# C/C++ language

NSWI170 Computer Systems

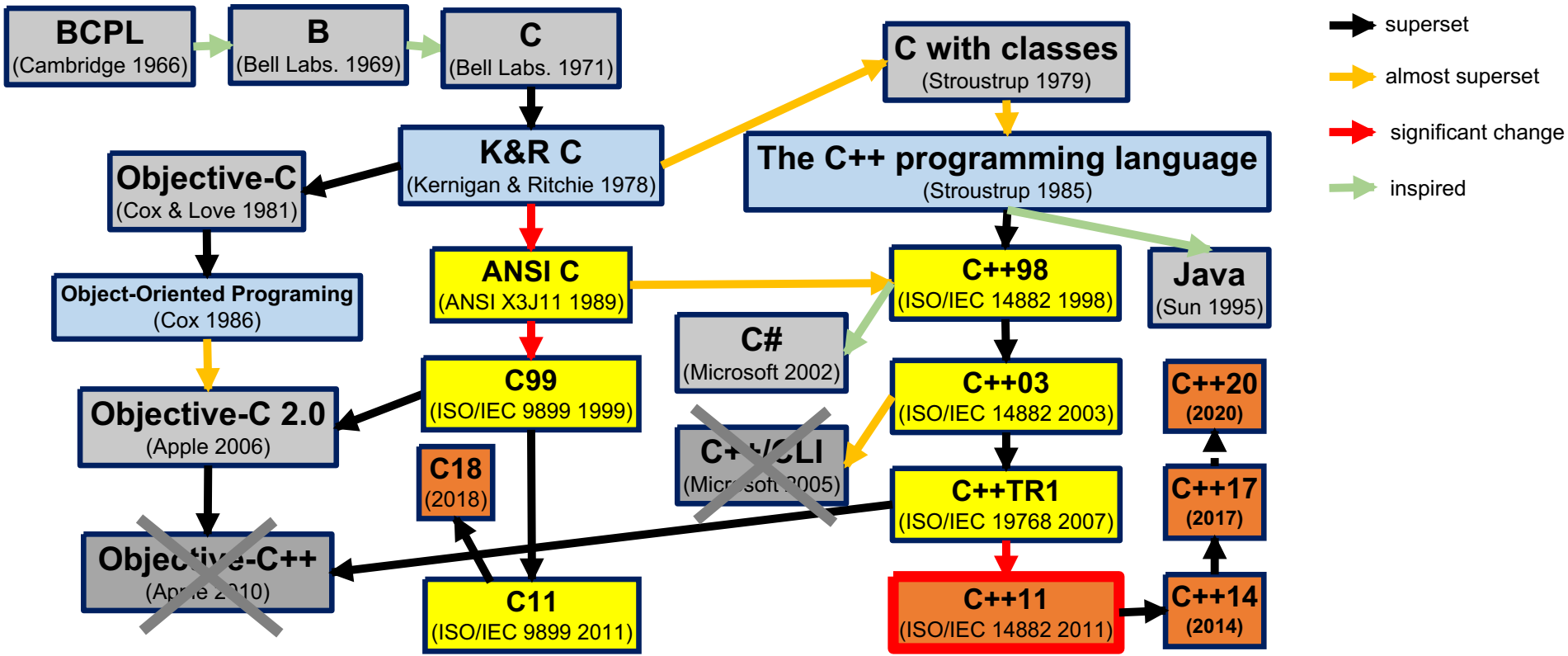
Jakub Yaghob, Martin Kruliš

# Features

- Procedural, structured, imperative programming language
  - Code organized in functions (that can return a value)
  - Explicit control flow (structure)
- Static type system
  - All variables, parameters, return values must have a type
  - Compiler checks type compatibility (e.g., in assignments)
- C/C++ constructs map efficiently to machine instructions
  - Good for operating systems, HPC, embedded systems, …
  - Explicit memory management (heap)
- Case sensitive, ignore all whitespaces (unlike Python)

Unlike Python, which has dynamic typing

# History


BCPL
(Cambridge 1966)
B
(Bell Labs. 1969)
C
(Bell Labs. 1971)
K&R C
(Kernigan & Ritchie 1978)
Objective-C
(Cox & Love 1981)
Object-Oriented Programing
(Cox 1986)
Objective-C 2.0
(Apple 2006)
Objective-C++
(Apple 2010)
ANSI C
(ANSI X3J11 1989)
C99
(ISO/IEC 9899 1999)
C18
(2018)
C11
(ISO/IEC 9899 2011)
C with classes
(Stroustrup 1979)
The C++ programming language
(Stroustrup 1985)
C++98
(ISO/IEC 14882 1998)
C++03
(ISO/IEC 14882 2003)
C++TR1
(ISO/IEC 19768 2007)
C++11
(ISO/IEC 14882 2011)
C++14
(2014)
C++17
(2017)
C++20
(2020)
C#
(Microsoft 2002)
C++/CLI
(Microsoft 2005)
Java
(Sun 1995)
superset
almost superset
significant change
inspired

# Hello World Example

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World!\n");
    return 0;
}
```

int bude většinou odpovídat ideální velikosti procesoru
- takže může být třeba jen 8-bit

# Constant Literals

- Integer numbers
  - Decimal number **123**, **-18**
  - Hexadecimal **0xc001bea3**
  - Binary **0b11001001**
  - Octal (0-prefixed) **0123**, **017**
- Floating point numbers
  **0.42**, **-1.234e-5**
- Boolean
  **true**, **false**

Do not use!

- String
  **"foo  bar"**
- Char
  **'a'**
- Escape sequence
  - **\n** – LF (line feed)
  - **\r** – CR (carriage return)
  - **\t** – TAB
  - **** – (one) backslash
  - **\'** – single quote
  - **\"** – double quote
  - **\xab** – char with value 0xab

# Basic types

- Integer types
  - Base
    **char**, **int**
  - Modifiers
    **short**, **long**, **long long**
    **signed**, **unsigned**
  - Auxiliary
    **size_t**
- Floating point types
  **float**, **double**

- Other types
  **void**, **bool**
- Implicit conversions
  - Quite complex rule
  - No data loss – no problem
    **int** -> **long**
  - Possible data loss – **warning**
    **double** -> **float**
  - Operands are converted to common type
    - Conversion rank

Bad!

# Variables

- Named values stored in memory
  - Must be declared before first use (it is a good idea to initialize them as well)

```
int i;   double x = 0.0;
```

    - **auto** keyword can be used instead of type (the type is inferred)

Handle with care!

- Variable scope
  - Determines where the value is stored and how (in code) it can be accessed
  - *Local variables* – exist within one block { } (e.g., in one function)
    - Function arguments are also local variables
  - *Global variables*
  - *Static local variables*

```
void foo() { static int counter = 0; … }
```

Like global variables, but better encapsulation

# Statements

- Compound statement (block)

```
{ }
```

- Expression statement

```
expr ;
```

- If statement

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

- Return from a function

```
return expr;
```

Indentation bears no meaning…. However, it makes code more readable (do it)!

# Statements

```
switch (expr) {
case 0:
        // something
        break;
case 1:
        // something else
        break;
case 2:
case 3:
        // common code for 2 and 3
        break;
default:
        // do something else otherwise
        break;
}
```

Switch may be better than many if-elses, but heavy branching is **bad**!

When you are trying to use switch-case, re-think your code again, maybe there is better way how to structure it…

# Statements

- Conditional loop

  **while** ( *expr* ) *stmt*

- Conditional loop with at least one iteration

  **do** *stmt* **while** ( *expr* );

- Regular loop

  **for** ( *einit*; *etest*; *einc* ) *stmt*

- Additional loop control statements

  **break**;

  **continue**;

# Expression Operators

- Arithmetic
  **+, - , * , / , %**
  - Beware **//**

  **++, --**
- Comparison
  **<, <=, >, >=, ==, !=**
- Bitwise
  **~, &, | , ^, <<, >>**
- Logical
  **&&, || , !**

- Pointers
  **&, ***
- Assignment
  **=, +=, -= , *= , /= , %=, &=, |= , ^=**
- Variable/type size in bytes
  **sizeof**
- Ternary expr (like if-else)
  **test ? e1 : e2**

# Arrays

- Sequence of elements of same base type
  - Occupying continuous chunk of memory
  - Zero based index, fixed size given at compilation
  - Correct alignment, row-major order

We can create dynamically allocated arrays (but it will not be covered in this course)

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

sizeof(int)

| 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 (array) of characters ended with zero (NUL) character
  - NUL character is added automatically for **"string"** literals
  - Typically represented as pointer to first character (pointers coming up soon…)
  - Array of characters is not necessarily a string!

```
char str[] = "Hippo";
```

| str[0] | str[1] | str[2] | str[3] | str[4] | str[5] |
|---|---|---|---|---|---|
| 'H' | 'i' | 'p' | 'p' | 'o' | '\0' |

```
char chars[] = { 'H', 'i', 'p', 'p', 'o' };
```

| str[0] | str[1] | str[2] | str[3] | str[4] |
|---|---|---|---|---|
| 'H' | 'i' | 'p' | 'p' | 'o' |

# Structures

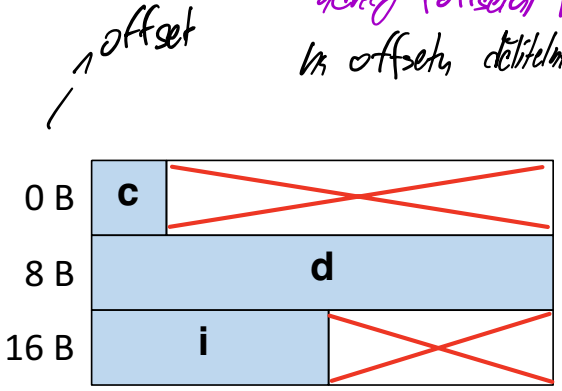
Struktura pojmenuje na data
Třídy pojmenují na stav

- Collection of fields (members)
  - Logically bound together
  - Inner alignment (padding), outer alignment (padding)

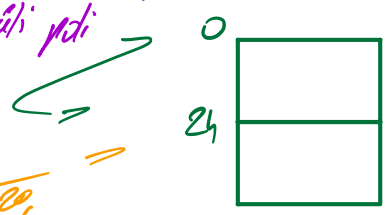
```
struct point2d {
   int x, y;
};

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

→ „leží na adrese, která je násobkem její velikosti"

Interní zarovnání: že umístění struktury respektuje dělitelnost adresy (offsetu) velikostí. Takže int (4B) musí být na offsetu dělitelném 4, double (8B) na offsetu dělitelném 8.




Vnější zarovnání: Musí být zarovnání na násobek největší položky, to určuje při




Kdyby to bylo 20, při dotazu na d bych se dostal na 28 a to není násobek 8.

# Constants and Enums

- Basic constants
  - Basically immutable variables

```
const int the_answer = 42;
```

- Compile-time constants (C++)
  - Do not exist in memory

```
constexpr int the_answer = 42;
```

Prefer compile-time when possible

- Enumerations
  - Special type with predefined set of allowed values
  - Compiles to int

```
class enum state_t { WAITING, RUNNING, STOPPED };
state_t s = state_t::WAITING;
```

# Preprocessor

- Preprocessor directives
  - Text-based replacements handled as the first step of compilation

```
#include <algorithm>
#include "my_module.h"

#define N 1000

#define ALLOW_DEBUG
#ifdef ALLOW_DEBUG
    printf("This code was executed\n");
#endif
```

Includes are used all the time, remember!

Avoid if you can (e.g., use **constexpr**)!

Good to know, but quite advanced (for special situations only)

# Application Entry-point

- Function named **main**
  - Return value is an exit code
    - Without return 0 exit code assumed
  - Note that all statements must be in some function or method
    - Unlike Python where main-level code is executed
- Simplistic version

```
int main() { ... }
```

- Advanced version

počet argumentů — pole ukazatelů

```
int main( int argc, char *argv[]) { ... }
```

Command line arguments as array of strings

# Pointers

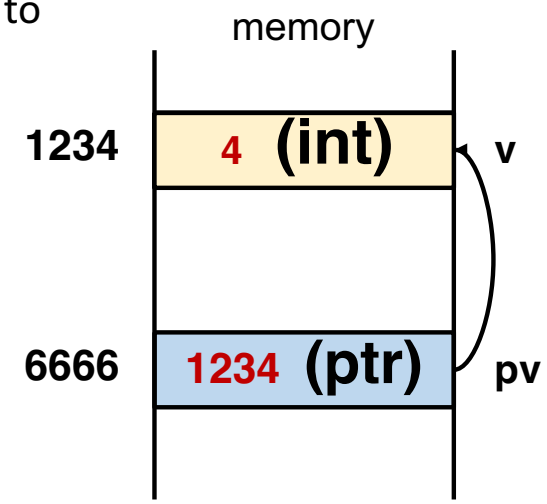
- Memory address abstraction
  - Pointer = number that holds an address (index of starting byte in memory)
  - Pointers are typed
    - What data type we can expect in memory where it points to
    - Pointers of different types are incompatible

```
int v = 8;
int *pv = &v;
*pv = 4;
```

„Vezmi adresu z proměnné"

ukazatel na int

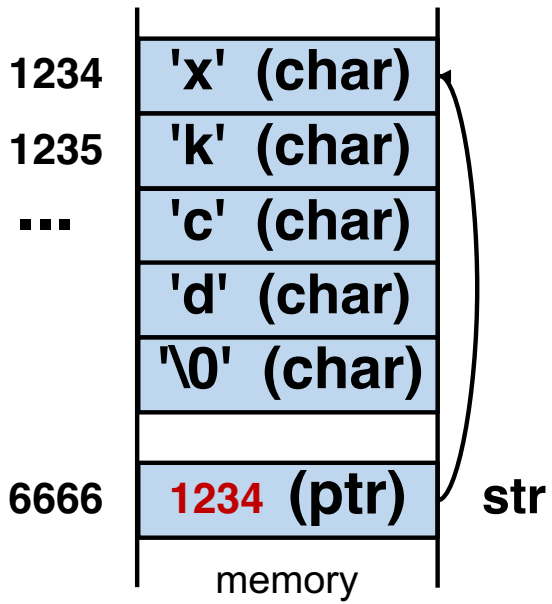
přístup na hodnotu zadané adresy – tedy na adresu 1234 zapiš 4.



# Pointers

- Array variable = pointer to the first element
  - Applies to strings as well
    - String = array of chars with extra NUL at the end

```
int  vals[]  = { 42, 54, 18 };
int  *vals = { 42, 54, 18 };

char  str[]  = "xkcd";
char  *str = "xkcd";
```



# Pointers

- Pointer address can be also computed by arithmetic operations

```
char *str = "Hippo";

for (int i = 0; i < 5; ++i) {
    doSomethingWithChar(str[i]);
}

for (int i = 0; *(str + i) != '\0'; ++i) {/* ... */}

while (*str) {
    // ...
    ++str;
}
```

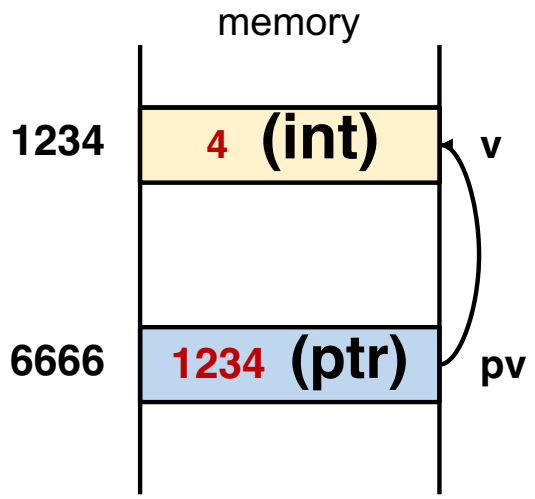
# References

- Reference (C++) — Referenci stejně jako pointer, jen ho nemůžu měnit
  - Fixed pointer (address not re-assignable)

```
int  v = 8;

int  &pv = v;

pv = 4;
```

už nemusím psát „*"!

Pointer se vždy nemění



# Functions

- Parameters in C always passed by value
- Output parameters must use pointers

```
struct point2d {
    float x, y;
};

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

Tohle se celé kopíruje -> drahé

-> tady se kopíruje jen pointer -> levné

"." přístup do lokální proměnné

"->" přístup přes pointer

# Functions

- Parameters in C++ passed by value or by reference
- Use references for output parameters

```
struct point2d {
    float x, y;
};

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

# Know The Size of Things

- The sizeof operator

```
sizeof( int )

struct data_t { ... };
sizeof(data_t)
```

- Also works for fixed-size arrays

```
int u[4];
sizeof(u) == 4 * sizeof( int )

char s[] = "Hello";
sizeof(s) == (5+1) * sizeof( char)
```

- Beware the size of pointers
  - The compiler only knows the size of the pointer itself (not the contents)

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

# C++ Classes

- C++ classes (very briefly)
  - Similar to structures
    - But combining data members with methods (embedded functions)
  - Encapsulation
    - Data holding a state and functions that can manipulate the state
  - Important concept
    - Will be covered in programming lectures in 2nd year…

```
class C {
public:
   C() : sum(0) {}
   void add(int a) { sum += a; }
   int get_sum() const {
      return sum;
   }
private:
   int sum;
};
```

# Discussion