms of the number of addresses available.

- Furthermore, a program is not allowed to use (read and/or write) all bytes: some are reserved by the operating system. If you try to access them (using a pointer), your program will crash (segmentation fault or bus error).
Memory: a more realistic approach

- In a real computer, memory is organized into blocks of 8 bits, called **bytes**.

- On most modern computers, each byte has its own address.

- Memory is **limited**, not only in terms of the number of RAM modules that are installed, but also in terms of the number of addresses available.

- Furthermore, a program is not allowed to use (read and/or write) all bytes: some are reserved by the operating system. If you try to access them (using a pointer), your program will crash (segmentation fault or bus error).

```c
int *p = 1;
*p = 0;  // segmentation fault (trying to write at address 1)
```
The size of the data types

- Each data type requires a certain number of bytes to be stored in memory, and this size can change as a function of the operating system (Windows, Linux, etc.) and the architecture of the system.

- The function `sizeof(type)` returns the size of the data type (in bytes).

```c
printf("%d",sizeof(char));      /* prints 1 */
printf("%d",sizeof(short));    /* prints 2 */
printf("%d",sizeof(int));      /* prints 4 */
printf("%d",sizeof(long));     /* prints 4 */
printf("%d",sizeof(float));    /* prints 4 */
printf("%d",sizeof(double));   /* prints 8 */
```
The size of pointers

- **Reminder**: a pointer is a variable that contains the address of another variable.

- Therefore, the size of any pointer is constant, regardless of the data type that it points to (since it contains only the address of the variable, which does not depend on its type, obviously).

```c
printf("%d", sizeof(char*));    /* prints 4 */
printf("%d", sizeof(short*));   /* prints 4 */
printf("%d", sizeof(int*));     /* prints 4 */
printf("%d", sizeof(long*));    /* prints 4 */
printf("%d", sizeof(float*));   /* prints 4 */
printf("%d", sizeof(double*));  /* prints 4 */
```

On a 32-bit computer
The size of pointers

- **Reminder**: a pointer is a variable that contains the address of another variable.
- Therefore, the size of any pointer is constant, regardless of the data type that it points to (since it contains only the address of the variable, which does not depend on its type, obviously).

```c
printf("%d",sizeof(char*));    /* prints 8 */
printf("%d",sizeof(short*));   /* prints 8 */
printf("%d",sizeof(int*));     /* prints 8 */
printf("%d",sizeof(long*));    /* prints 8 */
printf("%d",sizeof(float*));   /* prints 8 */
printf("%d",sizeof(double*));  /* prints 8 */
```

**On a 64-bit computer**
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(&p2[1]-1) /= 2;

    printf("i = %d\n", i);

    return 0;
}

Output:
computer:~> ./ pointers
20
A (tortuous) pointer example

```c
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(&p2[1]-1) /= 2;

    printf("i = %d\n", i);

    return 0;
}
```

Output:
```
computer:~> ./pointers
20
```
A (tortuous) pointer example

#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(&p2[1]-1) /= 2;

    printf("i = %d\n", i);

    return 0;
}

Output:
computer:~/> ./pointers
Output: 20
A (tortuous) pointer example

#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(&p2[1]-1) /= 2;

    printf("i = %d\n", i);

    return 0;
}

Output:
computer:~> ./pointers
20
A (tortuous) pointer example

```c
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(&p2[1]-1) /= 2;

    printf("i = %d\n", i);

    return 0;
}
```

Output:
```
computer:~> ./pointers
20
```
A (tortuous) pointer example

```c
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(&p2[1]-1) /= 2;

    printf("i = %d\n", i);

    return 0;
}
```

Output:
```
computer:~> ./pointers
```

```
20
```
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(p2[1] - 1) /= 2;

    printf("i = %d\n", i);

    return 0;
}
A (tortuous) pointer example

#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *((&p2[1]-1) / 2);

    printf("i = %d\n", i);

    return 0;
}
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(&p2[1]-1) /= 2;

    printf("i = %d\n", i);

    return 0;
}

Output:
computer:~> ./pointers

20
A (tortuous) pointer example

```c
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *((&p2[1]-1) /= 2);

    printf("i = %d\n", i);

    return 0;
}
```

Output:
```
computer:~> ./pointers
```

Output:
```
i = 5
```

Computer memory:
```
0: i = 5
4: p2[1]=?
8: p1 = 24
12: p2 = 4
16: 20
20: 24
24: &p2[1] == 8
28: &p2[1]-1 == 4
```
A (tortuous) pointer example

```c
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(&p2[1]-1) /= 2;

    printf("i = %d\n", i);

    return 0;
}
```

Output:
```
computer:~> ./pointers
computer:~> i = 5
```
Dynamic allocation of memory

- MATLAB automatically grows matrices as you continue to add more elements.
- These data structures are **dynamical** because they grow automatically in memory as you add data to them.
- In C, you **cannot** do that without managing memory yourself.
- In this code sample, for instance, the array `signal` can contain 50 integers and you cannot make it grow further.
- In many cases, you do not know at **compile time** the size of your data structure. In such cases, you need to allocate memory dynamically!

```c
int signal[50];
signal[0] = 0;
signal[1] = 4;
signal[2] = 5;
signal[3] = 4;
signal[4] = 3;
...```

This value has to be a constant!
Dynamic allocation of memory

- To allocate a certain amount of memory, you can use the function `malloc(size)`, where `size` is the number of bytes of memory requested (which does not have to be constant).
- `malloc` returns a pointer to the first byte of memory which has been allocated.
- As a result, the static array declaration `int signal[50]` becomes, in its dynamic version:

```c
int* signal = malloc(50 * sizeof(int));
signal[0]  = 0;
signal[1]  = 4;
signal[2]  = 5;
signal[3]  = 4;
signal[4]  = 3;
...
```
- This value does not have to be a constant!
Freeing the memory

- If you allocated some memory dynamically, the compiler will **not** take care of freeing the allocated block of memory when you no longer need it.
- Use the function `free(void *ptr)` to make the block available to be allocated again.
- If you perform a `malloc` without its `free` counterpart, you will create a **memory leak**.
- Therefore, write a `free` for each `malloc` you write!
- After you free memory, you can **no longer** access it!

```c
int* signal = malloc(50 * sizeof(int));
// ...
free(signal);
```
Dynamically allocating memory

#include <stdlib.h>

#define MAX_SIZE 1000000

int main() {
    int i;
    int *v; // a vector

    // create a vector of size i
    for (i = 1; i < MAX_SIZE; ++i) {
        v = malloc(i*sizeof(int));
        // do something with vector v
    }

    return 0;
}

- Each iteration of the loop, an increasingly larger chunk of memory is allocated with malloc.
- These chunks are never freed, and the program allocates a total of 2,000 GB of memory before terminating!
Dynamically allocating memory

```c
#include <stdlib.h>

#define MAX_SIZE 1000000

int main() {
    int i;
    int *v; // a vector

    // create a vector of size i
    for (i = 1; i < MAX_SIZE; ++i) {
        v = malloc(i*sizeof(int));
        // do something with vector v
        free(v); // free memory
    }

    return 0;
}
```

- Each iteration of the loop, an increasingly larger chunk of memory is allocated with `malloc`
- These chunks are never freed, and the program allocates a total of 2,000 GB of memory before terminating!
- Calling `free` inside the loop means that we never allocate more than 4 MB at a time
Beyond this lecture

- What you learned today are the *basics* of memory management, i.e., *what you need to know as a C programmer*.

- There are further subtleties, which we **do not expect you to understand in depth**, but it is worth knowing that they exist:
  - the ordering of individually addressable units (words, bytes, or even bits) within a longer data word (endianness) might differ from platform to platform
  - memory is actually divided into two parts: (i) the *stack*, on which variables that are declared at **compile time** are stored in order of **decreasing address**; (ii) the *heap*, on which variables that are **dynamically allocated** are stored.
  - there are further types of memory, which you cannot access in C without recouring to assembler instructions: (i) the *registers*, which are located inside the processor, are extremely fast, but very limited (a few hundreds of bytes); (ii) the *cache*, which is a fast, but small memory (a few megabytes), and is used by the processor to perform “caching” (i.e., pre-fetching and storing chunks of data that are likely to be used or re-used soon).

- Most of these details are **platform-dependent** (and therefore mostly handled by the compiler)
Debugging with gdb
A (tortuous) pointer example

#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *((&p2[1]-1) /= 2;

    return 0;
}
A (tortuous) pointer example

```c
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(p2[1]-1) /= 2;

    return 0;
}
```

What is the value of `i`?
A (tortuous) pointer example

#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *(&p2[1]-1) /= 2;

    return 0;
}

\( \text{What is the value of } i \)\?

\( \text{What about now?} \)
Debugging

- Debuggers allow you to step through and examine the effects of your code as it executes.
- Many IDEs have a visual debugger built in, but in this class we will use **gdb**, which operates from the command line.
- **gdb** has tons of features, but we only need to know a few for it to be an extremely powerful tool.

```
$ gcc -g -o pointers pointers.c
$ gdb ./pointers
(gdb) start
```
Basic commands

- Start your program by typing `start` at the `gdb` prompt.
- Your program will execute until it reaches a "breakpoint". A breakpoint is automatically inserted at the first line of your main function.
- Breakpoints are added with "break filename.c:<line>"
- Execution can be resumed with "continue"
Debugging example

(gdb)
Debugging example

(gdb) start
Debugging example

(gdb) start
Temporary breakpoint 1, main () at pointers.c:4
4 int i = 10;
(gdb)
Debugging example

(gdb) start
Temporary breakpoint 1, main () at pointers.c:4
4 int i = 10;
(gdb) break pointers.c:10
Debugging example

(gdb) start
Temporary breakpoint 1, main () at pointers.c:4
4 int i = 10;
(gdb) break pointers.c:10
Breakpoint 2 at 0x4011b8: file pointers.c, line 10.
(gdb)
Debugging example

(gdb) start
Temporary breakpoint 1, main () at pointers.c:4
4          int i = 10;
(gdb) break pointers.c:10
Breakpoint 2 at 0x4011b8: file pointers.c, line 10.
(gdb) continue
Debugging example

(gdb) start
Temporary breakpoint 1, main () at pointers.c:4
4     int i = 10;
(gdb) break pointers.c:10
Breakpoint 2 at 0x4011b8: file pointers.c, line 10.
(gdb) continue
Continuing.
Breakpoint 2, main () at pointers.c:10
10  *(p2[1]-1) /= 2;
(gdb)
Inspecting variables

- To inspect the values of different variables, use the "print" command

Breakpoint 2, main () at pointers.c:10
10 *(p2[1]-1) /= 2;
(gdb)
Inspecting variables

- To inspect the values of different variables, use the "print" command

Breakpoint 2, main () at pointers.c:10
10 *(&p2[1]-1) /= 2;
(gdb) print &i
Inspecting variables

- To inspect the values of different variables, use the "print" command

Breakpoint 2, main () at pointers.c:10
10 *(&p2[1]-1) /= 2;
(gdb) print &i
$i = (int *) 0x28abf8
(gdb)
Inspecting variables

- To inspect the values of different variables, use the "print" command

Breakpoint 2, main () at pointers.c:10
10 *(&p2[1]-1) /= 2;
(gdb) print &i
$1 = (int *) 0x28abf8
(gdb) print &p2[1]-1
Inspecting variables

- To inspect the values of different variables, use the "print" command

Breakpoint 2, main () at pointers.c:10
10 *(p2[1]-1) /= 2;
(gdb) print &i
$1 = (int *) 0x28abf8
(gdb) print &p2[1]-1
$2 = (int *) 0x28abf8
Step by step navigation

- Setting a breakpoint on every line of a function would be very tedious!
- Use the `step` and `next` commands to navigate through your code one line at a time
- `step` will enter function calls
- `next` will skip them

```c
int main() {
    myfunction(a);
    printf("a = %d\n", a);
    return 0;
}

void myfunction(int a) {
    // perform calculations
}
```
Step by step navigation

- Setting a breakpoint on every line of a function would be very tedious!
- Use the `step` and `next` commands to navigate through your code one line at a time
- `step` will enter function calls
- `next` will skip them

```c
int main() {
    myfunction(a);
    printf("a = %d\n", a);
    return 0;
}

void myfunction(int a) {
    // perform calculations
}
```
Step by step navigation

- Setting a breakpoint on every line of a function would be very tedious!
- Use the `step` and `next` commands to navigate through your code one line at a time
- `step` will enter function calls
- `next` will skip them

```c
int main() {
    myfunction(a);
    printf("a = %d\n", a);
    return 0;
}

void myfunction(int a) {
    // perform calculations
}
```
Step by step navigation

- Setting a breakpoint on every line of a function would be very tedious!
- Use the `step` and `next` commands to navigate through your code one line at a time
- `step` will enter function calls
- `next` will skip them

```c
int main() {
    myfunction(a);
    printf("a = %d\n", a);
    return 0;
}

void myfunction(int a) {
    // perform calculations
}
```
Step by step navigation

- Setting a breakpoint on every line of a function would be very tedious!
- Use the `step` and `next` commands to navigate through your code one line at a time
- `step` will enter function calls
- `next` will skip them

```c
int main() {
    myfunction(a);
    printf("a = %d\n", a);
    return 0;
}

void myfunction(int a) {
    // perform calculations
}
```
Conclusion
Take-home messages

- A pointer is a variable that contains the address of another variable.

- An array is not a pointer, but acts like one in most cases! Arrays simply address a sequence of values. Memory can be either **statically** (at compile time) or **dynamically** (at run time) allocated:
  - **Static allocation** does not require manual deallocation.
  - **Dynamic allocation** requires manual deallocation (using `free`).

- Recall that computer memory has multiple layers of complexity, **even though we do not expect you to know them in details**.

- Debugging with `printfs` is still okay, but a debugger like `gdb` can be much more useful in many situations