Signals, Instruments, and Systems – W11
An Introduction to Mobile Robotics – Filtering Methods for 2D Localization and C Programming Refresher
Signals, Instruments, and Systems – W11
An Introduction to Mobile Robotics – Filtering Methods for 2D Localization
Outline

• The multi-dimensional Kalman filter

• The Kalman filter algorithm applied to 2D localization

• Particle filters
The Multi-Dimensional Kalman Filter Algorithm and its Application to Localization
Two Key Sources of Information
Recap about 1D Kalman Filter from W10

- Kalman filters have a prediction component and an update or correction component.
- For localization, the prediction component is usually computed through a motion model combined with proprioceptive sensors.
- For localization, the update component is usually computed through a measurement model combined with exteroceptive sensors.
1D Time-Discrete Motion Model

• Noisy, continuous-time motion model:
  \[ \dot{x} = u + \sigma_w^2 \]
  \( x = \) system state
  \( u = \) constant speed (controllable input)
  \( \sigma_w^2 = \) variance Gaussian motion noise

W10, s. 28

• Notation change for time-discrete motion model:
  \[ x_t - x_{t-1} = u_t + \varepsilon_t \]
  \( t = \) time index, implying a certain time-discretization interval \( \Delta t \) or \( T \)
  \( u_t = \) variable speed assumed constant during \( \Delta t \) (controllable input)
  \( \varepsilon_t = \) Gaussian motion noise assumed constant during \( \Delta t \)
1D Time-Discrete Measurement Model

• Noisy, continuous-time measurement model:

\[ z = x + \sigma_z^2 \]

- \( x = \) system state
- \( z = \) observation
- \( \sigma_z^2 = \) variance Gaussian measurement noise

W10, s. 30-31

• Notation change for time-discrete measurement model:

\[ z_t = x_t + \delta_t \]

- \( t = \) time index, implying a certain time-discretization interval \( \Delta t \) or \( T \)
- \( \delta_t = \) Gaussian measurement noise assumed constant during \( \Delta t \)
Multi-Dimensional Kalman Filter

Estimates the state $x$ of a discrete-time controlled process that is governed by the linear stochastic difference equation

$$x_t = A_t x_{t-1} + B_t u_t + \varepsilon_t$$

with a measurement

$$z_t = C_t x_t + \delta_t$$

[Adapted from Thrun et al., 2005]
Components of a Kalman Filter

**$A_t$** Matrix (nxn) that describes how the state evolves from $t$ to $t-1$ without controls or noise.

**$B_t$** Matrix (nxm) that describes how the control $u_t$ changes the state from $t$ to $t-1$.

**$C_t$** Matrix (kxn) that describes how to map the state $x_t$ to an observation $z_t$.

**$\epsilon_t$** Gaussian random vectors representing the process and measurement noise that are assumed to be independent and normally distributed with covariance $R_t$ and $Q_t$, respectively.

[Adapted from Thrun et al., 2005]
Algorithm **Kalman_filter**($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$)

1. **Prediction:**
   
   $$\hat{\mu}_t = A_t \mu_{t-1} + B_t u_t$$

2. $$\hat{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$$

3. **Correction or update:**

4. $$K_t = \hat{\Sigma}_t C_t^T (C_t \hat{\Sigma}_t C_t^T + Q_t)^{-1}$$

5. $$\mu_t = \hat{\mu}_t + K_t (z_t - C_t \hat{\mu}_t)$$

6. $$\Sigma_t = (I - K_t C_t) \hat{\Sigma}_t$$

7. **Return** $\mu_t, \Sigma_t$

[Adapted from Thrun et al., 2005]
Mitigating Localization Uncertainties in Odometry Through Exteroceptive Sensors – The 2D Case
2D Localization with Kalman Filter

[From Siegwart and Nourbakhsh, 2004]
A Five-Step Iterative Recipe

1. Robot pose prediction
2. Actual observations
3. Predicted observations
4. Matching
5. Estimation with Kalman filter
Robot Pose Prediction

- **Input**: motion model of the vehicle, odometry, control actions
- **Output**: estimation mean and co-variance of the pose and the next timestep
Actual Observations

- **Input**: sensor model
- Coordinate transformation needed for getting to the local frame (in this case sensor frame)
- **Output**: depending on the sensing technology and feature (e.g., distance, line, door)
Predicted Observations

- **Input**: predicted robot pose and map
- Coordinate transformation needed for getting to the local frame (in this case sensor frame)
- **Output**: predicted feature observations
Matching

- **Input**: predicted feature observations and actual observations, all in the sensor frame
- **Objective**: matching actual observations to predicted features
- **Innovation metric**: difference predicted and observed measurements (also random variable, with mean and covariance)
- **Validation criterion**: based on innovation falling into validation gates
- **Output**: selection of features worth using in the estimation process
Estimation with Kalman Filter

- **Input**: pose prediction from the motion model and validated observations
- **Objective**: fuse pose prediction from motion model and all validated observations to create next pose
- **Output**: next mean and variance of the pose
Working Around Kalman Filter Limitations: Particle Filters for Localization
Feature-Based Localization

Belief representation through Gaussian distribution

- **Advantages:**
  - Compact (only mean and variance required)
  - Continuous
  - Powerful tools (Kalman Filter)

- **Disadvantages:**
  - Requires Gaussian noise assumption
  - Cannot represent ignorance (“kidnapped robot problem”)
  - Uni-modal

- **Problematic example:**
Particle Filter Localization

Belief representation through particle distribution

• Advantages:
  • Can model arbitrary beliefs
  • No assumptions on noise characteristic

• Disadvantages:
  • No unique solution
  • Not continuous
  • Computationally expensive
  • Tuning required
Particle Filter Localization
Particle Filter Localization
Conclusion
Take Home Messages

• Kalman filters can be formalized for multi-dimensional processes and observations; the Kalman filters assumptions in terms of linearity (motion and measurement models) and noise distributions (white Gaussian) must be fulfilled.

• Multi-dimensional Kalman filters can be used for fusing exteroceptive with proprioceptive sensory data in order to solve 2D (and 3D) problems.

• A Kalman filter assumes linear motion and sensor models characterized by white Gaussian noise: many problems in robot localization do not fulfill these assumptions.

• Particle filters allow for working around limitations of Kalman filters (in localization problems and beyond) at the price of additional complexity and computational cost.
Additional Literature – Week 11

Pointers

http://www.probabilistic-robotics.org/

Books

Signals, Instruments, and Systems - W11

C Programming Refresher
Outline

• Headers & Libraries
• Functions
• Scope of variables: Local, Global, Static
• Pointers and Argument passing
• Arrays, Strings
• Structures, Type Definitions, Enumarations
• Memory management
• Debugging with gdb
“.h” vs. “.c”

• Usually header files (“.h” files) should contain all the necessary functions, structures, typedef and enum declarations such that another programmer can use your code without having to look at your c file.

• C files contain the actual implementation and “hidden” declarations.
Libraries

- Libraries provide special functionality in the form of collections of ready-made functions:

<table>
<thead>
<tr>
<th>Library:</th>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdio.h</td>
<td>printf(const char* format,...)</td>
</tr>
<tr>
<td>math.h</td>
<td>sqrt(double x)</td>
</tr>
<tr>
<td>time.h</td>
<td>gettimeofday()</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>rand()</td>
</tr>
</tbody>
</table>

Usage:

#include <stdlib.h>
#include “my_library.h” : your own collection of function declarations
You need to review:

• Variables and constants (types, sign, global, local etc.)
• Operators (arithmetic, unary, bitwise etc.)
• Controlling execution flow (if, switch, while, do-while, for etc.)
• Standard libraries (stdin, stdout etc.)
• Preprocessor commands
Functions

/* function returning the max between two numbers */
int max(int num1, int num2) {
    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
Scope of variables

Code:

```c
#include<stdio.h>

void func_1();
int a, b = 10;

int main()
{
    func_1();
    func_1();
    func_1();
    // signal to operating system everything works fine
    return 0;
}

void func_1()
{
    int a = 1;
    static int b = 100;
    printf("a = %d\n", a);
    printf("b = %d\n\n", b);
    a++;
    b++;
}
```

Output:

```
a = 1
b = 100
a = 1
b = 101
a = 1
b = 102
```
Argument passing in C

- Reminder:
  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
```

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
```

```
Output:
a = 5
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
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
```
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
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
```

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**

**exchange memory area**
- **a = 7**
- **b = 7**

**copied arguments**
- **a = 5**
- **b = 7**
#include <stdio.h>

void exchange(int a, int b) {
    int tmp = a;
    a = b;
    \[\text{\textcolor{red}{b = tmp;}}\]
    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
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:~> Exchange: a = 7, b = 5
```
```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:~> Exchange: a = 7, b = 5
```
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:~> 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

```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></td>
</tr>
<tr>
<td>5464</td>
<td></td>
</tr>
<tr>
<td>5468</td>
<td></td>
</tr>
</tbody>
</table>

address-of operator
## Pointers

<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>
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} = 5460 )</td>
</tr>
</tbody>
</table>
Pointers

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5460</td>
<td>( \text{a} = 5 )</td>
</tr>
<tr>
<td>5464</td>
<td>( \text{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 (value pointed by this address)

<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
(value pointed by this address)

<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

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

```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 = 42</td>
</tr>
<tr>
<td>5464</td>
<td>b = 42</td>
</tr>
<tr>
<td>5468</td>
<td>pa = 5464</td>
</tr>
</tbody>
</table>
```
How to solve the problem?

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

```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!
#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
```

Addresses

Computer memory

```
0
4
8
12
16
20
24
28
```

Output:
```
42
4
8
12
16
20
24
28
```

```
```
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
```

Addresses

```
0
4  a = 5
8  b = 7
12
16
20
24
28
```

Computer memory
#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
```

Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
</tbody>
</table>

Copied arguments

Memory area

Computer memory
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
```
#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?

```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
```
#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?

#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
Arrays

- Arrays and pointers are closely related.

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

- `v` points to the first element of the array

<table>
<thead>
<tr>
<th>Type</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v</code></td>
<td>float*</td>
<td>4</td>
</tr>
<tr>
<td><code>v[1]</code></td>
<td>float</td>
<td>v+1=12</td>
</tr>
</tbody>
</table>

- `v` is the same as `(&v[0])`
- `v[0]` is the same as `*(v+0)`
- `v[1]` is the same as `*(v+1)`
#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><code>str</code></td>
<td>char*</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td><code>str[4]</code></td>
<td>char</td>
<td><code>str+4</code></td>
<td>‘o’</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);  computer:~> hello
  printf("%s", str+3);  computer:~> lo
  ```
#include <stdio.h>
#include <string.h>

struct Books {
    char  title[50];
    char  author[50];
    char  subject[100];
    int   book_id;
};

int main( ) {
struct Books Book1; /* Declare Book1 of type Book */

/* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;

/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

return 0;
}
Type Definition

Code:

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

typedef struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} Book;

int main() {
    Book book;
    strcpy( book.title, "C Programming");
    strcpy( book.author, "Nuha Ali");
    strcpy( book.subject, "C Programming Tutorial");
    book.book_id = 6495407;

    printf("Book title : %s\n", book.title);
    printf("Book author : %s\n", book.author);
    printf("Book subject : %s\n", book.subject);
    printf("Book book_id : %d\n", book.book_id);

    return 0;
}
```

Output:

Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Enumeration

Code:

```c
#include <stdio.h>

enum week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

int main()
{
    // creating today variable of enum week type
    enum week today;
    today = Wednesday;
    printf("Day %d", today + 1);
    return 0;
}
```

Output:

Day 4
Memory management

```c
(void *)memset(void *str, int c, size_t n)
```

1) Set **array** `double acc[3]` to 0:
   ```c
   memset(arr, 0, sizeof(arr));
   ```

2) Set **structure** `pose_t _pose` to 0:
   ```c
   memset(&_pose, 0, sizeof(pose_t));
   ```

```c
(void *)memcpy(void *dest, const void * src, size_t n)
```

1) Copy **array** `double* gps_position` to **array** `double _meas.gps[3]`:
   ```c
   memcpy(_meas.gps, gps_position, sizeof(_meas.gps));
   ```

2) Copy **structure** `pose_t odo_pose_enc` to **struct** `pose_t odo`:
   ```c
   memcpy(odo, &_odo_pose_enc, sizeof(pose_t));
   ```
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)
Debugging with gdb
A (tortuous) pointer example

```c
#include <stdio.h>

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

    p1 = &p2;
    *p1 = &i;
    *p2 /= 2;

    return 0;
}
```

What is the value of `p1`?

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) start
Temporary breakpoint 1, main () at pointers.c: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) 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
}
```
Conclusion
Take-home messages

- Headers and libraries are critical to have a good structure
- Know the difference: Local, global, static
- A pointer is a variable that contains the address of another variable
- Know the difference: pass by value VS pass by reference
- Know the syntax: structures, enumerations, type definitions
- Know the difference: memcpy VS memset
- An array is not a pointer, but acts like one in most cases! Arrays simply address a sequence of values.
- Debugging with printf is still okay, but a debugger like “gdb” can be more useful in many situations, there is also “valgrind” for memory management.
Additional Literature – Week 11

Programming in C
Stephen G. Koch

C Programming Language
Brian W. Kernighan,
Dennis M. Ritchie

Popular C link
http://www.c-faq.com