Computer Systems

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
  `0x7A`

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)
- `\\` … \
- `\'` … '
- `\"` … "
- `\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)

Variables

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)

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

Auto variables (C++)

- Variable type inferred from the initialization expression

```
auto a = 3;
```

Constants

Run-time: like variables

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

Compile-time only

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

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

Compile-time macro

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

```
#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;
```

Statements - switch

```
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 - 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

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

u[0]	u[1]	u[2]	u[3]
0	0	0	0

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
1	2	3	4	5	6

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!

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

str[0]	str[1]	str[2]	str[3]	str[4]	str[5]	str[6]
'H'	'e'	'l'	'l'	'o'	'!'	'\0'

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

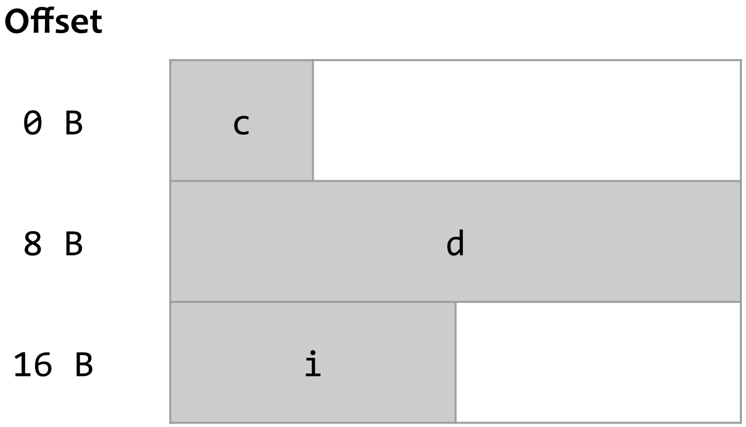
chars[0]	chars[1]	chars[2]	chars[3]	chars[4]	chars[5]
'H'	'e'	'l'	'l'	'o'	'!'

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;
}
```

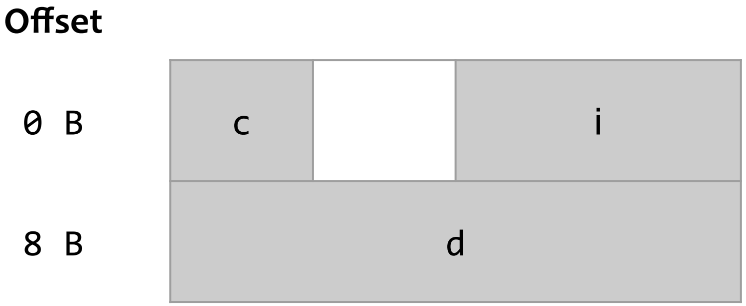


Structures

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;
   int i;
   double d;
}
```



Enums

Basically an `int` type

- Values assigned automatically

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

- Values can be forced if necessary (and selectively)

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

- Good practice is to add "support" for iteration

```
enum color_t {
   COLOR_FIRST = 0,
   COLOR_RED = COLOR_FIRST,
   COLOR_GREEN = 1,
   COLOR_BLUE = 2,
   COLOR_LAST = COLOR_BLUE
};
```

Preprocessor

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

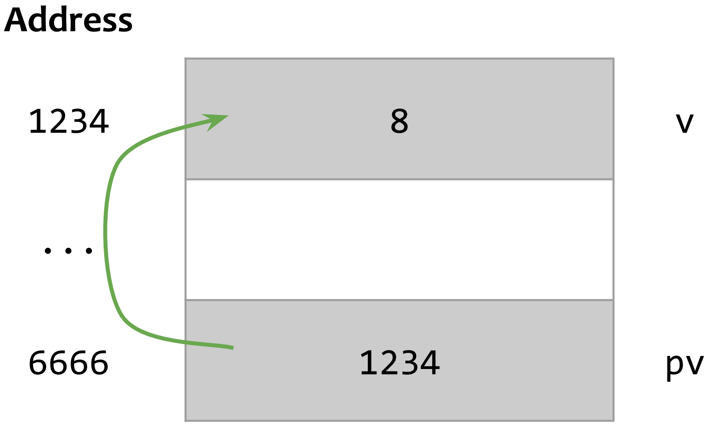
Abstraction of a location (address) in memory

- 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;
```



Address

1234 | 8 | v

. . .

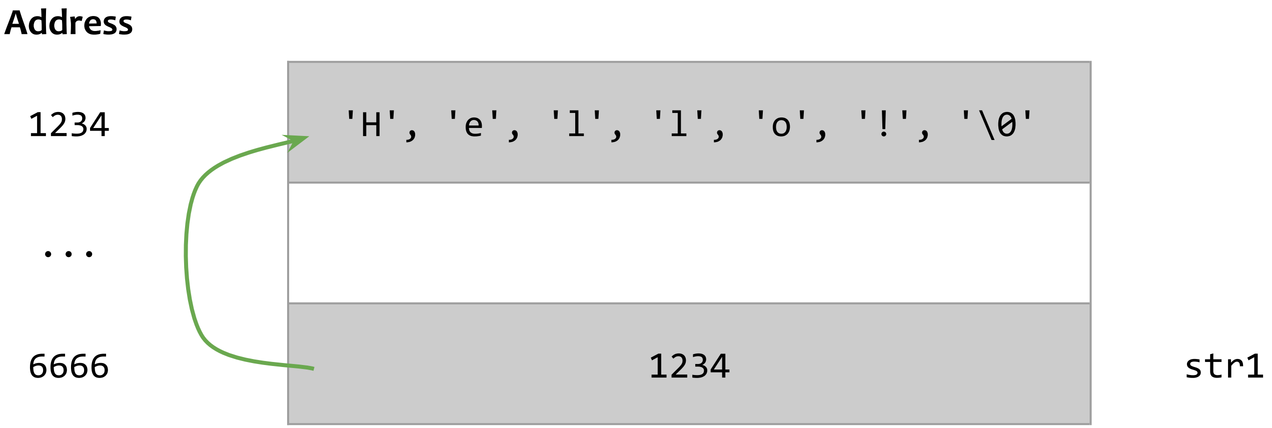
6666 | 1234 | pv

String and array variables: pointers

Array variable = pointer to first element

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

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

int vals1[] = { 1, 2, 3 };
int * vals2 = { 1, 2, 3 };
```



Address

1234 'H', 'e', 'l', 'l', 'o', '!', '\0'

...

6666 1234 str1

The size of things

The `sizeof` operator

- Returns the size of a type or variable in ***bytes***

  ```
  sizeof(int)
  sizeof(struct data)
  ```

- Also works for fixed-size array variables

  ```
  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

  ```
  const char * s_ptr = "World";
  sizeof(s_ptr) == sizeof(char *)
  sizeof(*s_ptr) == sizeof(char)
  ```

Functions, argument passing — C

Arguments in C always passed by value

- Array variables are in fact pointers (passed by value)

Output parameters use a pointer

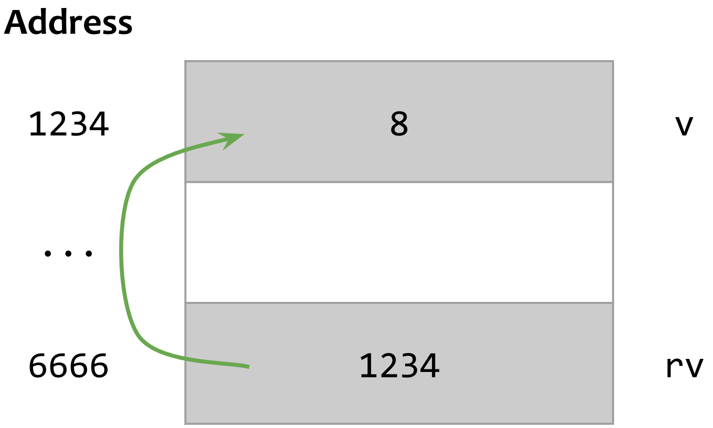
```
struct point2d {
   int x;
   int y;
};

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**

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



Functions, argument passing — C++

Arguments in C++ passed by value or reference

- Recall: reference must be initialized

Output parameters use a pointer

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

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

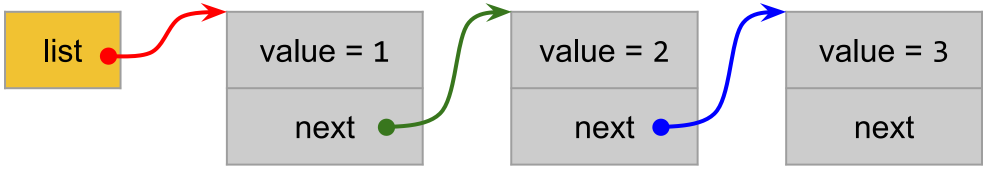
Advanced pointer example: linked list

Definition

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

node * list;
```

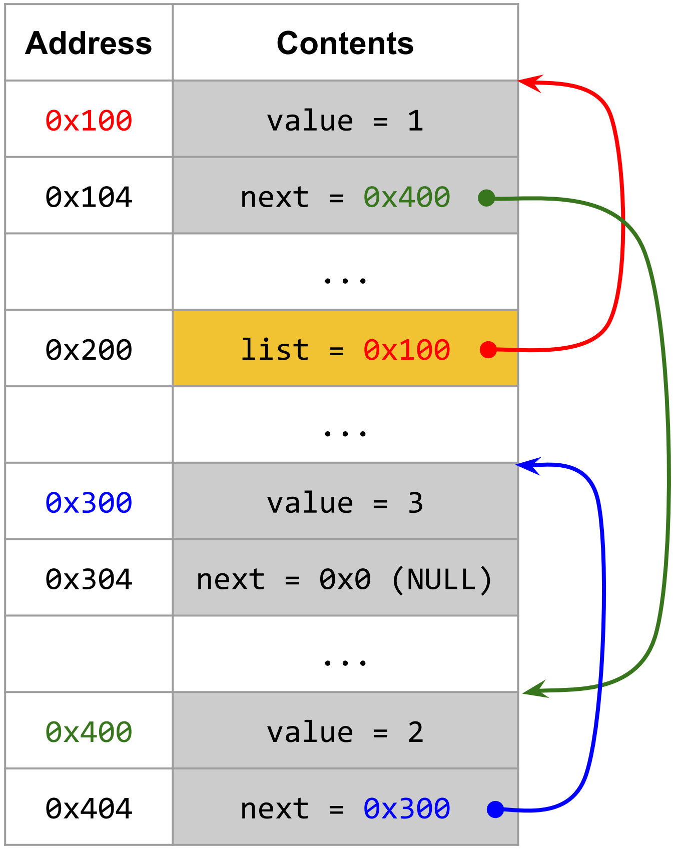
Logical view

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



Physical layout

- One of "infinitely" many possible...



Address	Contents
0x100	value = 1
0x104	next = 0x400
	...
0x200	list = 0x100
	...
0x300	value = 3
0x304	next = 0x0 (NULL)
	...
0x400	value = 2
0x404	next = 0x300