C language basics
(with tiny bits of C++)

Lubomír Bulej
KDSS MFF UK
Basic features

Procedural, imperative, structured (mostly)
● Code organized in functions that can return a value
● Explicit control flow, structured programming

Statically typed
● All variables/parameters/return values must have a type
● Incompatible types cannot be assigned

Explicit memory management (heap)
● Allocated heap memory must be deallocated manually
● Difficult & error prone!

Conceptually close to machine level code
● Maps efficiently to machine instructions
● Used for operating & embedded systems, HPC
● Should NOT be used for (extensive) string manipulation!
Constant literals

Integer numbers
- Decimal: 123, -18
- Hexadecimal: \x7A

Floating point
-1.234e-5

Char
'a'

Boolean (C++)
true, false

String
"Hello!"

Character escape sequences
- \n ... Line Feed (LF)
- \r ... Carriage Return (CR)
- \t ... Tab (character 9)
- \\ ... \n
- ' ... '
- " ... "
- \xAB ... character 0xAB
- \0 ... Zero character (NUL)
Basic types

**Integer types**
- Base: `char`, `int`
- Modifiers: `short`, `long`, `signed`, `unsigned`

**Floating point types**
- `float`, `double`

**Other types**
- `void`, `bool` (C++)

**Type definitions**
- `size_t`, `ssize_t`
- `off_t`, ...

**Precise sizes**
- `uint8_t`, `int32_t`, ...

**Strings?**
- A bit special… Wait until arrays and pointers.

**Implicit conversion**
- Towards higher rank (higher precision = higher rank)
Named value stored in memory

- Must be declared before first use
  - Variable type followed by variable name
    ```
    int i;
    ```
- Always strive to initialize variable at declaration
  - Helps keep track of how a variable got its value
    ```
    unsigned int u = 42;
    ```

Variable scope

- Determines where a variable can be accessed
  - *Local variables* only accessible within the block it was declared in (function, block in curly braces)
  - Function parameters are also local variables
  - *Global variables* accessible anywhere after declaration
Variables (2)

Storage class determines lifetime

- **Automatic** variables: lifetime starts when execution enters their scope and ends when execution leaves their scope
  - Default, no need to be specified explicitly
- **Static** variables: lifetime starts with declaration and lasts for the lifetime of a program (special keyword needed)

```c
static int s = 0;
```

Auto variables (C++)

- Variable type inferred from the initialization expression

```c
auto a = 3;
```
**Run-time: like variables**

```c
const int j = 33;
```

**Compile-time only**

- Does not exist in memory
- Compiler understands it (C++)

```c
constexpr int C = 13;
```

**Compile-time macro**

- Handled by pre-processor
- Appears as a literal to the compiler

```c
#define C 13
```

**Const**

- Immutable, accessible at runtime (it exists in memory), immutable
Statements

Expression statement

- Variable assignments considered an expression

```
expr;
```

Compound statement (block)

```
{} {}
```

Conditional statement

```
if (expr) stmt
if (expr) stmt else stmt
```

Return form a function

```
return expr;
```
switch (expr) {
    case 0:
        // Code for value 0
        break;
    case 1:
        // Code for value 1
        break;
    case 2:
    case 3:
        // Common code for values 2 and 3
        break;
    default:
        // Code for all other values
        break;
}

Statements - switch
Statements - iteration

While loop

while (expr) stmt

Do-while loop

do stmt while (expr);

For loop

for (expr_init; expr_test; expr_post) stmt

Jumps

break;
continue;
Operators

Arithmetic
+ , - , * , / , % (modulo), ++ (increment), -- (decrement)

Comparison
< , <= , > , >= , == (equal) , != (not equal)

Bitwise
~ (bit inversion) , & , | , ^ (xor) , << (shift left logical) , >>

Logical
&& , || , ! (not)

Pointers
& (address of) , * (pointer dereference) , -> (struct dereference)

Assignment (with arithmetic and bitwise operations)
= , += , -= , *= , /= , %= , &= , |= , ^= , <<= , >>=
Arrays

Sequence of elements of the same type

- Laid out in a contiguous chunk of memory
- Each element identified by a zero-based index
- Correct alignment, row-major order

```c
int u[4];
int p[] = { 1, 2, 3 };
int a[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a[0][0]</th>
<th>a[0][1]</th>
<th>a[0][2]</th>
<th>a[1][0]</th>
<th>a[1][1]</th>
<th>a[1][2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Strings

Sequence of characters ending with zero (NUL) character

- Represented as array of char elements
  - Zero (NUL) character added automatically
- Interchangeable with pointer to character
  - Pointers coming up next...
- Array of characters not necessarily a string!

```c
char str[] = "Hello!";
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'H'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>'!'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

```c
char chars[] = { 'H', 'e', 'l', 'l', 'o', '!' };
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'H'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>'!'</td>
</tr>
</tbody>
</table>
Structures

Sequence of elements of the same type
- Collection of fields (members)
- Alignment (produces padding)
  - Typically fields aligned to their size, aggregates (structures) aligned to largest field alignment

```
struct point2d { int x; int y; }

struct data {
  char c;
  double d;
  int i;
}
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 B</td>
<td>c</td>
</tr>
<tr>
<td>8 B</td>
<td>d</td>
</tr>
<tr>
<td>16 B</td>
<td>i</td>
</tr>
</tbody>
</table>
Structures

Sequence of elements of the same type

- Collection of fields (members)
- Alignment (produces padding)
  - Typically fields aligned to their size, aggregates (structures) aligned to largest field alignment

```c
struct point2d { int x; int y; }

struct data {
    char c;
    int i;
    double d;
}
```

```
<table>
<thead>
<tr>
<th>Offset</th>
<th>c</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>d</td>
</tr>
</tbody>
</table>
```
Enums

Basically an int type

- Values assigned automatically

```c
enum color_t { COLOR_RED, COLOR_GREEN, COLOR_BLUE };
```

- Values can be forced if necessary (and selectively)

```c
enum color_t {
    COLOR_RED = 0, COLOR_GREEN, COLOR_BLUE = 2
};
```

- Good practice is to add "support" for iteration

```c
enum color_t {
    COLOR_FIRST = 0,
    COLOR_RED = COLOR_FIRST,
    COLOR_GREEN = 1,
    COLOR_BLUE = 2,
    COLOR_LAST = COLOR_BLUE
};
```
Strange keywords/directives starting with #

- Handled by preprocessor (mostly)
- Produces text at source code level (mostly)
  - Used for parametrization at source code level (conditional compilation)

```
#include <module.h> ... import relative to system defined path
#include "module.h" ... import relative to this file

#define MACRO_NAME macro literal value

#ifdef MACRO_NAME
#endif
```
Pointers

Abstraction of a location (address) in memory
● Pointer = variable holding an address
  ○ Operations capture address manipulations
● Pointers are typed
  ○ Pointing at a particular data type
  ○ Different pointer types are incompatible
● Pointer-related operators
  & ... Take an address of a variable (produces pointer value)
  * ... Dereference (follow) the pointer to the value

```
int v = 8;
int * pv = &v;
*pv = 4;
```
Array variable = pointer to first element

- Applies to strings as well
  - String = array of `char` with extra NUL character

```c
char str1[] = "Hello!";
char * str2 = "Hello!";
```

```c
int vals1[] = { 1, 2, 3 };
int * vals2 = { 1, 2, 3 };
```
The size of things

The sizeof operator

- Returns the size of a type or variable in **bytes**
  ```c
  sizeof(int)
  sizeof(struct data)
  ```
- Also works for fixed-size array variables
  ```c
  int u[4];
  sizeof(u) == 4 * sizeof(int)
  char s[] = "Hello";
  sizeof(s) == (5 + 1) * sizeof(char)
  ```
- Beware in the case of pointer types
  - The compiler only knows the size of the pointer variable, or the data type it points at
  ```c
  const char * s_ptr = "World";
  sizeof(s_ptr) == sizeof(char *)
  sizeof(*s_ptr) == sizeof(char)
  ```
Arguments in C always passed by value
  ● Array variables are in fact pointers (passed by value)

Output parameters use a pointer

```c
struct point2d {
  int x;
  int y;
};

void copy_point(point2d in, point2d * out) {
  out->x = in.x;
  out->y = in.y;
}
```
**References**

**Alias to a variable**

- Must be initialized, cannot be reassigned
  - A bit safer than pointers
- Consider it a fixed pointer
  - Does not support pointer arithmetics
  - Bit more complicated, but let's leave it at that...
- Below: note the absence of `&` applied to the variable `v`
  - Variable `rv` is an alias to variable `v`

```c
int v = 8;
int & rv = v;
rv = 4;
```

![Address diagram](image-url)
Arguments in C++ passed by value or reference

- Recall: reference must be initialized

Output parameters use a pointer

```c
struct point2d {
    int x;
    int y;
};

void copy_point(point2d in, point2d & out) {
    out->x = in.x;
    out->y = in.y;
}
```
Advanced pointer example: linked list

Definition

```c
struct node {
    int value;
    node * next;
};
node * list;
```

Logical view

- “Chain” of nodes
- Variable `list` is a pointer to the first `node`

Physical layout

- One of "infinitely" many possible...

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>value = 1</td>
</tr>
<tr>
<td>0x104</td>
<td>next = 0x400</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x200</td>
<td>list = 0x100</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x300</td>
<td>value = 3</td>
</tr>
<tr>
<td>0x304</td>
<td>next = 0x0 (NULL)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x400</td>
<td>value = 2</td>
</tr>
<tr>
<td>0x404</td>
<td>next = 0x300</td>
</tr>
</tbody>
</table>