

PTfS-CAM

(Short) Introduction to the C programming language

Jan Eitzinger



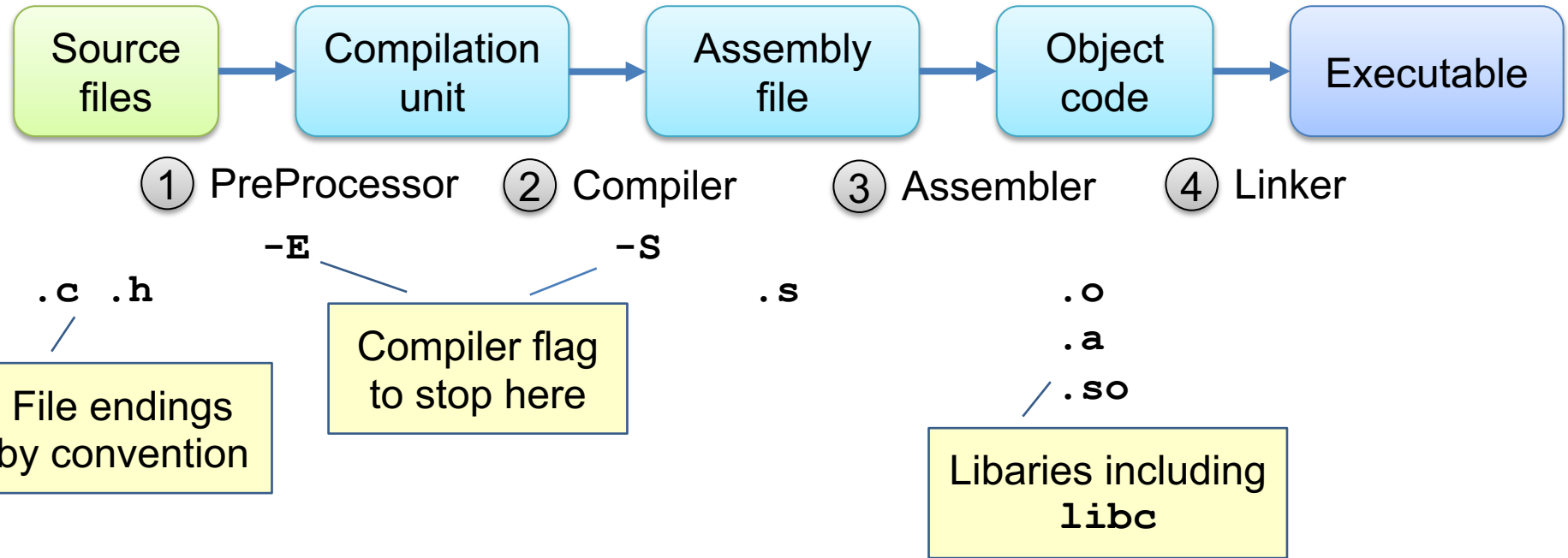
History and Background

- **Procedural** structured programming language
 - **Static type system** and lexical variable scope
 - **ANSI** and **ISO standard** (last stable release C17)
 - Many other programming languages use a **C-like syntax**
-
- Developed in **1972** at **Bell labs** as part of the **UNIX OS**
 - **Right level of abstraction** for **low level access** to underlying **machine architecture** while still providing support for portable **complex software**
 - Still among the **most popular** programming languages (**TIOBE** 1. place)
 - **Major implementations:** GCC, LLVM Clang, Intel ICC, MS Visual C++



Getting from source code to executable

- **Multiple steps required to get from source to executable** (usually wrapped by `cc` command)



C pre-processor

Macro pre-processor directives preceded by

- **File inclusion**

```
#include <stdio.h>  
#include "includes/stdio.h"
```

Search in standard
include paths

Also search in
current directory

- **Macro expansion**

```
#define PI 3.14159  
#define RADTODEG(x) ((x) * 57.29578)
```

Enforce order of
operations:
RADTODEG(r + 1)

- **Conditional compilation**

```
#ifdef __unix__  
#include <unistd.h>  
#else defined _WIN32  
#include <windows.h>  
#endif
```

Program structure and basic syntax

- **All source code** in C is contained in **subroutines**
- By convention **subroutines that return a value are called functions**
- The **declaration** prototype of a subroutine is also called its **signature**
- There **must be one** special subroutine called **`main()`** which is the **entry point** for the program

- **Higher level program structure** as, e.g. modular programming, is based on **symbol naming conventions**
- Source text is **free form** using a **semi-colon** as statement **terminator**
- **Curly braces** group **blocks of statements**

- **Single line comments** are beginning with **`// This is a comment!`**
- **Multiline comments** are enclosed in **`/* comment */`**

Control flow primitives (selection)

- Conditional execution

```
if (<condition>) { } else if (<condition>) { } else { }
```

C allows to omit braces.
TIP: Always use braces!

- For loops

```
for ( init; condition; increment ) { statement(s); }
```

```
for( a = 10; a < 20; a = a + 1 ) {  
    x = x + 1;  
}
```

- While loops

```
while(condition) { statement(s); }
```

Basic types and variable declaration/definition

Type	Storage size	Value range
<code>char</code>	1 byte	-128 to 127
<code>int</code>	4 bytes	-2,147*10 ⁶ to 2,147*10 ⁶
<code>unsigned int</code>	4 bytes	0 to 4,294,967,295
<code>float</code>	4 bytes	1.2E-38 to 3.4E+38
<code>double</code>	8 bytes	2.3E-308 to 1.7E+308

- All **variables** have to be **defined with a specific type**

```
int i, j, k;
float alpha=0.23, beta= 0.99;
```

Variables are initially **undefined!** Always initialize ALL variables.

```
extern int i;
```

extern enforces a variable declaration

- Variables defined **inside a function** or block are **local**
- Variables defined **outside of functions** are **global**

C arithmetic and relational and logical operators

Operator	Description	Operator	Description
+	Adds two operands	==	Equal
-	Subtracts second from first operand	!=	Not equal
*	Multiplies both operands	>	Greater than
/	Divides numerator by denominator	<	Smaller than
%	Modulus Operator and remainder of after an integer division	>=	Greater than or equal
++	Increment operator increases the integer value by one	<=	Smaller than or equal
--	Decrement operator decreases the integer value by one	&&	Logical AND
			Logical OR
		!	Logical NOT

C types pitfalls and operator precedence

- **Statically typed** variables but **weakly enforced!**
- If the types of **two operands** do **not match** the compiler performs **implicit arithmetic type conversion**
- When **mixing signed with unsigned** integers both **get converted** to unsigned type! **TIP: Do not use unsigned integers!**
- **Floating point literals** are of type **double**. Use **f** suffix for float literals.

- Within an expression, **higher precedence** operators will be evaluated **first**

Higher

Lower

Postfix(([])->+ + --)

Unary(+ - ! &)

Multiplicative(* / %)

Additive(+ -)

TIP: Always be **explicit** and **use brackets**.

Arrays in C

```
type arrayName [ arraySize ];
```

Array size must be an integer constant greater than zero!

```
double balance[10];
```

- Initializing arrays

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

- Accessing array elements

```
double salary = balance[9];
```

Array indices are always zero based!

- C arrays are allocated on the stack

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Pointers

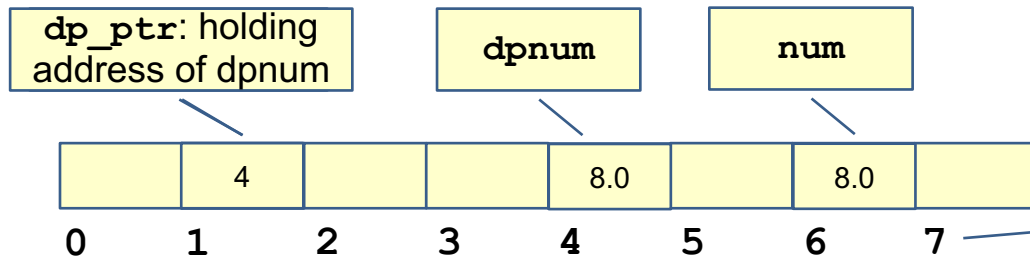
- **Memory** in computers is accessed using **unsigned integers** (called addresses). An address is the key to a storage cell in memory.
- A **pointer** is a variable whose value is the **address of another variable**

- **Definition:** `type *var-name; e.g. double *dp_ptr;`

Use meaningful variable naming

- Get address of a variable: `double dpnum = 8.0; dp_ptr = &dpnum;`

- Access value at pointer location: `double num = *dp_ptr;`



Real memory is addressed byte-wise!

Pointer arithmetic and square bracket notation

Interlude: Addressing modes in the X86-64 ISA

```
movaps [rdi + rax * 8], xmm3
```

BASE + INDEX * SCALE

You can perform computations with pointers: **Pointer arithmetic!**

```
double* A; double* B; double x; int i = 2;
```

```
B = A + i;
```

Reflects addressing mode:

$B = A + i * 8b$

```
x = *(A + i);
```

Get value at $A + i$

A `void*`
pointer type is
a generic
(without type)
pointer

Square bracket shorthand notation:

```
x = A[i]; is equivalent to x = *(A + i);
```

For performant code always use square bracket notation!

Memory management

- Dynamic memory allocation (at runtime) is provided by **libc**

```
#include <stdlib.h>
```

```
int main() {  
    double* A;  
    A = (double*) malloc( 200 * sizeof(double) );  
    // Do computations  
    free(A);  
}
```

Allocate memory on heap.
Takes **size in bytes** as only
argument and returns **void***

Operator to get storage size
of an object or type in bytes

Release memory. **A** must hold
the address that was returned
on allocation!

Structures and typedef

- A **structure** is a **user defined data type** that allows to combine data items of **different kinds**

```
struct Books {  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
};
```

```
Less clumsy way  
typedef struct {  
    ...  
} Books;  
  
Books book1;
```

Pointers to structs:

```
Books *book_ptr = &book1;
```

Accessing items of ptr to struct:

```
int id = book_ptr->book_id;
```

- Declare variable of type struct Books

```
struct Books book1;
```

- Access struct items with member access operator .

```
int id = book1.book_id;
```

Subroutines (aka functions)

```
return_type function_name( parameter list )  
{ body of the function }
```

Example for function definition:

```
int max(int num1, int num2) {  
    int result;  
    if (num1 > num2) { result = num1; } else { result = num2; }  
    return result;  
}
```

Function parameters
are default passed
call by value

For call by reference you need to pass pointers as function parameters!


Pitfalls of call by value

```
void allocate_mem(double *ptr, int size){
    ptr = (double*) malloc( size * sizeof(double) );
}
```

- Using above function

```
double *my_ptr = NULL;
allocate_mem(my_ptr, 1000);
double x = my_ptr[899];
```

TIP: Always use special **NULL** value to initialize pointers!

 Segmentation fault! What happened?

- Solution

```
void allocate_mem(double **ptr, int size);
allocate_mem(&my_ptr, 1000);
```


Strings and programm output

- There is **no string data type** in C!
- **Strings** are **one-dimensional arrays** of characters **terminated** by a **null character '\0'**

```
char greeting[] = "Hello";
```

String literal enclosed by "" is always zero terminated

libc provides routines for IO in `<stdio.h>`

```
#include <stdio.h>
```

...

```
printf("Greeting message: %s\n", greeting );
```

printf format string

More information on format strings:

```
https://en.wikipedia.org/wiki/Printf\_format\_string
```

File IO

- Opening a file

```
FILE *fopen( const char * filename, const char * mode );
```

- Writing to file

```
fprintf(FILE *fp, const char *format, ...);
```

printf format string

- Example

```
#include <stdio.h>
```

```
main() {
```

```
    FILE *fp;
```

```
    fp = fopen("/tmp/test.txt", "w");
```

```
    fprintf(fp, "This is testing for fprintf...\n");
```

```
    fclose(fp);
```

```
}
```

Command line arguments

- Command line arguments are passed as **arguments** to the **main routine**

Number of arguments

List of arguments. `argv[0]`
holds program name!

```
int main( int argc, char **argv ) {  
    if( argc == 2 ) {  
        printf("The argument supplied is %s\n", argv[1]);  
    } else if( argc > 2 ) {  
        printf("Too many arguments supplied.\n");  
    } else {  
        printf("One argument expected.\n");  
    }  
}
```

Putting it all together

- A simple but non-trivial C example code:

- The Bandwidth Benchmark

`https://github.com/RRZE-HPC/TheBandwidthBenchmark`

- Further reading

`https://www.tutorialspoint.com/cprogramming/`