Function Calling Conventions in C (x86 Architecture)

1. Introduction

Function calling conventions define how function arguments are passed, how the return value is handled, and how the stack is managed during a function call. This varies between 32-bit and 64-bit x86 architectures.

---

2. Calling Conventions in x86 (32-bit)

In 32-bit x86, the function calling conventions are mainly:

2.1 cdecl (C Declaration)

- Arguments are pushed onto the stack **from right to left**.
- Caller cleans the stack after the function returns.
- Return value is stored in **EAX**.

Example:

```
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}

int main() {
    int result = sum(3, 4);
    printf("Result: %d\n", result);
    return 0;
}
```

Assembly Code (gcc -m32 -S output snippet):

```
push    4   ; Push second argument
push    3   ; Push first argument
call    sum ; Call function
add     esp, 8 ; Clean up the stack (caller cleans up)
```

2.2 stdcall (Standard Call)

- Arguments are pushed **right to left**.
- **Callee cleans the stack**.
- Used in Windows API functions.

Assembly Code (MSVC stdcall function):

```
push    4   ; Push second argument
push    3   ; Push first argument
call    sum ; Call function
; No stack cleanup needed by caller
```

2.3 fastcall

- First two arguments passed via **ECX, EDX** registers.
- Remaining arguments pushed on the stack.
- **Callee cleans the stack**.

Example:

```
__fastcall int multiply(int a, int b) {
    return a * b;
}
```

2.4 thiscall (Used for C++ Member Functions)

- `this` pointer is passed in **ECX**.
- Other arguments are passed on the stack.

---

3. Calling Conventions in x86-64 (64-bit)

In 64-bit architecture, function arguments are primarily passed in registers, reducing the reliance on the stack, leading to better performance.

3.1 System V ABI (Linux/macOS)

- First **six** integer/pointer arguments are passed in **RDI, RSI, RDX, RCX, R8, R9**.
- Floating-point arguments are passed in **XMM0–XMM7**.
- Remaining arguments are passed on the stack.
- Caller cleans the stack.
- The return value is stored in **RAX** (or XMM0 for floating-point values).

Assembly Code Example:

```
mov    edi, 3   ; First argument (RDI)
mov    esi, 4   ; Second argument (RSI)
call   sum      ; Call function
```

3.2 Microsoft x64 ABI (Windows)

- First **four** integer arguments are passed in **RCX, RDX, R8, R9**.

- Floating-point arguments are passed in **XMM0–XMM3**.
- Remaining arguments are passed on the stack.
- Caller cleans the stack.
- The return value is stored in **RAX** (or XMM0 for floating-point values).
- A "shadow space" (32 bytes) is allocated on the stack before function calls for register arguments.

Example Function Call in Windows x64 ABI:

```
mov    rcx, 3   ; First argument (RCX)
mov    rdx, 4   ; Second argument (RDX)
call   sum      ; Call function
```

3.3 Changing the Calling Convention in x86-64

Unlike x86-32, where multiple calling conventions exist, x86-64 architectures have standard ABIs. However, you can override them manually:

Using `__attribute__((sysv_abi))` (GCC/Clang, Linux/macOS)

```
#include <stdio.h>

int __attribute__((sysv_abi)) sum(int a, int b) {
    return a + b;
}
```

This forces the function to use System V calling convention explicitly.

Using `__vectorcall` (MSVC, Windows)

```
#include <stdio.h>

int __vectorcall sum(int a, int b) {
    return a + b;
}
```

This modifies how arguments are passed in registers (favoring SIMD registers for floating-point).

Inline Assembly to Override Conventions

You can manually override register assignments before function calls using inline assembly:

```
mov    rdi, 3   ; Force System V ABI style
mov    rsi, 4
call   sum
```

3.4 Comparison: System V vs Microsoft x64 ABI

Feature	System V (Linux/macOS)	Microsoft x64 (Windows)
Integer Arguments	RDI, RSI, RDX, RCX, R8, R9	RCX, RDX, R8, R9

Feature	System V (Linux/macOS)	Microsoft x64 (Windows)
Floating Arguments	XMM0–XMM7	XMM0–XMM3
Stack Cleanup	Caller	Caller
Shadow Space	None	32 bytes
Return Value	RAX (XMM0 for FP)	RAX (XMM0 for FP)

4. Observing Calling Conventions Programmatically

We can inspect calling conventions using **disassembly and debugging**.

4.1 Using GDB (Linux/macOS)

Compile with debugging symbols:

```
gcc -m32 -g -o test test.c
```

Run in GDB:

```
gdb ./test
break sum
run
info registers  # Check argument registers or stack usage
```

4.2 Using objdump

To view the assembly:

```
gcc -m32 -S test.c -o test.s
objdump -d ./test
```

4.3 Using MSVC Debugger (Windows)

Compile:

```
cl /FAs /Zi test.c
```

Run in Debug Mode and observe register values.

5. Conclusion

Understanding function calling conventions helps in debugging, reverse engineering, and low-level optimization. The key differences between **x86 (32-bit)** and **x86-64 (64-bit)** are in how arguments are passed and how stack cleanup is handled.

- **x86-32:** Mostly stack-based argument passing.
- **x86-64:** Register-based argument passing for improved performance.

- You can modify the calling convention using attributes like `sysv_abi`, `vectorcall`, or inline assembly.

Application Binary Interface (ABI) - Short Notes

An **Application Binary Interface (ABI)** defines the low-level interface between software (programs) and the underlying operating system or hardware. It ensures that binaries compiled with different compilers or languages can interact correctly.

Key Components of an ABI

1. **Calling Conventions** – Defines how function arguments are passed (registers vs stack), return values, and stack cleanup responsibilities.
2. **Register Usage** – Specifies which registers are preserved across function calls and which are caller/callee-saved.
3. **Stack Frame Layout** – Determines how function stack frames are organized, including local variables and return addresses.
4. **Data Alignment & Padding** – Defines how data structures are aligned in memory for performance and compatibility.
5. **Exception Handling** – Specifies how exceptions are propagated and handled across function calls.
6. **System Calls Interface** – Defines how user-space programs interact with the operating system kernel.

Common ABIs

- **System V ABI (Linux/macOS x86-64)** – Uses registers RDI, RSI, RDX, RCX, R8, R9 for function arguments.
- **Microsoft x64 ABI (Windows)** – Uses RCX, RDX, R8, R9 for function arguments, with shadow space.
- **ARM EABI (Embedded Systems)** – Defines ARM register conventions for efficient execution.

In C, you can implement a **variable argument list function** using the `<stdarg.h>` library, which provides macros to handle functions with a variable number of arguments.

Steps to Implement a Variadic Function

1. **Include `<stdarg.h>`** for handling variable arguments.
2. **Use `va_list`** to declare a variable to traverse the arguments.
3. **Initialize the list using `va_start`**, passing the last known fixed parameter.
4. **Extract each argument using `va_arg`**, specifying the expected type.
5. **Clean up using `va_end`** to avoid undefined behavior.

Example: A Function to Calculate the Sum of N Numbers

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

// Function to compute sum of given numbers
int sum(int count, ...) {
    va_list args;          // Declare variable argument list
    va_start(args, count); // Initialize the list, 'count' is the last fixed
argument

    int total = 0;
    for (int i = 0; i < count; i++) {
        total += va_arg(args, int);  // Fetch the next argument
    }

    va_end(args); // Cleanup
    return total;
}

int main() {
    printf("Sum of 3, 4, 5: %d\n", sum(3, 3, 4, 5));
    printf("Sum of 10, 20, 30, 40: %d\n", sum(4, 10, 20, 30, 40));
    return 0;
}
```

Explanation:

- The first parameter `count` determines how many arguments follow.
- The `va_list args` holds the arguments.
- `va_arg(args, int)` retrieves the next argument in the list.
- `va_end(args)` releases the memory.

Example: A Function That Accepts Different Data Types

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

// Function to print different data types
void printValues(int num, ...) {
    va_list args;
    va_start(args, num);

    for (int i = 0; i < num; i++) {
        switch (i) {
            case 0: printf("Integer: %d\n", va_arg(args, int)); break;
            case 1: printf("Double: %f\n", va_arg(args, double)); break;
            case 2: printf("Char: %c\n", va_arg(args, int)); break; // char is
promoted to int
        }
    }

    va_end(args);
}

int main() {
    printValues(3, 42, 3.14, 'A');
    return 0;
}
```

Key Notes:

- `double` arguments are automatically promoted (hence `va_arg(args, double)`).
- `char` arguments are promoted to `int` (hence `va_arg(args, int)`).

Limitations of Variadic Functions

- No type safety (incorrect types may lead to undefined behavior).
- No way to determine the number of arguments without an explicit count.
- Cannot directly handle `struct` types (must use pointers).

Implementing a `printf`-like Function Using Variadic Arguments in C

The `printf` function in C is a perfect example of a **variadic function** that accepts different argument types based on format specifiers. Below is a simplified implementation of a custom `my_printf` function.

Custom `printf` Implementation

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

// Custom printf function
void my_printf(const char *format, ...) {
    va_list args;
    va_start(args, format);

    for (const char *ptr = format; *ptr != '\0'; ptr++) {
        if (*ptr == '%' && *(ptr + 1) != '\0') {
            ptr++;  // Move to the format specifier
            switch (*ptr) {
                case 'd': {
                    int i = va_arg(args, int);
                    printf("%d", i);
                    break;
                }
                case 'f': {
                    double f = va_arg(args, double);
                    printf("%f", f);
                    break;
                }
                case 'c': {
                    char c = (char)va_arg(args, int); // char is promoted to int
                    printf("%c", c);
                    break;
                }
                case 's': {
                    char *s = va_arg(args, char *);
                    printf("%s", s);
                    break;
                }
                default:
                    putchar('%');
                    putchar(*ptr);
            }
        } else {
            putchar(*ptr);
        }
    }

    va_end(args);
}

int main() {
    my_printf("Hello %s! Your score is %d and percentage is %f. Grade: %c\n",
              "Alice", 95, 92.5, 'A');
    return 0;
}
```

How It Works:

1. **`va_start(args, format)`** initializes the argument list.
2. **Iterates through the format string** character by character.
3. **When `%` is found**, the next character determines the data type:
   - `%d` → Extracts an `int` using `va_arg(args, int)`.
   - `%f` → Extracts a `double` using `va_arg(args, double)`.
   - `%c` → Extracts a `char` (promoted to `int`).
   - `%s` → Extracts a `char*` (string).
4. **Calls `printf` for each extracted value.**
5. **If no format specifier is found, prints the character as is.**
6. **`va_end(args)`** is called to clean up.

Example Output:

```
Hello Alice! Your score is 95 and percentage is 92.500000. Grade: A
```

Key Takeaways:

- `va_list`, `va_start`, `va_arg`, and `va_end` are essential for variadic functions.
- `printf`-like functions need to handle multiple data types carefully.
- `char` arguments are promoted to `int`, and `float` is promoted to `double` when passed to a variadic function.