C Programming
Introduction, part 1

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

Pontus Ekberg
Two lab slots

- Tuesday Aug 29th, 8:15 – 12:00
- Wednesday Sep 6th, 8:15 – 12:00

- Both start with a short lecture part, followed by time to work on the assignments in the lab (room 1515)
- Refresher!
- Not meant as a course for beginners
  → We assume that you all had some education in programming before
Assignments, Instructions

• Assignments to be solved individually, or in groups of two people

• Deadline for submitting solutions: Friday September 8th, 17:00

• Instruction document found on Studentportalen

• This document contains 18 exercises in (roughly) order of increasing difficulty
  → At least 4 exercises have to be solved to pass the lab
# Exercises

1) Output  
2) Input  
3) Conditionals  
4) Loops  
5) Loops II  
6) Functions  
7) Functions II  
8) Arrays  
9) Pointers and Strings  
10) Malloc and sorting strings  
11) Sorting arrays in linear time  
12) Recursion  
13) Efficient Fibonacci numbers  
14) Cmdl. arguments and file I/O  
15) Doubly linked list  
16) Function pointers  
17) Hamming weight  
18) SCCs in graphs
What is C?

- Imperative programming language
- Statically typed
- Low-level
- Ubiquitous
- Created by Dennis Ritchie in early 1970s to be the language for UNIX
- Standardized in C89 (ANSI C), C99 and C11
- Inspiration for many other languages, e.g., C++, Objective-C, Java, C#
- Good FAQ at http://c-faq.com/
#include <stdio.h>

int x = 5; /* x is a global variable */

/*
  This is a function called square. It takes a single int parameter and
  returns an int.
*/
int square(int n) {
    return n*n;
}

int main(int argc, char **argv) {
    int n = 16;

    printf("The square of %d is %d.\n", x, square(x));
    printf("The square of %d is %d.\n", n, square(n));

    if (x > n) {
        printf("%d is larger than %d.\n", x, n);
    } else {
        printf("%d is not larger than %d.\n", x, n);
    }

    return 0;
}
#include <stdio.h>

int x = 5; /* x is a global variable */

/*
 This is a function called square. It takes a single int parameter and
 returns an int.
 */
int square(int n) {
    return n*n;
}

int main() {
    int n = 16;

    printf("The square of %d is %d.\n", x, square(x));
    printf("The square of %d is %d.\n", n, square(n));

    if (x > n) {
        printf("%d is larger than %d.\n", x, n);
    } else {
        printf("%d is not larger than %d.\n", x, n);
    }

    return 0;
}
`#include <stdio.h>

int x = 5; /* x is a global variable */

/*
 * This is a function called square. It takes a single int parameter and
 * returns an int.
 */
int square(int n) {
    return n*n;
}

int main() {
    int n = 16;

    printf("The square of %d is %d.\n", x, square(x));
    printf("The square of %d is %d.\n", n, square(n));

    if (x > n) {
        printf("%d is larger than %d.\n", x, n);
    } else {
        printf("%d is not larger than %d.\n", x, n);
    }

    return 0;
}

The square of 5 is 25.
The square of 16 is 256.
5 is not larger than 16.
Defining variables

- `type_name` var_name;
- `type_name` var_name = initial_value;
- `int` x;
- `char` c = 'A';
- `unsigned long long` bignum = 1000000000000;
Defining variables

- `type_name` var_name;
- `type_name` var_name = initial_value;
- `int` x;
- `char` c = 'A';
- `unsigned long long` bignum = 10000000000000000000;

Note 1: A variable is defined only in its scope
Defining variables

- `type_name  var_name;`
- `type_name  var_name = initial_value;`
- `int  x;`
- `char  c = 'A';`
- `unsigned long long  bignum = 1000000000000;`

Note 1: A variable is defined only in its scope

Note 2: Reading an uninitialized variable (unless global or static) leads to `undefined` behavior
# Built-in integer types

<table>
<thead>
<tr>
<th>Type name</th>
<th>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>At least 8 bits</td>
<td>The smallest addressable unit that can contain a single character</td>
</tr>
<tr>
<td>short</td>
<td>At least 16 bits</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>At least 16 bits</td>
<td>The “default” integer type</td>
</tr>
<tr>
<td>long</td>
<td>At least 32 bits</td>
<td></td>
</tr>
<tr>
<td>long long</td>
<td>At least 64 bits</td>
<td>Only since C99</td>
</tr>
</tbody>
</table>
## Built-in integer types

<table>
<thead>
<tr>
<th>Type name</th>
<th>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>At least 8 bits</td>
<td>The smallest addressable unit that can contain a single character</td>
</tr>
<tr>
<td>short</td>
<td>At least 16 bits</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>At least 16 bits</td>
<td>The “default” integer type</td>
</tr>
<tr>
<td>long</td>
<td>At least 32 bits</td>
<td></td>
</tr>
<tr>
<td>long long</td>
<td>At least 64 bits</td>
<td>Only since C99</td>
</tr>
</tbody>
</table>

Note: Each can be specified as **signed** or **unsigned**
# Built-in integer types

<table>
<thead>
<tr>
<th>Type name</th>
<th>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char (neither by default)</td>
<td>At least 8 bits</td>
<td>The smallest addressable unit that can contain a single character</td>
</tr>
<tr>
<td>short (signed by default)</td>
<td>At least 16 bits</td>
<td></td>
</tr>
<tr>
<td>int (signed by default)</td>
<td>At least 16 bits</td>
<td>The “default” integer type</td>
</tr>
<tr>
<td>long (signed by default)</td>
<td>At least 32 bits</td>
<td></td>
</tr>
<tr>
<td>long long (signed by default)</td>
<td>At least 64 bits</td>
<td>Only since C99</td>
</tr>
</tbody>
</table>

Note: Each can be specified as signed or unsigned
Other integer types

- Since C99, there are more types defined in `stdint.h`
- Usually better to include `inttypes.h` for some extras

<table>
<thead>
<tr>
<th>Type name</th>
<th>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>intN_t</td>
<td>Exactly N bits (N = 8, 16, 32, 64, ?)</td>
<td>Only available if possible for the implementation</td>
</tr>
<tr>
<td>uintN_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int_leastN_t</td>
<td>At least N bits (N = 8, 16, 32, 64, ?)</td>
<td>The smallest integer type available with at least N bits</td>
</tr>
<tr>
<td>uint_leastN_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int_fastN_t</td>
<td>At least N bits (N = 8, 16, 32, 64, ?)</td>
<td>The fastest integer type available with at least N bits</td>
</tr>
<tr>
<td>uint_fastN_t</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Built-in floating-point types

<table>
<thead>
<tr>
<th>Type name</th>
<th>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>Usually 32 bits</td>
<td>Usually IEEE-754 single precision floating point</td>
</tr>
<tr>
<td>double</td>
<td>Usually 64 bits</td>
<td>Usually IEEE-754 double precision floating point</td>
</tr>
<tr>
<td>long double</td>
<td>At least the size of double</td>
<td>?</td>
</tr>
</tbody>
</table>
sizeof

- To find out the size in memory of any data type, you can use the `sizeof` operator.
- `sizeof` gives the size in units of the size of `char`.
  - `sizeof(char)` is 1 by definition.
- The given value is of the unsigned integer type `size_t`.
Arrays

• An array is a **fixed-size** sequence of elements of the same type
• Array elements are always stored contiguously in memory
• There is no string type, C uses arrays of `char`
Arrays

• An array is a **fixed-size** sequence of elements of the same type
• Array elements are always stored contiguously in memory
• There is no string type, C uses arrays of char

**Example array definitions:**

```c
int arr1[10];
int arr2[10] = {1,2,3,4,5};
int arr3[] = {1,2,3,4,5};
char s[] = "Hello";
```
Arrays

- An array is a **fixed-size** sequence of elements of the same type
- Array elements are always stored contiguously in memory
- There is no string type, C uses arrays of `char`

**Example array definitions:**

```c
int arr1[10];
int arr2[10] = {1,2,3,4,5};
int arr3[] = {1,2,3,4,5};
char s[] = "Hello";
```
Arrays, cont.

• Every element in an array can be read and written to independently.
• Trying to read or write outside the bounds of an array *hopefully* crashes the program – **C performs no bounds checking**!
Arrays, cont.

- Every element in an array can be read and written to independently
- Trying to read or write outside the bounds of an array hopefully crashes the program – **C performs no bounds checking!**

**Example array uses:**

```c
int arr[] = {10,11,12,13,14};
int x = arr[0];  /* Sets x to 10 */
arr[5] = 42;     /* Anything happens! */
```
Multidimensional arrays

• You can create $n$-dimensional arrays for any $n \geq 1$
• In memory these look just like a single large array, but the compiler will calculate the indices for you

Example:

```c
int arr1[3][2] = {{0,1},{2,3},{4,5}};
/* arr2 is exactly like arr1 in memory */
int arr2[6] = {0,1,2,3,4,5};
arr1[2][0]; /* Evaluates to 4 */
```
Pointers (very briefly)

• A pointer is a memory address
• Pointers can be stored in variables of pointer type
• Using referencing (&) we can get a pointer to any variable
• Using dereferencing (*) we can get the value add the address pointed to

```c
int a = 5; /* a is an integer */
int *ptr; /* ptr is a pointer to an integer */
ptr = &a; /* The value of ptr is a's address */
*ptr += 2; /* a is now 7 */
```
Pointers (very briefly)

- A pointer is a memory address
- Pointers can be stored in variables of pointer type
- Using referencing (&) we can get a pointer to any variable
- Using dereferencing (*) we can get the value added the address pointed to

```c
int a = 5; /* a is an integer */
int *ptr; /* ptr is a pointer to an integer */
ptr = &a; /* The value of ptr is a's address */
*ptr += 2; /* a is now 7 */
```
Arithmetic expressions

- Basic expressions: +, −, *, /, %
  → work as expected

Short forms:

- \( n++ \) sets \( n = n + 1 \) and evaluates to old value of \( n \)
- \( ++n \) sets \( n = n + 1 \) and evaluates to new value of \( n \)
- \( n-- \) and \( --n \) similar
- \( n\times3 \) is equal to \( n = n \times 3 \) etc.
Boolean expressions

- There is no boolean type in C (well, there is in C99)
- Boolean expressions evaluate to an int
  \[ \theta \] is interpreted as \textit{false}
  \[ \text{everything else} \] is interpreted as \textit{true}
- E.g., \texttt{if(42)} and \texttt{if(-3)} will take the \texttt{if}-branch,
- \texttt{if(0)} will not
Boolean expressions

- **Comparisons:** `==, !=, <, >, <=, =>`
  - → work as expected
- **Conjunction:** `&&`
- **Disjunction:** `||`
- **Negation:** `!`
Boolean expressions

- Comparisons: ==, !=, <, >, =<, =>
  → work as expected
- Conjunction: &&
- Disjunction: ||
- Negation: !

**Warning:** Don't mix up = with ==, & with &&, or | with ||

A C compiler will happily let you write things like:

```c
if (a = 5) {
    // do something if a equals 5
}
```
if (a == 5) {
    // do something when a == 5
} else if (a > 0) {
    // do something when a > 0, but a != 0
} else {
    // do something when a <= 0
}
while loops

```java
int n = 4096;
while (n >= 1) {
    // do something
    n = n / 2;
}
```
for loops

int i;

for (i = 0; i < 10; i++) {
    // do something
}


for loops

```c
int i;

for (i = 0; i < 10; i++) {
    // do something
}
```
for loops

```c
int i;

for (i = 0; i < 10; i++) {
    // do something
}
```

```c
int i;

i = 0;
while (i < 10) {
    // do something
    i++;  
}
```
break and continue

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop
break and continue

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

```c
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
    printf("%c\n", s[i]);
}
```
break and continue

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

```c
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
    printf("%c
", s[i]);
}
```
break and continue

- The break statement immediately exits the innermost loop
- The continue statement immediately exits the current iteration of the innermost loop

```c
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
    if (s[i] == 'l') {
        break;
    }
    printf("%c\n", s[i]);
}
```
break and continue

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

```c
char s[] = "Hello World!"; int i;
for (i = 0; s[i] != 0; i++) {
    if (s[i] == 'l') {
        break;
    }
    printf("%c\n", s[i]);
}
```
break and continue

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

```c
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
    if (s[i] == 'l') {
        continue;
    }
    printf("%c\n", s[i]);
}
```
break and continue

• The `break` statement immediately exits the innermost loop

• The `continue` statement immediately exits the current iteration of the innermost loop

```c
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
    if (s[i] == 'l') {
        continue;
    }
    printf("%c\n", s[i]);
}
```
More...

- **do-while loops**
- **switch statements**
- **gotos**
Functions

- Functions are declared with a return type and types for all parameters
  → use `void` as return type if the function returns nothing
Functions

- Functions are declared with a return type and types for all parameters

  → use **void** as return type if the function returns nothing

```c
void print_n_times(char c, int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%c\n", c);
    }
}
```
Functions

- Functions are declared with a return type and types for all parameters

  → use `void` as return type if the function returns nothing

```c
void print_n_times(char c, int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%c\n", c);
    }
}

int is_ascii_lowercase(char c) {
    if (c >= 'a' && c <= 'z') {
        return 1;
    }
    return 0;
}
```
Functions, cont.

- Arguments to functions are pass-by-value
  → use pointers when you want pass-by-reference
Arguments to functions are pass-by-value
→ use pointers when you want pass-by-reference

```c
#include <stdio.h>

void add_two(int n) {
    n += 2;
}

int main() {
    int a = 10;
    add_two(a);
    printf("%d\n", a); /* Prints 10 */

    return 0;
}
```
Functions, cont.

- Arguments to functions are pass-by-value
  → use pointers when you want pass-by-reference

```c
#include <stdio.h>

void add_two(int *n) {
    *n += 2;
}

int main() {
    int a = 10;
    add_two(&a);
    printf("%d\n", a); /* Prints 12 */
    return 0;
}
```
printf and scanf

- Provides *formatted* input and output, respectively.
- Declared in `<stdio.h>` (use `#include <stdio.h>`)
- The first argument should be a *format string* which specifies how the remaining arguments are to be printed
printf and scanf

- Provides *formatted* input and output, respectively.
- Declared in `stdio.h` (use `#include <stdio.h>`)  
- The first argument should be a *format string* which specifies how the remaining arguments are to be printed

```c
int num = 37;
char ch = 'P';
double pi = 3.14159;
printf("The value of num is %d\n", num);
printf("%c follows %c in the alphabet\n", ch+1, ch);
printf("pi is approximately %f\n", pi);
```
printf and scanf

• Provides *formatted* input and output, respectively.
• Declared in `stdio.h` (use `#include <stdio.h>`)
• The first argument should be a *format string* which specifies how the remaining arguments are to be printed

```c
int num = 37;
char ch = 'P';
double pi = 3.14159;

printf("The value of num is %d\n", num);
printf("%c follows %c in the alphabet\n", ch+1, ch);
printf("pi is approximately %f\n", pi);
```

The value of num is 37  
Q follows P in the alphabet
pi is approximately 3.141590
printf and scanf

• Provides *formatted* input and output, respectively.
• Declared in *stdio.h* (use `#include <stdio.h>`)  
• The first argument should be a *format string* which specifies how the remaining arguments are to be printed

```c
int num = 37;
char ch = 'P';
double pi = 3.14159;

printf("The value of num is %d\n", num);
printf("%c follows %c in the alphabet\n", ch+1, ch);
printf("pi is approximately %f\n", pi);
```

The value of num is 37  
Q follows P in the alphabet    
pi is approximately 3.141590
# Some printf format identifiers

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Notes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>Prints an int in decimal</td>
<td>printf(&quot;%d is a number&quot;, 5); → 5 is a number</td>
</tr>
<tr>
<td>%o / %x / %X</td>
<td>Prints an int in octal / hexadecimal / HEXADECIMAL</td>
<td>printf(&quot;%o %x %X&quot;, 59, 59, 59); → 73 3b 3B</td>
</tr>
<tr>
<td>%.nf</td>
<td>Prints a float or double with $n$ digits after the point</td>
<td>printf(&quot;%.3f&quot;, 10.0 / 3); → 3.333</td>
</tr>
<tr>
<td>%c</td>
<td>Prints a char as a character</td>
<td>printf(&quot;%c %c&quot;, 'A', 66); → A B</td>
</tr>
<tr>
<td>%s</td>
<td>Prints a string</td>
<td>printf(&quot;I have %s cats&quot;, &quot;two&quot;); → I have two cats</td>
</tr>
</tbody>
</table>

There are many more options, check documentation or the web!
# Some `scanf` format identifiers

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Notes</th>
<th>Example</th>
</tr>
</thead>
</table>
| `%d`       | Reads an int in decimal | `int a; scanf("%d", &a);`
| `%f / %lf` | Reads a float / double | `double a; printf("%lf", &a);`
| `%c`       | Reads a char as a character | `char a; scanf("%c", &a);`
| `%s`       | Reads a string, will automatically append a null char | `char a[10]; scanf("%s", a);`
# Some `scanf` format identifiers

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Notes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>Reads an int in decimal</td>
<td>int a;\nscanf(&quot;%d&quot;, &amp;a);</td>
</tr>
<tr>
<td>%f / %lf</td>
<td>Reads a float / double</td>
<td>double a;\nprintf(&quot;%lf&quot;, &amp;a);</td>
</tr>
<tr>
<td>%c</td>
<td>Reads a char as a character</td>
<td>char a;\nscanf(&quot;%c&quot;, &amp;a);</td>
</tr>
<tr>
<td>%s</td>
<td>Reads a string, will automatically append a null char</td>
<td>char a[10];\nscanf(&quot;%s&quot;, a);</td>
</tr>
</tbody>
</table>

**Warning:** `printf` and `scanf` (and many more C functions) are unsecure when used naively. A bit more on this next time.
Using **ssh** to our servers

- You can use Secure Shell (**ssh**) to connect to the department's UNIX servers from your own computer.
- Some instructions on [https://www.it.uu.se/datordrift/faq/ssh](https://www.it.uu.se/datordrift/faq/ssh)
- List of available Linux hosts on [https://www.it.uu.se/datordrift/maskinpark/linux](https://www.it.uu.se/datordrift/maskinpark/linux)
- List of available Solaris hosts on [https://www.it.uu.se/datordrift/faq/unixinloggning](https://www.it.uu.se/datordrift/faq/unixinloggning)