# Structuur van computerprogramma's 2

# 1. Basics on C and Program Organization

- C in a nutshell:
  - Statically typed, general-purpose programming language .
  - Compilation pipeline: preprocessor, compiler, linker.
  - Small language: thin abstraction layer, few keywords, low runtime overhead.
  - Lacks built-in runtime type safety, exceptions, range checks, garbage collection, and object-oriented facilities.
- Brief History:
  - Invented by Dennis Ritchie, AT&T Bell Labs, 1972.
  - The C Programming Language ("K&R C"): 1978.
  - ANSI C (C89/C90), C99, C11, C17 (current, 2018), each adding features or fixes.
- Interpretation vs. Compilation:
  - Interpreters execute code line-by-line. Compilers translate files into machine code, producing executables.

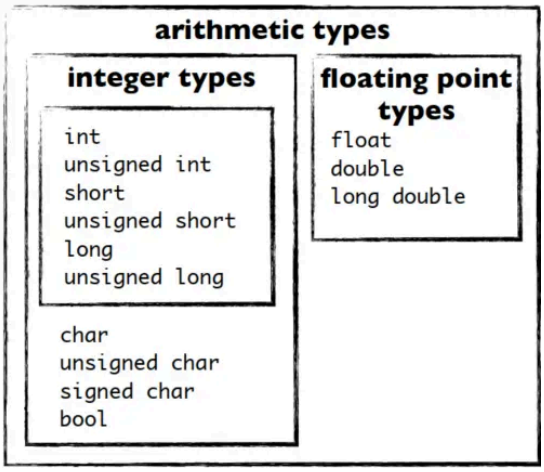
## Data, Values, and Types

- Data is stored in memory regions called "objects" with unique addresses.
- A variable refers to an object holding a value; its type determines allowed values and required memory size.
- Basic declaration: `Type VariableName [=value];` (e.g., `int x = 4;` )
- Four basic arithmetic types: `char` , `int` , `float` , `double` ; with optional `signed` , `unsigned` , `short` , `long` modifiers .
- A type implies a size used to reserve the amount of memory

| Type | Size | Value Range |
|---|---|---|
| char | 1 byte | -128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| int | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned int | 4 bytes | 0 to 4,294,967,295 |
| short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| long | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned long | 4 bytes | 0 to 4,294,967,295 |
| float | 4 byte | 1.2E-38 to 3.4E+38 |
| double | 8 byte | 2.3E-308 to 1.7E+308 |
| long double | 10 byte | 3.4E-4932 to 1.1E+4932 |

*image-3.webp*

  - use `sizeof(type) = #` to see the amount of bytes needed to store the object
- Strings: arrays of `char` ending in `\0` (null terminator) .
- C is weakly typed, supports implicit conversion and explicit casting (such as from `int` to `double` ).




*Arithmetic Types in C.*

## Functions

- All arguments are passed by value.
- Functions defined with `ReturnType Name(ParameterList) { ... }`

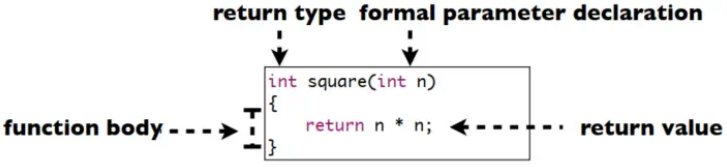



*image-4.webp*

- Declaration (prototype) allows use before definition; definition allocates code or memory.
- Example call mechanics:
  1. Actual parameters evaluated, stored as local (formal) parameters.
  2. Control enters function body.
  3. On `return`, exits to caller .

## Compilation Process

- Preprocessor: Handles directives ( `#include` , `#define` ).
- Compiler: Checks code, translates to machine/object code.
- Linker: Connects compiled code to libraries, forming an executable.
  - Error Types:
    - Compile-time: Syntax errors, type mismatches (detected by compiler).
    - Linker: Undefined references (symbol declared but not found), multiple definitions.
    - Runtime: Segmentation faults (invalid memory access), division by zero.
- Header s ( `.h` ) typically contain shared declarations; source s ( `.c` ) contain definitions.

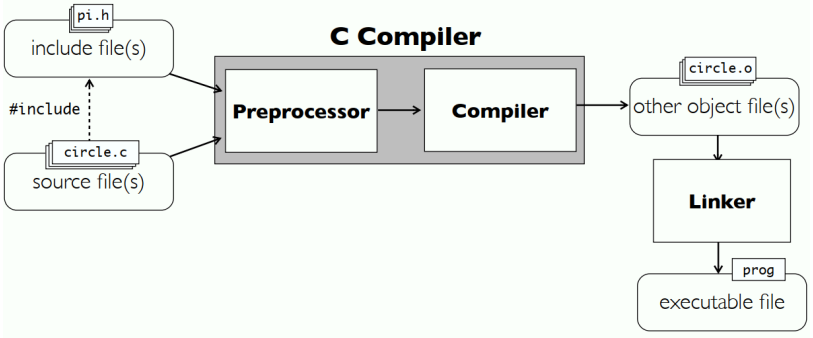



*image-59.png*

- Macros: object-like ( `#define MSG "Hi"` ) and function-like ( `#define SUM(X,Y) ((X)+(Y))` ).

- Macros vs. Functions:
  - Type Safety: Macros are untyped (text substitution); functions check types.
  - Side Effects: Macros evaluate parameters every time they appear (e.g., `SQUARE(x++)` increments twice); functions evaluate arguments once.
  - Performance/Size: Macros cause code bloat (expanded inline) but no function call overhead; functions save space but have call overhead (unless inlined).
  - Scope: Macros ignore scope (visible from definition to end of file); functions obey block scope.
- Conditional compilation: `#ifdef`, `#ifndef`, etc. (used for header guards).

### Program Organization

- Use `extern` to declare variables/functions defined elsewhere.
- Standard headers declare common functions (e.g., `<stdio.h>` for I/O).
- `main` is the starting point, returns `int` (status code).

### Lexical Elements

- Identifiers: names for variables, functions, types. Must start with a letter or `_`, case-sensitive, not a reserved keyword.
- Comments: `/* ... */`

---

# 2. Fundamental Programming Concepts of C

## Basic Data Types & Operators

- Scalar types: single value (`int`, `float`, `char`).
- Compound types: arrays (group fixed number of elements of same type) .
- Character constants: single quotes, value = ASCII/Unicode code .
  - Example: `'x' == 120` (ASCII).
- Arrays: fixed size, elements accessed via indexes (starting at 0).
- Strings: arrays of `char` ending in `\0` .
- Arithmetic operators: `+`, `-`, `*`, `/`, `%`. Use parentheses to control precedence.
- Relational/logical: `==`, `!=`, `<`, `>`, `<=`, `>=`, `&&`, `||`, `!`.
- Assignment: `=`, `+=`, `-=`, etc.
- Statements (instructions, no value) vs. expressions (produce value). A statement is an expression plus `;`.
- Braces `{}` group statements into blocks; blocks create scope.

## Control Flow

- if statement:

```
if (condition)
    statementIfTrue;
else
    statementIfFalse; // optional
```

Ternary shorthand: `z = (x > y) ? x : y;`

- while statement:

```
while (condition)
    statement;
```

- for statement:

```
for (init; condition; increment)
    statement;
```

- All three clauses are optional (can create infinite loops).
- do-while statement:

```
do
    statement;
while (condition);
```

- Always executes the statement at least once.
- switch statement:

```
switch (expr) {
    case 1: ... break;
    case 2: ... break;
    default: ... break;
}
```

- Use `break` to avoid "fall-through".
- break, continue, and return:
  - `break` : exit innermost loop or switch.
  - `continue` : next iteration of loop.
  - `return` : exit function, optionally returning value.

## Functions (Details)

- Formal parameters are local variables within the function .
- Functions can have side-effects if they use or change global variables.
- Functions without explicit return type default to returning `int` (bad practice!).
- Variadic functions (functions that take a variable number of parameters) use `<stdarg.h>` and ellipsis `...` , e.g., `double average(int count, ...)` .

## Variable Types and Scope

- Automatic variables: default in blocks, on the stack, deallocated at block exit.
- Static variables: persist between calls, accessible in scope defined.
- Variables outside functions: global scope (visible everywhere after definition).
- Blocks define variable lifetimes and visibility.
  - Macros: Preprocessor directives do not observe C scope rules (ignore `{}` ).
- `static` outside functions: scope only (not visible to other s).
- `register` variables: Hint to compiler to store variable in a CPU register for speed (cannot take address with `&` ).

---

# 3. Data Structures and Abstraction

## 3.1 Pointers

- A pointer is a variable that contains the memory address of another variable.
- Every variable ( `int x` , `float y` ) has a memory address where its value is stored.
- Declaration: `type *name;`
  - `int *ip;` // `ip` is a pointer to an integer.
  - `char *cp;` // `cp` is a pointer to a character.

### Pointer Operators

1. `&` (Address-of operator): Returns the memory address of a variable.

```
int x = 5;
int *ip;
ip = &x; // ip now "points" to x.
```

2. `*` (Dereference / Indirection operator): Retrieves the value located at a specific address.

```
int y = *ip; // y is now 5 (the value ip points to)
*ip = 10;    // The value of x is now 10
```

### The `NULL` Pointer

- A pointer that points to nothing should be given the value `NULL` .
- This is a special value that indicates the pointer is invalid.
- It is good practice to initialize pointers to `NULL` .
- `NULL` is defined in `<stdio.h>` and `<stdlib.h>` .

```
int *p = NULL;
```

## 3.2 Pointers and Functions (Pass-by-Value)

- C always uses pass-by-value. This means that when you pass a variable to a function, the function receives a copy of that variable.
- Modifying the copy within the function has no effect on the original.

```
void swap_failed(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    // Only the copies are swapped!
}

int main() {
    int x = 10, y = 20;
    swap_failed(x, y);
    // x is still 10, y is still 20
}
```

- Solution: To modify a variable *inside* a function, we must pass a pointer to that variable.
- We pass the *addresses* of `x` and `y` . The function receives copies of the *pointers*, but they still point to the original `x` and `y` .

```c
void swap_correct(int *pa, int *pb) {
    int temp = *pa; // Get the value at address 'pa' (of x)
    *pa = *pb;      // Put the value of y at the address of x
    *pb = temp;     // Put the old value of x at the address of y
}

int main() {
    int x = 10, y = 20;
    swap_correct(&x, &y); // Pass the *addresses*
    // x is now 20, y is now 10
}
```

## 3.3 Pointers and Arrays

- The name of an array (e.g., `a`) is *not* a variable. It is an expression that *decays* into a pointer to the first element (`&a[0]`).
- Because of this, pointers and arrays in C are inextricably linked.

> **The Great Equivalence**
> If a is an array and p is a pointer (p = a;), then:
> - `a[i]` is exactly the same as `*(a + i)`
> - `&a[i]` is exactly the same as `a + i`

### Pointer Arithmetic

- When you perform arithmetic on pointers, the calculation is automatically scaled by the `sizeof` the type.
- `p + i` does *not* mean `address p + i bytes`, but `address p + (i * sizeof(*p)) bytes`.
- This allows you to "step" through an array using a pointer.

```c
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int *p = a; // p points to a[0]

// These are all equivalent:
printf("%d", a[3]);      // 3
printf("%d", *(a + 3));  // 3
printf("%d", *(p + 3));  // 3
printf("%d", p[3]);      // 3 (yes, this is allowed!)
```

- You can also subtract pointers (`p - q`) to get the number of *elements* between them.
  - Multi-dimensional Arrays vs. Pointer Arrays:
    - `int a[2][3]`: Contiguous 2D block of memory. `a[i]` is a sub-array. Address calculation: `base + (i*cols + j)*size`.
    - `int p` (or `int *p[2]`): Array of pointers (often to rows). "Rows" can be scattered in memory. `p[i]` is a pointer. Access involves an extra memory lookup.

## 3.4 Arrays of Pointers

- You can have an array where each element is a pointer.
- A common example is an array of `char*` (strings).
- `char *lineptr[100];` // An array of 100 pointers-to-char.

- This is extremely useful for sorting text. Instead of moving the complete (long) strings, you only move the (small) pointers in the array.

## 3.5 Dynamic Memory Allocation (The Heap)

Dynamic memory allocation involves storing data on the heap at runtime, rather than the stack. This is used for data whose size is unknown until runtime or changes during execution.

- `malloc`: The primary allocation function. It reserves `n` bytes on the heap and returns a pointer (of type `void*`) to the beginning of that block. The memory is not initialized (it contains "garbage").
  - `void*` is a generic pointer that can point to any type. You must *cast* it to the correct type.
  - Returns `NULL` if no more memory is available.

```c
#include <stdlib.h>
int *p = (int *) malloc(10 * sizeof(int)); // Allocate space for 10 integers
if (p == NULL) {
    // Error: out of memory
}
```

- `calloc`: `(size_t n, size_t size)`: Reserves memory for an array of `n` elements, each `size` bytes large. The memory is initialized to zero.
- `realloc`: `(void *ptr, size_t new_size)`: Changes the size of a previously allocated block (`ptr`).
- `free`: The deallocation function. It returns the memory block to the heap.
  - Memory Leak: If you forget to call `free` for every `malloc` / `calloc` / `realloc`, the memory remains reserved. This is a *memory leak*.
  - You must *never* call `free` on a pointer that did not come from `malloc` (or its family).

```c
free(p); // Free the memory block
p = NULL; // Good practice to avoid "dangling pointers"
```

## 3.6 User-Defined Types: Structures (`struct`)

- A `struct` is a collection of one or more variables (of possibly different types) that are bundled together under a single name.
- It is the basis for "objects" in C.

### Declaration and Usage

```c
struct point {
    int x;
    int y;
}; // Semicolon is mandatory!

int main() {
    // Declaration of a struct variable
    struct point pt;

    // Access members with the '.' (dot) operator
    pt.x = 10;
    pt.y = 20;

    // Initialization can also be done directly
    struct point max = { 320, 200 };
}
```

### Pointers to Structures

- It is very common to work with pointers to `structs` (e.g., when passing them to functions).

```
struct point *pp = &pt;
```

- To access members via a pointer, there are two ways:
  1. `(*pp).x = 10;` // Parentheses are necessary (operator precedence)
  2. `pp->x = 10;` // The '->' (arrow) operator. This is the standard.

## 3.7 Self-referential Structures

- A `struct` can contain a pointer to its own type.
- This is the building block for all linked data structures (linked lists, trees, etc.).

```
// Definition of a 'node' in a binary tree
struct tnode {
    int value;
    struct tnode *left;  // Points to the left subtree
    struct tnode *right; // Points to the right subtree
};
```

## 3.8 Type Aliasing ( `typedef` )

- `typedef` creates an *alias* (synonym) for a type. It makes the code much more readable.
- `typedef int Length;` // `Length` is now a synonym for `int` .
- It is used *extremely* often with `structs` :

```
// Cumbersome way
struct tnode { ... };
struct tnode *p;

// With typedef
typedef struct tnode {
    int value;
    struct tnode *left;
    struct tnode *right;
} Tnode; // 'Tnode' is now the name of the type

// Now you can just write:
Tnode *p; // Much clearer!
```

- Wrong Usages:
  - Hiding pointers in typedefs for non-opaque types (e.g., `typedef char* String;` ) can make code confusing ( `const String` is `char* const` , not `const char*` ).
  - Typedefs do not create new types for type-checking, just aliases.

## 3.9 `union` and `bit-fields`

- `union` : Similar to a `struct` , but all members share the same memory. A `union` can only hold one of its members at a time. The size of the `union` is the size of its *largest* member.
- `bit-fields` : Allows you to define variables that are smaller than a byte (e.g., 1 bit, 4 bits). Used for memory-mapping or register-level C.

```
struct {
    unsigned int is_keyword : 1; // uses 1 bit
    unsigned int is_extern  : 1; // uses 1 bit
    unsigned int is_static  : 1; // uses 1 bit
} flags;

flags.is_keyword = 1; // Set the bit
   ```
```

- Downsides:
  - Portability: Ordering of bits (LSB vs MSB first) is implementation-defined.
  - Address: You cannot take the address ( `&` ) of a bit-field member.
  - Performance: Accessing bit-fields may be slower (requires masking/shifting).

## 3.10 Advanced Pointers: Command-Line Arguments

- The `main` function can receive arguments from the command-line.
- The standard signature is: `int main(int argc, const char * argv[])`
  - `argc` (argument count): An `int` that counts the number of arguments (including the program name itself).
  - `argv` (argument vector): An array of pointers to strings.
    - `argv[0]` is the name of the program.
    - `argv[1]` is the first argument.
    - `argv[argc-1]` is the last argument.
    - `argv[argc]` is guaranteed to be `NULL`.

```
// Example: ./echo hello world
// argc == 3
// argv[0]    -> "./echo"
// argv[1]    -> "hello"
// argv[2]    -> "world"
// argv[3]    -> NULL

int main(int argc, const char * argv[]) {
    // Start at i=1 to skip the program name
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
    return 0;
}
```

## 3.11 Advanced Pointers: Function Pointers

- A function pointer is a variable that contains the address of a function.
- This allows you to pass functions as arguments to other functions (callbacks).
- Declaration: `return_type (*name)(parameter_types);`

```
// 'comp' is a pointer to a function that
// accepts two 'const void*' parameters
```
```

```
// and returns an 'int'.
int (*comp)(const void *, const void *);
```

### Example: `qsort`

- The standard `qsort` function in `stdlib.h` is the classic example.
- It sorts an array, but it needs a function pointer to know *how* to compare elements.

```
#include <stdlib.h>

// Comparison function for integers
int compare_ints(const void *a, const void *b) {
    int int_a = *((int*)a);
    int int_b = *((int*)b);
    return int_a    - int_b; // <0, 0, or >0
}

int main() {
    int numbers[] = {5, 2, 9, 1, 5};
    int n = 5;

    // Use qsort
    qsort(numbers,          // The array
          n,                // Number of elements
          sizeof(int),      // Size of one element
          compare_ints);    // The function pointer

    // 'numbers' is now {1, 2, 5, 5, 9}
}
```

## 3.12 Advanced Pointers: Jump Tables

- A jump table is an array of function pointers.
- It provides a highly efficient alternative to a `switch` statement, especially when the cases are dense (e.g., 0, 1, 2, 3...).
- Instead of a `switch` checking multiple cases, you can use an integer to directly index into the array and call the correct function, which is often an `O(1)` operation.

### Example: `switch` vs. Jump Table

"Before" - Using a `switch` statement:

```
// 'op' is 0 for add, 1 for subtract, etc.
int a = 10, b = 5;
int res;
switch(op) {
    case 0: // Add
        res = a + b;
        break;
    case 1: // Subtract
        res = a    - b;
        break;
    case 2: // Multiply
        res = a * b;
        break;
```

```
    case 3: // Divide
        res = a / b;
        break;
}
```

"After" - Using a Jump Table:

```c
#include <stdio.h>

// 1. Define the functions that will be in the table
int op_add(int a, int b) { return a + b; }
int op_sub(int a, int b) { return a   - b; }
int op_mul(int a, int b) { return a * b; }
int op_div(int a, int b) { return a / b; }

// 2. Define a function pointer type for clarity (optional, but recommended)
typedef int (*OperationFunction)(int, int);

int main() {
    // 3. Create the jump table (array of function pointers)
    OperationFunction op_funct[] = {
        op_add,
        op_sub,
        op_mul,
        op_div
    };

    int op = 1; // 1 = subtract
    int a = 10, b = 5;

    // 4. Call the function directly from the table
    // Ensure 'op' is within bounds! (0 <= op < 4)
    int res = op_funct[op](a, b);

    printf("Result: %d\n", res); // Prints "Result: 5"
    return 0;
}
```

## 3.13 Abstract Data Types (ADT)

- An ADT is a way to hide data (encapsulation) by separating the interface from the implementation.
- This is the basis of object-oriented programming.

1. Interface (Header file, `.h`):
    - Public API.
    - Contains `typedef`s and function prototypes (what the user *is allowed* to see).
    - The *internal* `struct` is often hidden (an opaque type) by only declaring a pointer, not the `struct` itself. E.g., `typedef struct stack *Stack;`
2. Implementation (Source file, `.c`):
    - Private implementation.
    - Contains the *full* definition of the `struct` (e.g., `struct stack { ... }`).
    - Contains the code (body) of all functions from the `.h` file.
    - The user cannot access this directly.

### 3.13.1 Example: Stack (LIFO)

- `stack.h` (Interface)

```c
#ifndef STACK_H
#define STACK_H

// Opaque type: User knows 'Stack' exists,
// but not what's inside 'struct stack'.
typedef struct stack *Stack;

Stack stack_create(void);
void  stack_destroy(Stack s);
void  stack_push(Stack s, int value);
int   stack_pop(Stack s);
int   stack_is_empty(Stack s);

#endif
```

- `stack.c` (Implementation)

```c
#include "stack.h"
#include <stdlib.h>

// Full definition, hidden from the user.
// (Implementation here as a linked list)
struct node {
    int value;
    struct node *next;
};

struct stack {
    struct node *top;
};

Stack stack_create(void) {
    Stack s = malloc(sizeof(struct stack));
    if (s) {
        s->top = NULL;
    }
    return s;
}

void stack_destroy(Stack s) {
    // ... code to pop all nodes and free(s) ...
}

void stack_push(Stack s, int value) {
    // ... code to create a new node and add it to the top ...
}

int stack_pop(Stack s) {
    // ... code to remove top node, return value, and free node ...
    // (Must check for empty stack)
}

int stack_is_empty(Stack s) {
```

```
    return s->top == NULL;
}
```

### 3.13.2 Example: Queue (FIFO)

- Another classic ADT, follows the same principle.
- Interface: `queue_create()`, `queue_destroy()`, `queue_insert(q, val)`, `queue_delete(q)`, `queue_is_empty(q)`.
- Implementation: Is often efficiently implemented with a circular array (an array that "wraps around") to make `insert` (at the rear) and `delete` (from the front) fast `O(1)` operations.

---

# 4. Input and Output

- Standard Library (`stdio.h`): I/O facilities are provided by the standard library, not the language keywords.
- Streams:
  - `stdin` (Standard Input), `stdout` (Standard Output), `stderr` (Standard Error).
  - Usually connected to keyboard/screen but can be redirected.

## Character I/O

- `int getchar(void)`: Returns next char from stdin or `EOF`.
- `int putchar(int c)`: Writes char to stdout.

## Formatted I/O

- `printf(format, ...)`: Output formatted text.
  - `%d` (int), `%f` (float), `%s` (string), `%c` (char), `%x` (hex).
  - Modifiers for width and precision (e.g., `%4d`, `%.2f`).
- `scanf(format, ...)`: Input formatted text.
  - Requires pointers to variables (e.g., `&x`).
  - Returns number of items matched.

## File I/O

- Opening/Closing:
  - `FILE *fopen(const char *filename, const char *mode)`
  - Modes: `"r"` (read), `"w"` (write, overwrites), `"a"` (append), `"b"` (binary).
  - `int fclose(FILE *stream)`: Closes the file, flushes buffers.
- Reading/Writing:
  - `fgetc(fp)`, `fputc(c, fp)`: Character level.
  - `fgets(buf, max, fp)`, `fputs(str, fp)`: Line level.
  - `fprintf(fp, ...)`, `fscanf(fp, ...)`: Formatted.
- Random Access:
  - `fseek(fp, offset, origin)`: Move pointer (SEEK_SET, SEEK_CUR, SEEK_END).

## UNIX System Interface (Low-level)

- File Descriptors: Integers identifying open files (0, 1, 2 for stdin/out/err).
- System Calls:

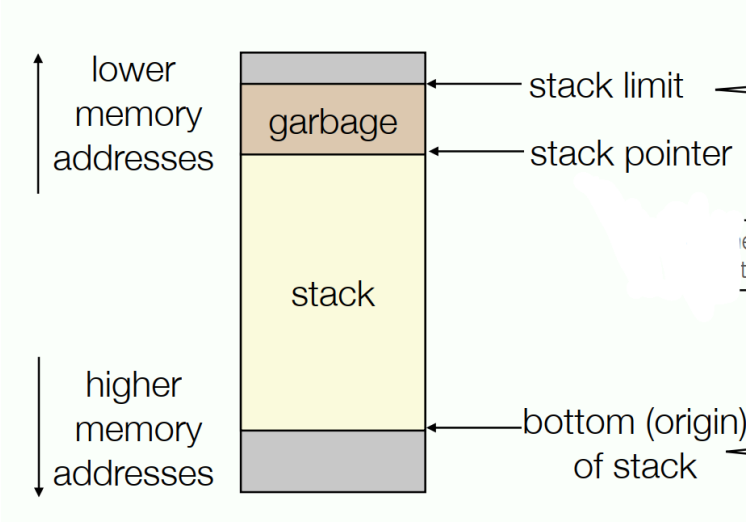
    - `open(name, flags, perms)` : Returns file descriptor. Flags: `O_RDONLY` , `O_WRONLY` , `O_CREAT` .
    - `read(fd, buf, count)` : Reads bytes into buffer.
    - `write(fd, buf, count)` : Writes bytes from buffer.
    - `close(fd)` : Closes descriptor.
- Buffered vs Unbuffered:
    - Standard I/O ( `stdio` ) is buffered (minimizes system calls).
    - System calls ( `unistd.h` ) are unbuffered (direct kernel interaction).
- Error Handling:
    - `errno` : Global integer variable (in `<errno.h>` ) set by system calls/library functions on error.
    - `perror(const char *s)` : Prints `s` followed by the error message corresponding to `errno` to stderr.
    - `exit(int status)` : Terminates the program, flushing buffers and closing files. `0` = success, non-zero = failure.

---

# 5. Execution, Security, and Structured Control Flow

## 5.1 Execution Model

- Memory Layout:
    - Text Segment: Code (read-only).
    - Data Segment: Initialized globals/statics.
    - BSS: Uninitialized globals/statics (zeroed).
    - Heap: Dynamic memory ( `malloc` ).
    - Stack: Local variables, function calls.




*when the stack pointer < stack limit, the program suffers from stack overflow and receives a segmentation fault*

- Stack Frames:
    - Created on function call. Contains:
    - Function arguments.
    - Return address.
    - Saved Frame Pointer ( `EBP` ).
    - Local variables.

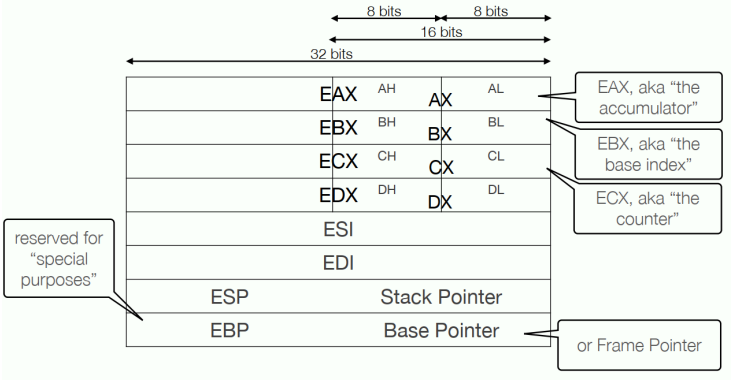



*image-61.png*

- Calling Conventions ( `cdecl` ):
  - Arguments pushed right-to-left.
  - Caller cleans up stack.
- Function Call Steps:
  1. Epilogue: Before a function starts (caller side), arguments are pushed (32-bit) or moved to registers (64-bit).
  2. Prologue: Inside the called function, setup the stack frame (push old `EBP` , set `EBP` to current `ESP` , allocate space for locals).
  3. Body: Execute function code.
  4. Epilogue: Restore stack pointer and base pointer (leave), return (pop return address).
- Calling Conventions:
  - cdecl (32-bit x86):
  - Arguments passed on the stack (pushed right-to-left).
  - Return value in `EAX` .
  - Caller cleans up the stack.
- System V AMD64 ABI (64-bit):

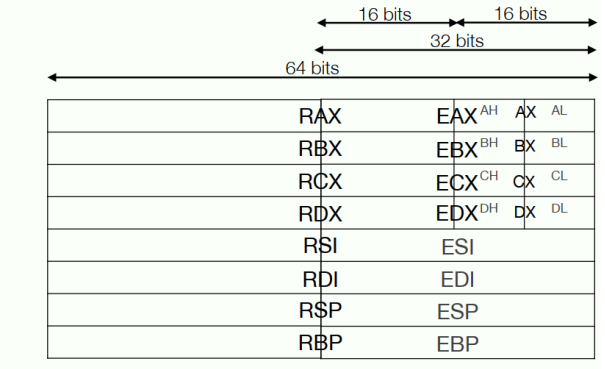



*image-62.png*

  - First 6 integer/pointer arguments passed in registers: `RDI` , `RSI` , `RDX` , `RCX` , `R8` , `R9` .
    - Remaining arguments on the stack.
    - Return values in `RAX` (and `RDX` ).
    - "Red zone": 128 bytes below stack pointer safe from signal handlers.
    - Recursion: Each call creates a new stack frame.

## 5.2 Security Vulnerabilities

- Buffer Overflow: Writing past the end of an array (stack or heap).
  - Can overwrite return address to execute malicious code (shellcode).
  - Defense: Bounds checking, Stack Canaries (values placed before return address), ASLR (Address Space Layout Randomization).
  - unsafe functions: `gets` , `strcpy` , `sprintf` (use `fgets` , `strncpy` , `snprintf` instead).

- Format String Vulnerabilities:
  - `printf(user_input)` allows attackers to read/write memory using `%x` or `%n`.
- Stack Exhaustion: Infinite recursion fills the stack.
  - Tail Call Optimization (Trampolinig): Technique to convert recursion to iteration or jumps to avoid stack growth.
  - Trampoline: A loop that iteratively calls thunks (functions returning the next step).

## 5.3 Structured Control Flow

- Goto Statement: Unconditional jump. Considered "harmful" (Dijkstra) as it leads to "spaghetti code".
- Structured Alternatives: Loops (`for`, `while`), `if/else`, functions.
- Non-Local Jumps (`setjmp.h`):
  - `setjmp(env)`: Saves current CPU state (registers, stack pointer, PC) to `env`. Returns 0.
  - `longjmp(env, val)`: Restores state saved in `env`. `setjmp` returns `val`.
  - Used for exception handling (try/catch simulation) and coroutines.
- Continuations: Represents "the rest of the computation".

---

# 6. Rust: Memory Safety without GC

- Goals: Memory safety, concurrency, performance (comparable to C++).
- Strong vs Weak Typing:
  - C (Weakly Typed): Allows implicit conversions (e.g., `char` to `int`, `void*` to any pointer). Flexible but prone to type errors.
  - Rust (Strongly Typed): Enforces strict type checking. No implicit casting for most types (must use `as` keyword). Prevents interpreting memory as the wrong type.
- Cargo: Build system and package manager (`cargo new`, `cargo build`, `cargo run`).
  - `Cargo.toml`: Metadata and dependencies.

## 6.1 Variables and Mutability

- Variables are immutable by default: `let x = 5;` (`x = 6` is an error).
- Use `mut` for mutability: `let mut x = 5;`.
- Shadowing: Redeclaring a variable (`let x = ...`) shadows the previous one, allowing type change.

## 6.2 Ownership System

The core feature ensuring memory safety without a garbage collector.

1. Each value has a variable that’s called its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped (freed).

- Move Semantics: Assigning a value to another variable *moves* ownership (for non-`Copy` types like `String`). The old variable becomes invalid.
  - `let s1 = String::from("hello"); let s2 = s1;` -> `s1` is now invalid.
- Clone: Use `.clone()` for a deep copy if needed.
- Copy Trait: Simple types (integers, bool, char) implement `Copy` (assignment copies bits, old variable remains valid).

## 6.3 References and Borrowing

Instead of taking ownership, functions can borrow values using references ( `&` ).

- Immutable References ( `&T` ): Read-only access.
- Mutable References ( `&mut T` ): Read/write access.
- The Rules of Borrowing:
  1. At any given time, you can have either one mutable reference or any number of immutable references.
  2. References must always be valid (no dangling references).