A whirlwind tour of C

Echtzeitsysteme WS 2014/2015

heise@in.tum.de
What you should already know

- The basic datatypes (e.g. int, float)
- Basic control flow (e.g. if/else, for, while)
- What functions are
- What classes and objects are
- How to use a compiler
Hello C

Code

```c
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

Build/Output

```
$ gcc main.c
$ .a.out
Hello World
$
```

Functions

- Reuse and structure code
- Parameters and return value
Functions

Code

```c
#include <stdio.h>

int fac(int x)
{
    return (x <= 1) ? 1 : x * fac(x - 1);
}

int main()
{
    printf("%d\n", fac(5));
    return 0;
}
```

Output

```
./a.out
120
```

$
Functions

- Return type
- Function name
- Argument 0 type
- Argument 0 name

```
int fac(int x)
{
    return (x <= 1) ? 1 : x * fac(x - 1);
}
```

arbitrary number of arguments possible

```
type function(type0 arg0, type1 arg1, ..., typeN argN)
{
    ...
}
```
Arrays

Declaration

- **Type**: 
- **Name**: 
- **Dimension**: 

```c

// type name[ dimension ];

// type name[ dimension1 ][ dimension2 ];

...  
```

Initialization

```c

int array[ 4 ];
array[ 0 ] = 0;
array[ 1 ] = 5;
array[ 2 ] = 8;
array[ 3 ] = 3;

int array[ 4 ] = { 3, 7, 9, 2 };  
int array[] = { 3, 7, 9, 2 };  
```
Arrays

- Special initialization for char arrays / strings
- The following char arrays are equivalent

```c
char str[] = "String";
char str2[] = {'S', 't', 'r', 'i', 'n', 'g', '\0'};
```
Pointers

- A variable name refers to a particular location in memory and stores a value there.
- If you refer to the variable by name then:
  - the memory address is looked up
  - the value at the address is retrieved or set
- C allows us to perform these steps independently:
  - `&x` evaluates to the address of `x` in memory
  - `*( &x )` dereferences the address of `x` and retrieves the value of `x`
  - `*( &x )` is the same thing as `x`
Pointers

Code

```cpp
#include <iostream>

int main()
{
    int x;
    int* p = &x;

    x = 10;
    printf("%d\n", *p);

    *p = 5;
    printf("%d\n", x);

    return 0;
}
```

Output

```
$ ./a.out
10
5
$ 
```
Pointers

... 0x123 0x124 0x125 0x126 ...

x     p
Pointers

Example

```c
#include <stdio.h>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while( *cptr != '\0' ) {
        len++;
        cptr++;
    }
    printf("%d\n", len);
}
```
# Pointers

## Example

```c
#include <stdio.h>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while( *cptr != '\0' ) {
        len++;
        cptr++;
    }
    printf("%d\n", len);
}
```
# Pointers

- Example

```c
#include <stdio.h>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while( *cptr != '\0' ) {
        len++;
        cptr++;
    }
    printf("%d\n", len);
}
```
#include <iostream>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while ( *cptr != '\0' ) {
        len++;
        cptr++;
    }

    std::cout << len << std::endl;
}

Example

#include <stdio.h>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while ( *cptr != '\0' ) {
        len++;
        cptr++;
    }

    printf("%d\n", len);
}
Pointers

Example

```c
#include <stdio.h>

int main()
{
    char* cptr = "bla";
    int len = 0;

    while( *cptr != '\0' ) {
        len++;
        cptr++;
    }

    printf("%d\n", len);
}
```
# Pointers

- **Pointers and arrays**

```c
int array[ 5 ];
...
    array  ≡  &array[ 0 ]
    *array  ≡  array[ 0 ]
*( array + 1 )  ≡  array[ 1 ]  ≡  1[ array ]
...
```

- **Arithmetic pointer operations modify the address by sizeof( type ) bytes**

```c
#include <stdio.h>

int main()
{
    char*  x = 0x0;
    float* y = 0x0;

    printf("%p\n", ( x + 1 ));
    printf("%p\n", ( y + 1 ));
}
```

$ ./a.out
0x1
0x4
$
Pointers

`const int* ptr`

- Declares a changeable pointer to a constant integer
- value cannot be changed
- pointer can be changed to point to a different constant integer

`int* const ptr`

- Declares a constant pointer to a changeable integer
- value can be changed
- pointer cannot be changed to point to a different integer

`const int* const ptr`

- Neither the value nor the address can be changed
No guarantees that a pointer points to a valid address

```c
int* ptr = 0xdeadbeef;
int* ptr = 0x0;

int* function()
{
    int x;
    return &x;
}

int* p = malloc( sizeof( int ) * 5 );
free( p );
```
Memory management

- Dynamic memory allocation possible using malloc/free

```c
... int* x = malloc( sizeof( int ));
... int* y = malloc( sizeof( int ) * 10 );
... float** z;
    z = malloc( sizeof( float* ) * 10 );
    z[ 0 ] = malloc( sizeof( float ) * 3 );
    z[ 1 ] = malloc( sizeof( float ) * 5 );
... free( x );
free( y );
free( z[ 0 ] );
free( z[ 1 ] );
free( z );
...```

- If allocated memory is not correctly freed using ‘free’ it is wasted and cannot be reused

- Pointers to freed memory still contain the address
Structures

- Definition

```c
struct name {
    type member1;
    ...
    type memberN;
};
```

- Allows the definition of new datatypes
- Structures allow to collect variables into a new datatype
- Essential for structuring C code
- Enhanced reusability and readability
Structures

Example

```c
struct vector2 {
    float x;
    float y;
};

typedef struct {
    float x;
    float y;
    float z;
} vector3;

int main()
{
    struct vector2 myvec;
    vector3 myvec3;

    myvec.x = 0.0f;
    myvec.y = 0.0f;
    myvec3.x = 0.0f;
    myvec3.y = 0.0f;
    myvec3.z = 0.0f;
}
```

Structures are often treated as the data part of objects in C and functions operate on the structs like methods on the members.
Questions?