# Advanced System Programming

Advanced System Programming

# Agenda

- ARM Assembly Language
- Static Library
- Dynamic Library
- Error Handling
- Asserts

# Agenda

- **Introduction**

  Architecture

  Programmers Model

  Instruction Set

# History of ARM

- **ARM (Acorn RISC Machine) started as a new, powerful, CPU design for the replacement of the 8-bit 6502 in Acorn Computers (**Cambridge, UK, 1985**)**
- **First models had only a 26-bit program counter, limiting the memory space to 64 MB (**not too much by today standards, but a lot at that time**).**
- **1990 spin-off: ARM renamed Advanced RISC Machines**
- **ARM now focuses on Embedded CPU cores**
  - IP licensing: Almost every silicon manufacturer sells some microcontroller with an ARM core. Some even compete with their own designs.
  - Processing power with low current consumption
    - Good MIPS/Watt figure
    - Ideal for portable devices
  - Compact memories: 16-bit opcodes (Thumb)
- **New cores with added features**
  - Harvard architecture (ARM9, ARM11, Cortex)
  - Floating point arithmetic
  - Vector computing (VFP, NEON)
  - Java language (Jazelle)

# Facts

- **32-bit CPU**
- **3-operand instructions (typical): ADD Rd,Rn,Operand2**
- **RISC design…**
  - Few, simple, instructions
  - Load/store architecture (instructions operate on registers, not memory)
  - Large register set
  - Pipelined execution
- **… Although with some CISC touches…**
  - *Multiplication* and *Load/Store Multiple* are complex instructions (many cycles longer than regular, RISC, instructions)
- **… And some very specific details**
  - No stack. Link register instead
  - PC as a regular register
  - Conditional execution of all instructions
  - Flags altered or not by data processing instructions (selectable)
  - Concurrent shifts/rotations (at the same time of other processing)
  - …

# Agenda

Introduction

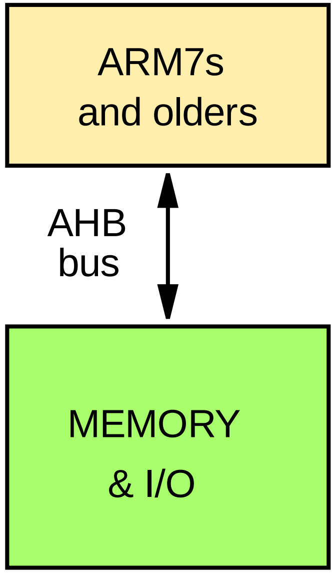
- **Architecture**

Programmers Model

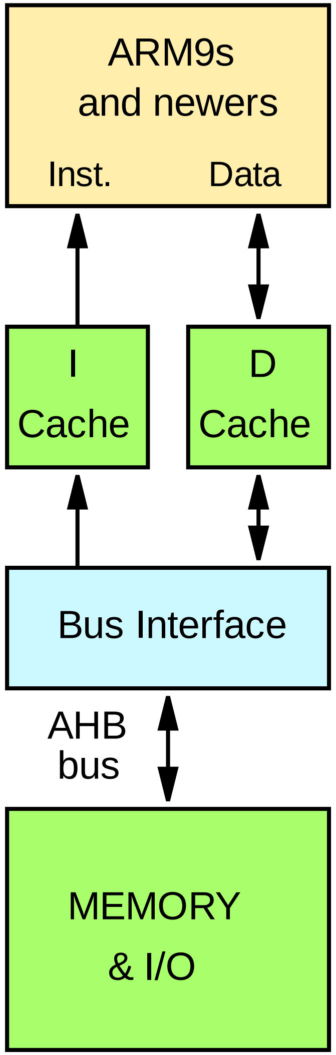
Instruction Set

# Topologies

**Von Neumann**



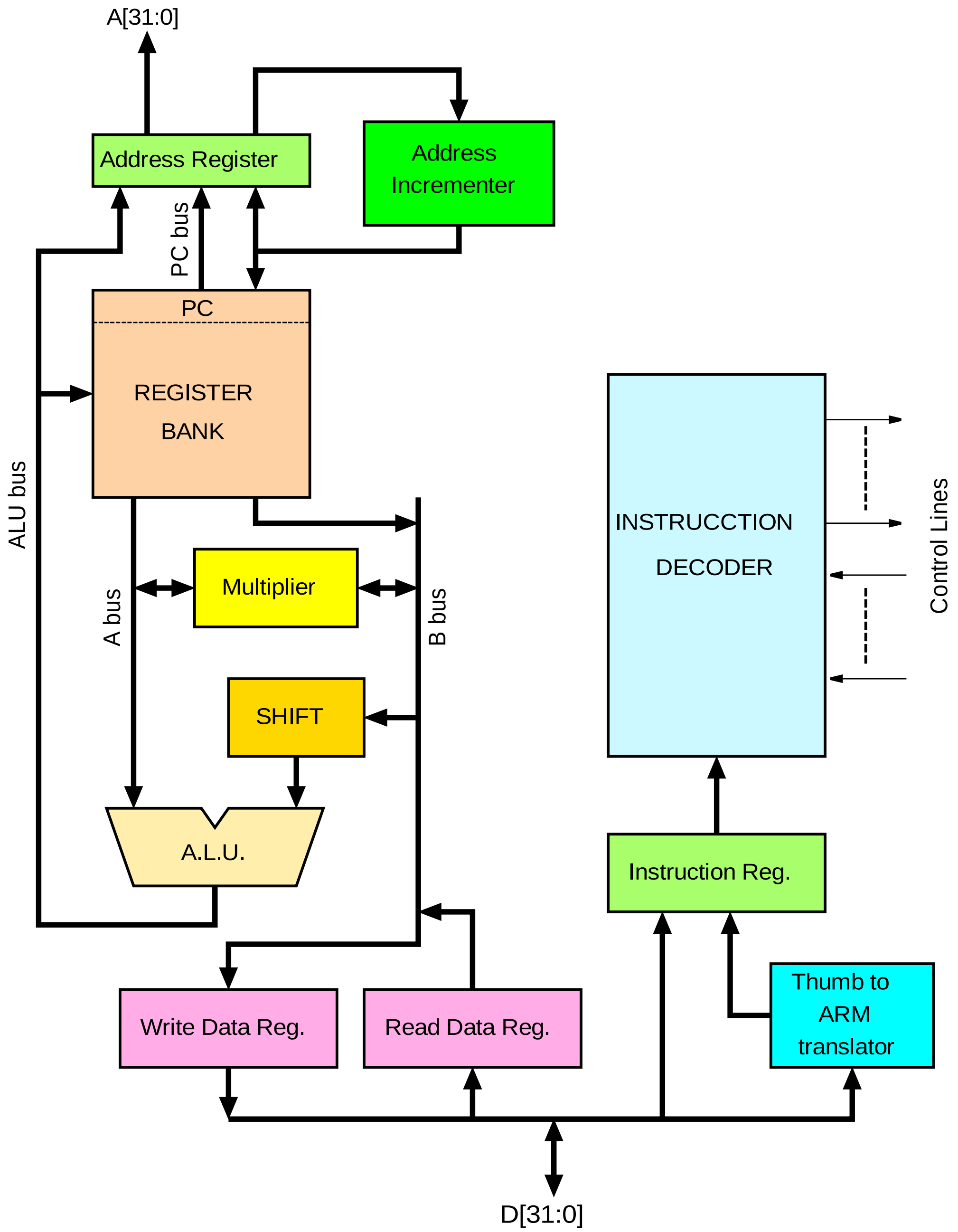
**Harvard**



## Memory-mapped I/O:

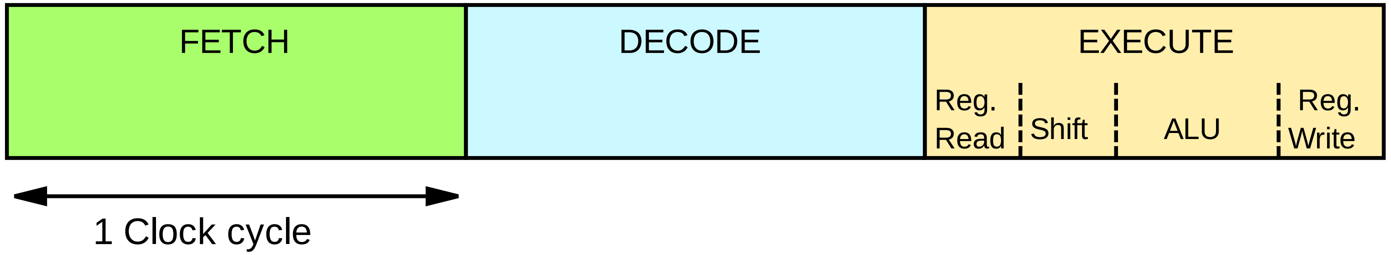
- No specific instructions for I/O (use Load/Store instr. instead)
- Peripheral’s registers at some memory addresses

# ARM7TDMI Block Diagram

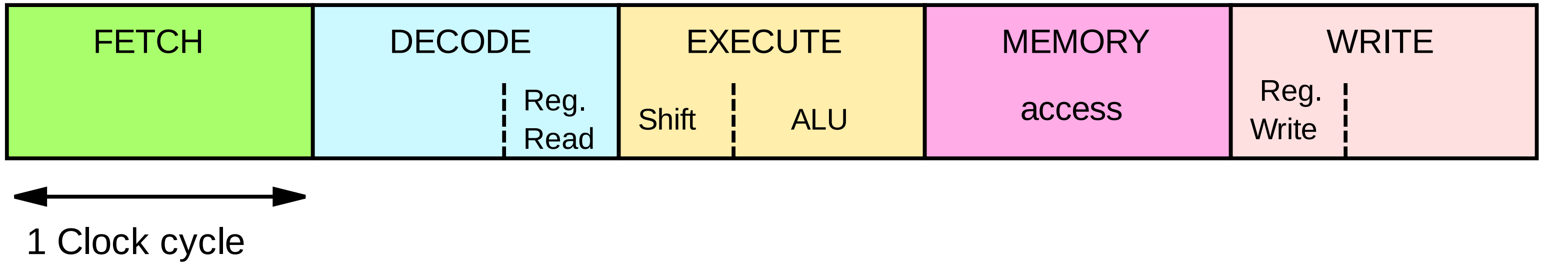


# ARM Pipelining examples

ARM7TDMI Pipeline

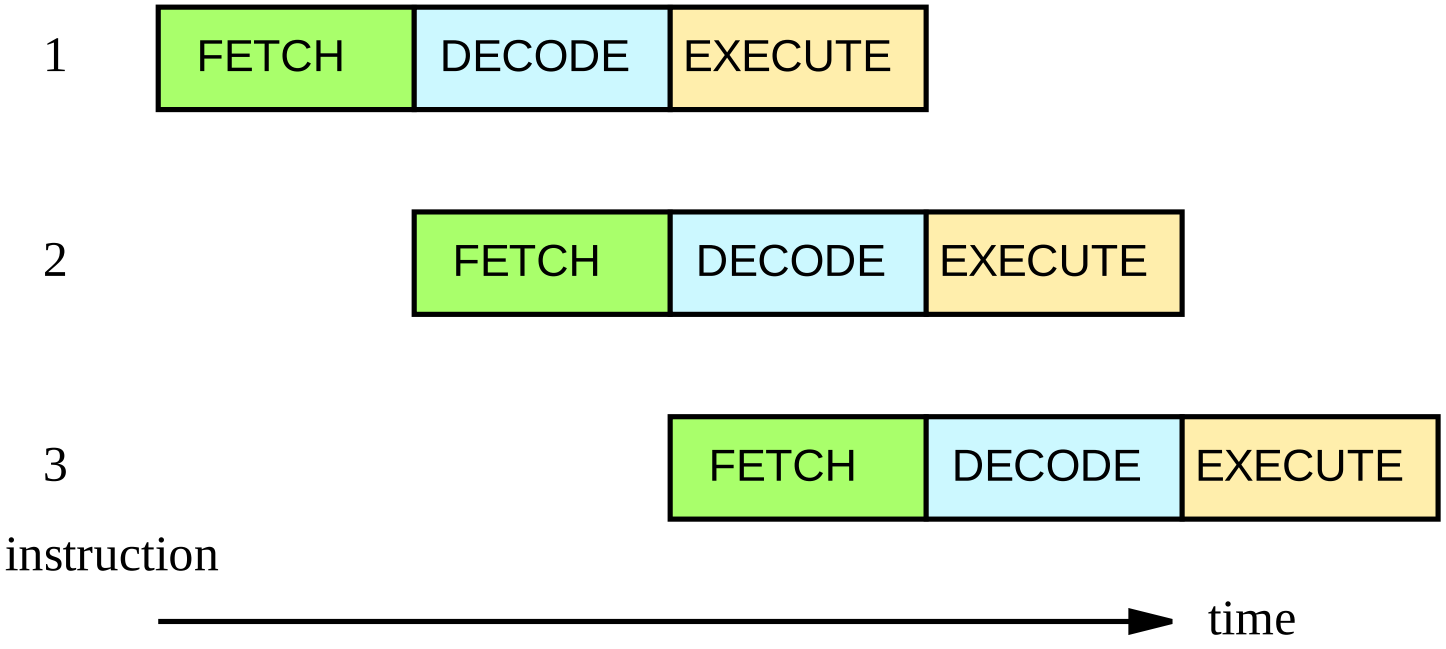


ARM9TDMI Pipeline



- Fetch: Read Op-code from memory to internal Instruction Register
- Decode: Activate the appropriate control lines depending on Opcode
- Execute: Do the actual processing

# ARM7TDMI Pipelining (I)



1 FETCH DECODE EXECUTE

2 FETCH DECODE EXECUTE

3 FETCH DECODE EXECUTE

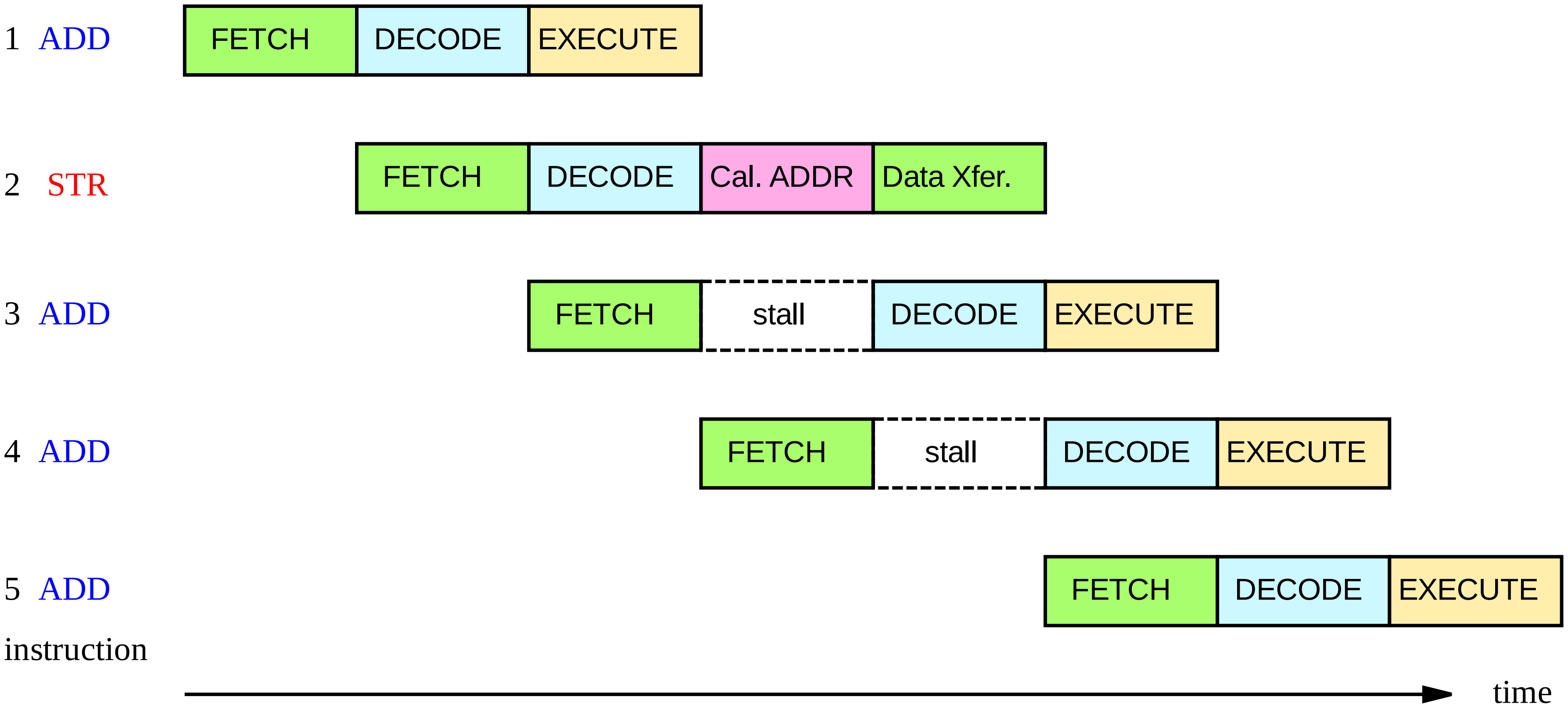
instruction

time

- Simple instructions (like ADD) Complete at a rate of one per cycle

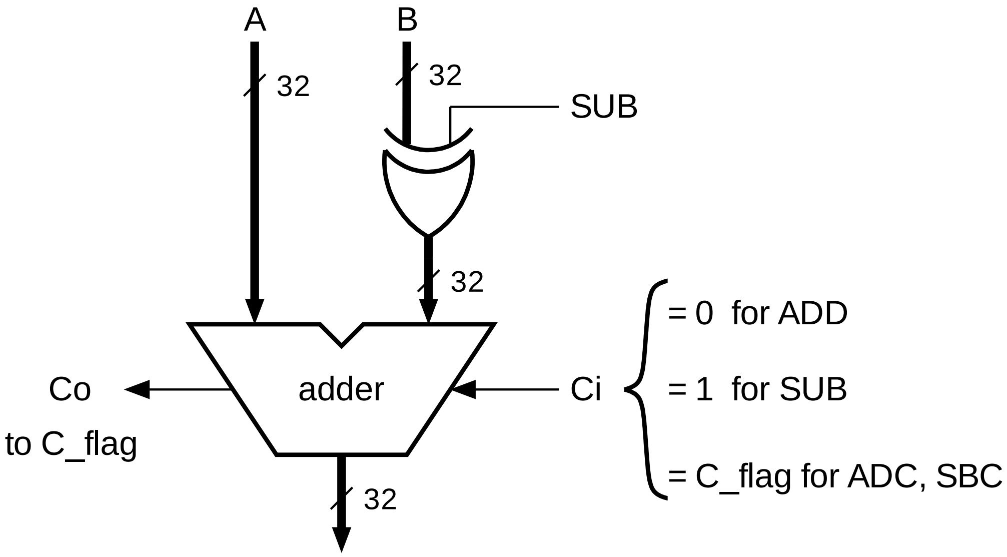
# ARM7TDMI Pipelining (II)

- More complex instructions:



STR : 2 effective clock cycles (+1 cycle)

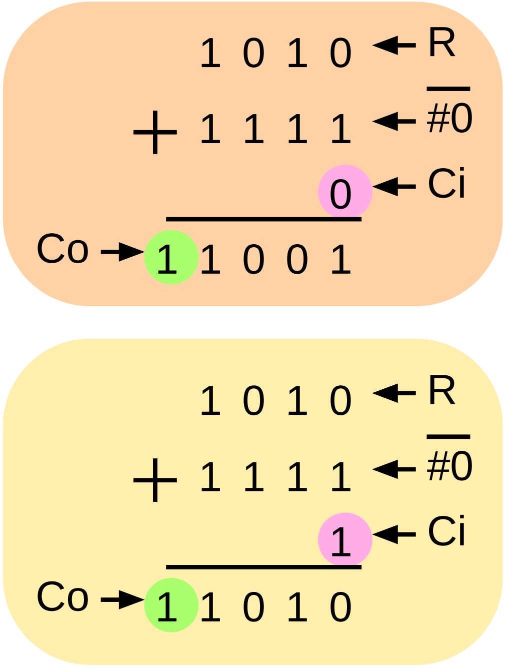
# Arithmetic and Carry Flag



ALU equivalent for arithmetic instructions

Carry flag behavior for subtraction

SBC R, #0 (4-bit examples)



Carry acts as an inverted borrow

- Same as 6502, PowerPC (Borrow = not Carry)
- In contrast with Z80, Intel x86, m68k, many others (Borrow = Carry)

# Agenda

Introduction

Architecture

- **Programmers Model**

Instruction Set

# Data Sizes and Instruction Sets

- **The ARM is a 32-bit architecture.**

- **When used in relation to the ARM:**
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)

- **Most ARM's implement two instruction sets**
  - 32-bit **ARM** Instruction Set
  - 16-bit **Thumb** Instruction Set

# Processor Modes

- **The ARM has seven operating modes:**
  - **User** : unprivileged mode under which most tasks run
  - **FIQ** : entered when a high priority (fast) interrupt is raised
  - **IRQ** : entered when a low priority (normal) interrupt is raised
  - **SVC** : (Supervisor) entered on reset and when a Software Interrupt instruction is executed
  - **Abort** : used to handle memory access violations
  - **Undef** : used to handle undefined instructions
  - **System** : privileged mode using the same registers as user mode

# The Registers

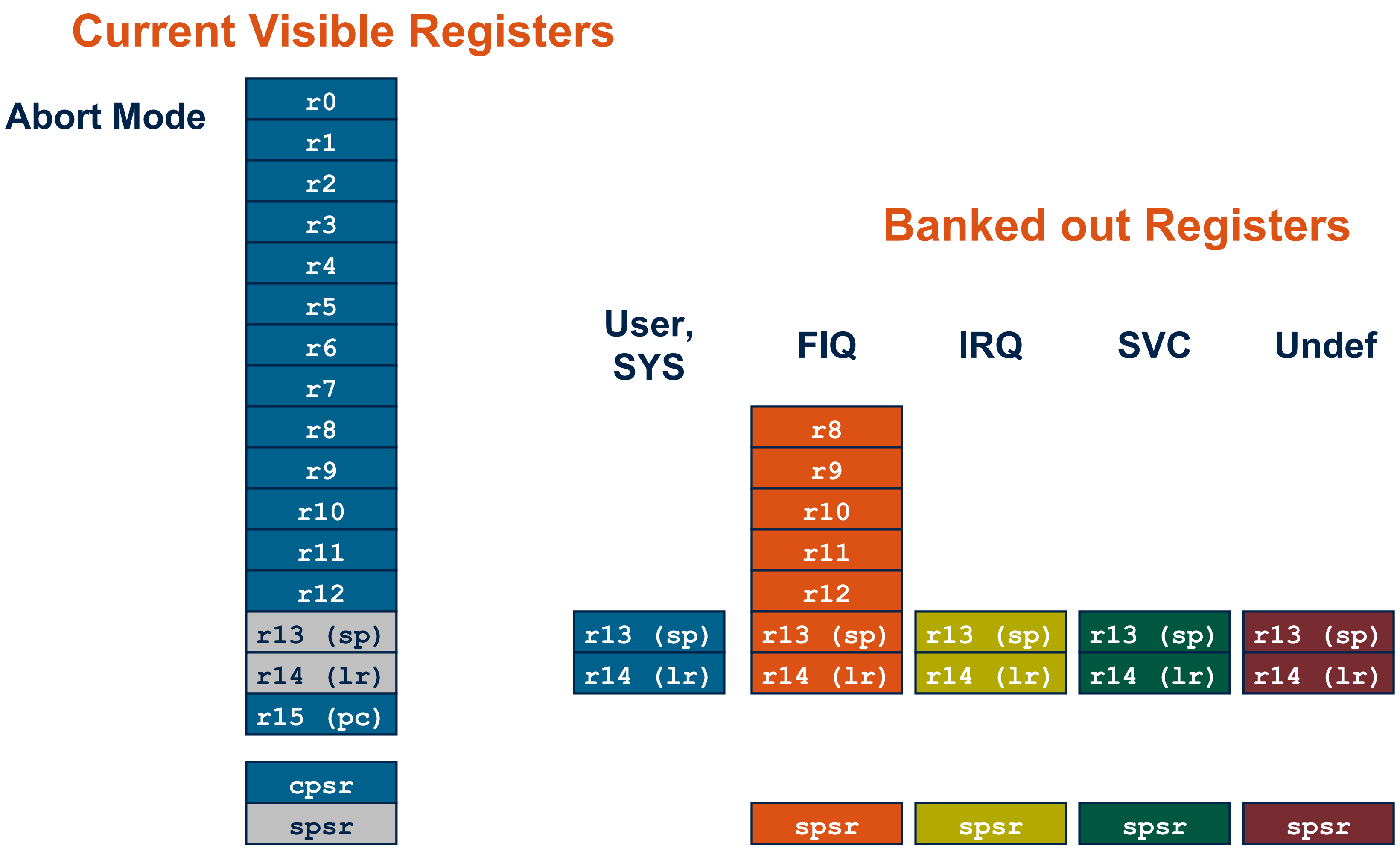
- **ARM has 37 registers all of which are 32-bits long.**
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers

- **The current processor mode governs which of several banks is accessible. Each mode can access**
  - a particular set of r0-r12 registers
  - a particular r13 (the stack pointer, sp) and r14 (the link register, lr)
  - the program counter, r15 (pc)
  - the current program status register, cpsr

**Privileged modes (except System) can also access**

- a particular spsr (saved program status register)

# The ARM Register Set



## Current Visible Registers

Abort Mode

| |
|---|
| r0 |
| r1 |
| r2 |
| r3 |
| r4 |
| r5 |
| r6 |
| r7 |
| r8 |
| r9 |
| r10 |
| r11 |
| r12 |
| r13 (sp) |
| r14 (lr) |
| r15 (pc) |
| cpsr |
| spsr |

## Banked out Registers

| User, SYS | FIQ | IRQ | SVC | Undef |
|---|---|---|---|---|
| | r8 | | | |
| | r9 | | | |
| | r10 | | | |
| | r11 | | | |
| | r12 | | | |
| r13 (sp) | r13 (sp) | r13 (sp) | r13 (sp) | r13 (sp) |
| r14 (lr) | r14 (lr) | r14 (lr) | r14 (lr) | r14 (lr) |
| | spsr | spsr | spsr | spsr |

# Special Registers

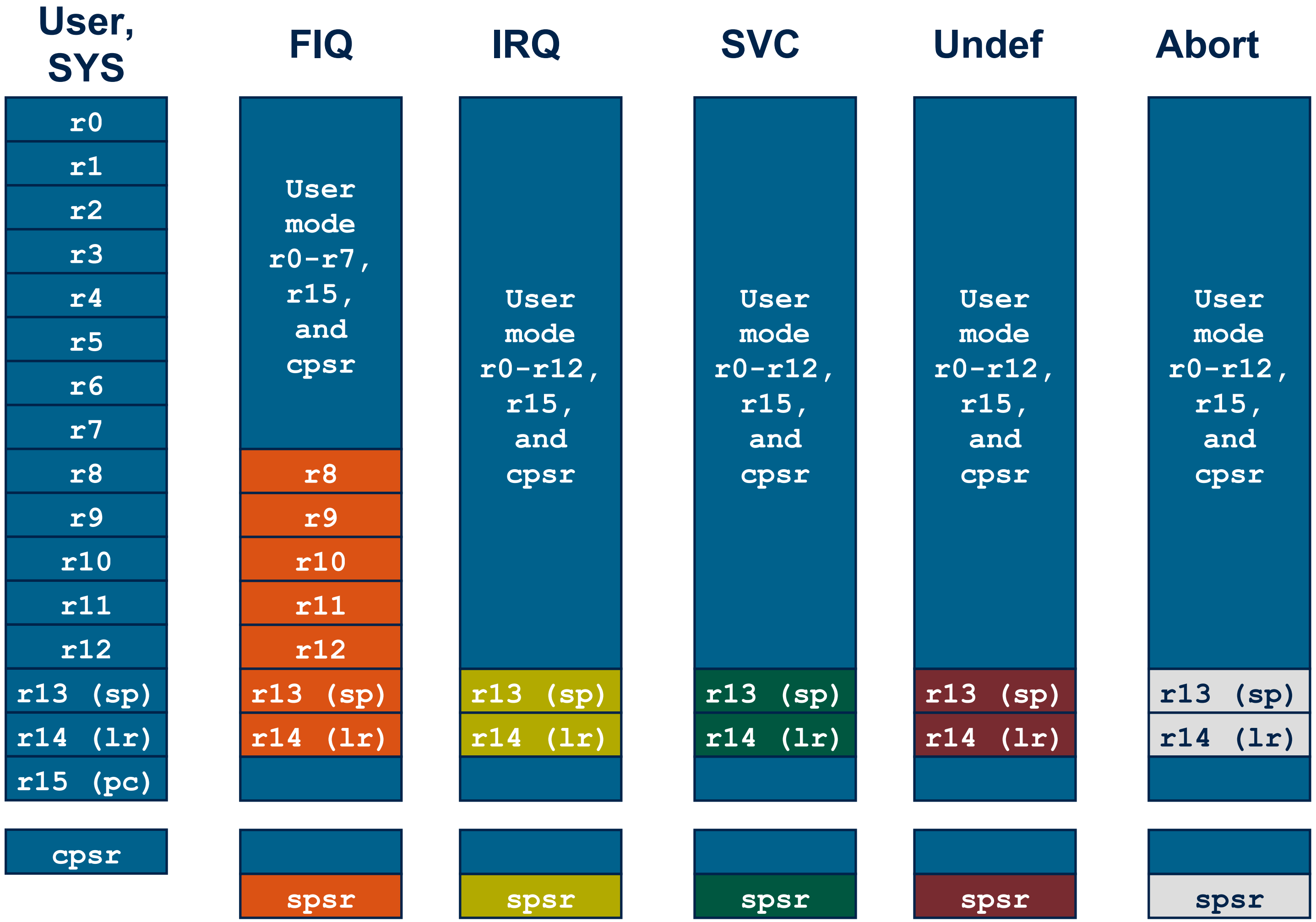
- **Special function registers:**
  - **PC** (R15): Program Counter. Any instruction with PC as its destination register is a program branch
  - **LR** (R14): Link Register. Saves a copy of PC when executing the BL instruction (subroutine call) or when jumping to an exception or interrupt routine

    - It is copied back to PC on the return from those routines
  - **SP** (R13): Stack Pointer. There is **no stack** in the ARM architecture. Even so, R13 is usually reserved as a pointer for the program-managed stack
  - **CPSR** : Current Program Status Register. Holds the visible status register
  - **SPSR** : Saved Program Status Register. Holds a copy of the previous status register while executing exception or interrupt routines

    - It is copied back to CPSR on the return from the exception or interrupt

    - No SPSR available in User or System modes

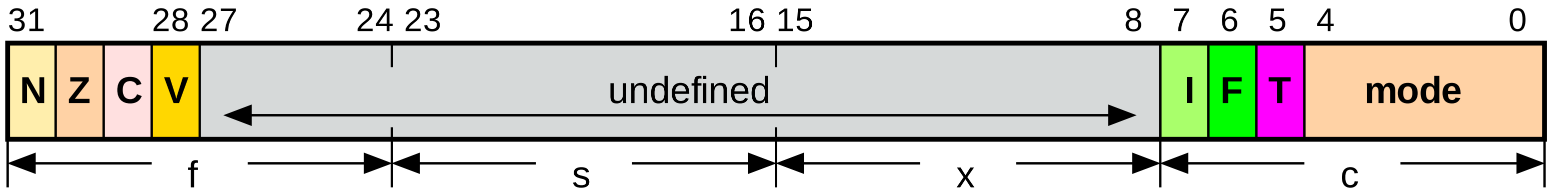
# Register Organization Summary



| User, SYS | FIQ | IRQ | SVC | Undef | Abort |
|---|---|---|---|---|---|
| r0 | User mode r0-r7, r15, and cpsr | User mode r0-r12, r15, and cpsr | User mode r0-r12, r15, and cpsr | User mode r0-r12, r15, and cpsr | User mode r0-r12, r15, and cpsr |
| r1 | | | | | |
| r2 | | | | | |
| r3 | | | | | |
| r4 | | | | | |
| r5 | | | | | |
| r6 | | | | | |
| r7 | | | | | |
| r8 | r8 | | | | |
| r9 | r9 | | | | |
| r10 | r10 | | | | |
| r11 | r11 | | | | |
| r12 | r12 | | | | |
| r13 (sp) | r13 (sp) | r13 (sp) | r13 (sp) | r13 (sp) | r13 (sp) |
| r14 (lr) | r14 (lr) | r14 (lr) | r14 (lr) | r14 (lr) | r14 (lr) |
| r15 (pc) | | | | | |
| cpsr | | | | | |
| | spsr | spsr | spsr | spsr | spsr |

Note: System mode uses the User mode register set

# Program Status Registers



- **Condition code flags**
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed

- **Mode bits**

| | |
|---|---|
| 10000 | **User** |
| 10001 | **FIQ** |
| 10010 | **IRQ** |
| 10011 | **Supervisor** |
| 10111 | **Abort** |
| 11011 | **Undefined** |
| 11111 | **System** |

Interrupt Disable bits.

I = 1: Disables the IRQ.

F = 1: Disables the FIQ.

T Bit **(Arch. with Thumb mode only)**

T = 0: Processor in ARM state

T = 1: Processor in Thumb state

**Never** change T directly (use BX instead)

Changing T in CPSR will lead to unexpected behavior due to pipelining

**Tip**: Don’t change undefined bits.

This allows for code compatibility with newer ARM processors

# Program Counter (R15)

- **When the processor is executing in ARM state:**
  - All instructions are 32 bits wide
  - All instructions must be word aligned
  - Therefore the PC value is stored in bits [31:2] and bits [1:0] are zero
  - Due to pipelining, the PC points 8 bytes ahead of the current instruction, or 12 bytes ahead if current instruction includes a register-specified shift

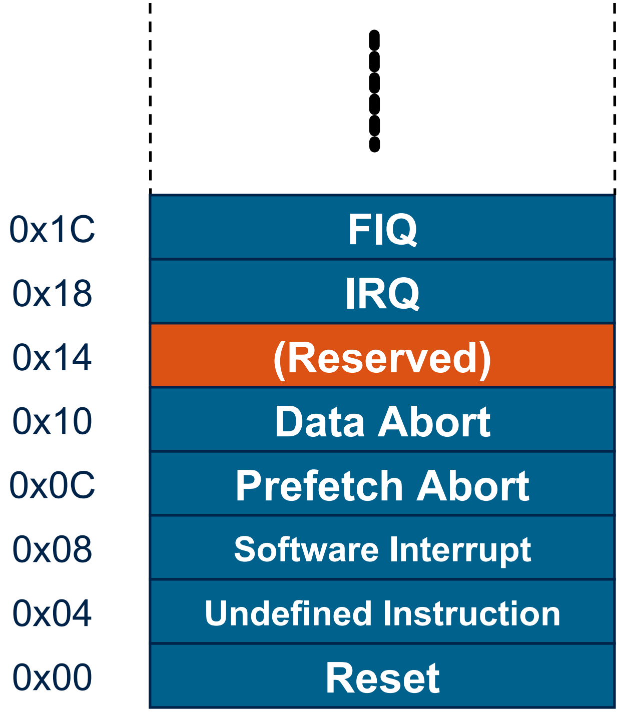
- **When the processor is executing in Thumb state:**
  - All instructions are 16 bits wide
  - All instructions must be halfword aligned
  - Therefore the PC value is stored in bits [31:1] and bit [0] is zero

# Exception Handling

- **When an exception occurs, the ARM:**
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits:
    - Changes to ARM state
    - Changes to related mode
    - Disables IRQ
    - Disables FIQ (only on fast interrupts)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- **To return, exception handler needs to:**
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

    *(more about this later…)*

**This can only be done in ARM state.**



| | |
|---|---|
| 0x1C | FIQ |
| 0x18 | IRQ |
| 0x14 | (Reserved) |
| 0x10 | Data Abort |
| 0x0C | Prefetch Abort |
| 0x08 | Software Interrupt |
| 0x04 | Undefined Instruction |
| 0x00 | Reset |

Vector Table

# Agenda

Introduction

Architecture

Programmers Model

- **Instruction Set (for ARM state)**

# Conditional Execution and Flags

- **ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.**
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
 CMP    r3,#0
  BEQ    skip
  ADD    r0,r1,r2
skip
```

```
CMP    r3,#0
 ADDNE  r0,r1,r2
```

- **By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using “S” (comparisons always set the flags).**

```
loop
  …
  SUBS r1,r1,#1
  BNE  loop
```

decrement r1 and set flags

if Z flag clear then branch

# Condition Codes

- **The 15 possible condition codes are listed below:**
  - Note AL is the default and does not need to be specified

| Suffix | Description | Flags tested |
|---|---|---|
| `EQ` | Equal | **Z=1** |
| `NE` | Not equal | **Z=0** |
| `CS/HS` | Unsigned higher or same | **C=1** |
| `CC/LO` | Unsigned lower | **C=0** |
| `MI` | Minus | **N=1** |
| `PL` | Positive or Zero | **N=0** |
| `VS` | Overflow | **V=1** |
| `VC` | No overflow | **V=0** |
| `HI` | Unsigned higher | **C=1 & Z=0** |
| `LS` | Unsigned lower or same | **C=0 or Z=1** |
| `GE` | Greater or equal | **N=V** |
| `LT` | Less than | **N!=V** |
| `GT` | Greater than | **Z=0 & N=V** |
| `LE` | Less than or equal | **Z=1 or N=!V** |
| `AL` | Always | |

# Examples of conditional execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);
    CMP       r0,#0
    MOVEQ     r0,#1
    BLEQ      func
```

- Set the flags, then use various condition codes

```
if (a==0) x=0;
if (a>0)  x=1;
    CMP       r0,#0
    MOVEQ     r1,#0
    MOVGT     r1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
    CMP       r0,#4
    CMPNE     r0,#10
    MOVEQ     r1,#0
```

# Data processing Instructions

- **Consist of :**
  - Arithmetic: ADD ADC SUB SBC RSB RSC
  - Logical: AND ORR EOR BIC
  - Comparisons: CMP CMN TST TEQ
  - Data movement: MOV MVN

- **These instructions only work on registers, NOT memory.**

| 31 28 | | 25 | 24 21 | 20 | 19 16 | 15 12 | 11 0 |
|---|---|---|---|---|---|---|---|
| cond. | 0 0 | L | op-code | S | Rn | Rd | Operand 2 |

L, Literal: 0: Operand 2 from register, 1: Operand 2 immediate

- **Syntax:**

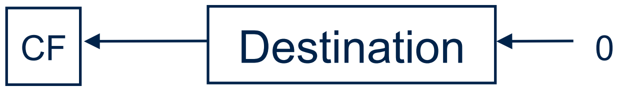
  `<Operation>{<cond>}{S} Rd, Rn, Operand2`

  - {S} means that the Status register is going to be updated
  - Comparisons always update the status register. Rd is not specified
  - Data movement does not specify Rn

- **Second operand is sent to the ALU via barrel shifter.**

# The Barrel Shifter

## LSL : Logical Left Shift



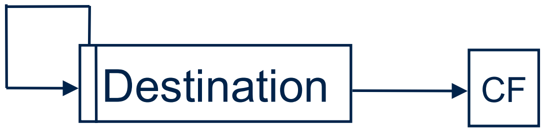
CF

Destination

0

Multiplication by a power of 2

## ASR: Arithmetic Right Shift

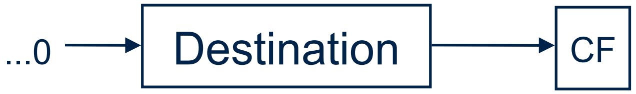


Destination

CF

Division by a power of 2,
preserving the sign bit

## LSR : Logical Shift Right



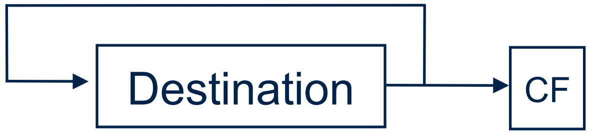
...0

Destination

CF

Division by a power of 2

## ROR: Rotate Right

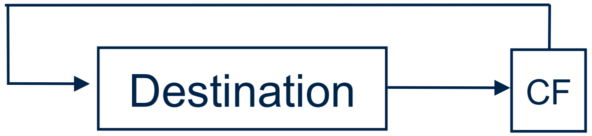


Destination

CF

Bit rotate with wrap around
from LSB to MSB

## RRX: Rotate Right Extended

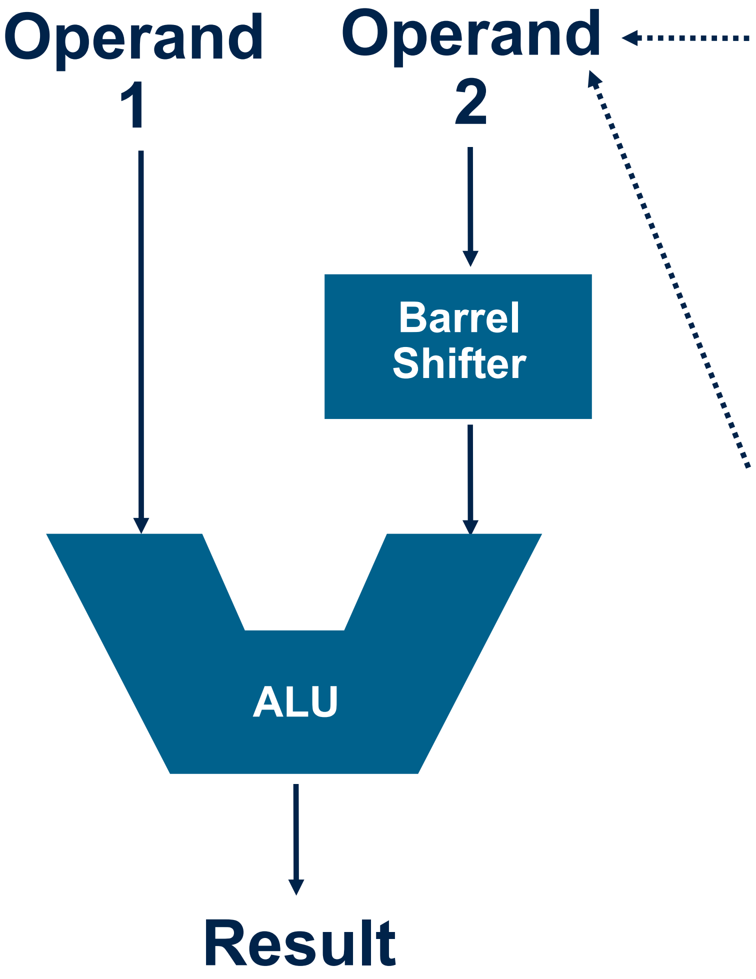


Destination

CF

Single bit rotate with wrap around
from CF to MSB

# Using the Barrel Shifter: The Second Operand



## Register, optionally with shift operation

- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by a power of 2

Example: ADD R1, R2, R3, LSL #2

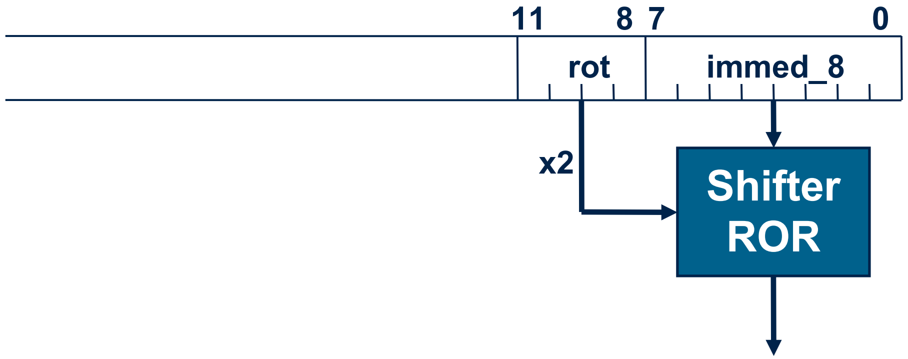
(R2 + R3*4) -> R1

## Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers

# Immediate constants (1)

- **No ARM instruction can contain a 32 bit immediate constant**
  - All ARM instructions are fixed as 32 bits long
- **The data processing instruction format has 12 bits available for operand2**



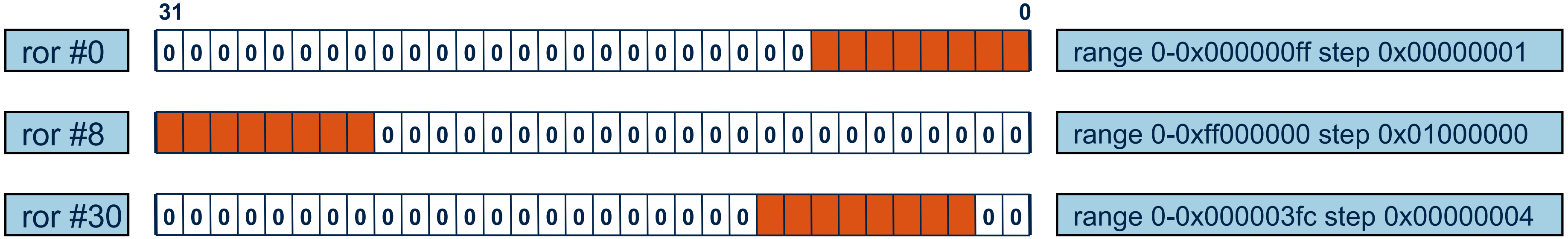
**Quick Quiz:**

```
0xe3a004ff
MOV r0, #???
```

- **4 bit rotate value (0-15) is multiplied by two to give range 0-30 in steps of 2**
- **Rule to remember is "8-bits shifted by an even number of bit positions".**

# Immediate constants (2)

- **Examples:**



ror #0 — range 0-0x000000ff step 0x00000001

ror #8 — range 0-0xff000000 step 0x01000000

ror #30 — range 0-0x000003fc step 0x00000004

- **The assembler converts immediate values to the rotate form:**
  - `MOV r0,#4096          ; uses 0x40 ror 26`
  - `ADD r1,r2,#0xFF0000      ; uses 0xFF ror 16`

- **The bitwise complements can also be formed using MVN:**
  - `MOV r0, #0xFFFFFFFF      ; assembles to MVN r0,#0`

- **Values that cannot be generated in this way will cause an error.**

# Loading 32 bit constants

- **To allow larger constants to be loaded, the assembler offers a pseudo-instruction:**
  - `LDR rd, =const` (notice the “=“ sign)
- **This will either:**
  - Produce a **`MOV`** or **`MVN`** instruction to generate the value (if possible).

**or**

  - Generate a **`LDR`** instruction with a PC-relative address to read the constant from a *literal pool* (Constant data area embedded in the code).
- **For example**
  - `LDR r0,=0xFF          =>       MOV r0,#0xFF`
  - `LDR r0,=0x55555555    =>       LDR r0,[PC,#Imm12]`

```
                                        ...
                                        ...
                                        DCD 0x55555555
```

- **This is the recommended way of loading constants into a register**

# Loading addresses: ADR

- The Assembler includes the pseudo-instruction ADR, intended to load an address into a register

  **ADR Rd, label**

- ADR will be translated into a data processing instruction which uses PC as the source operand
- For example:

```
        .text
        .arm
        .globl  _start

_start: mov     r0,#1
        adr     r1,msg1
        mov     r2,#12
        swi     0x900004
        swi     0x900001

msg1:   .ascii "Hello World\n"
```

Note: PC is 8 bytes ahead of the current instruction (pipelining)

```
8074:   e3a00001   mov r0, #1
8078:   e28f1008   add r1, pc, #8
807c:   e3a0200c   mov r2, #12
8080:   ef900004   swi 0x00900004
8084:   ef900001   swi 0x00900001
8088:   6c6c6548
808c:   6f57206f
8090:   0a646c72
```

# Data processing instr. FLAGS

- **Flags are changed only if the S bit of the op-code is set:**

  Mnemonics ending with “**s**”, like “movs”, and comparisons: **cmp**, **cmn**, **tst**, **teq**

- **N and Z have the expected meaning for all instructions**
  - **N**: bit 31 (sign) of the result
  - **Z**: set if result is zero
- **Logical instructions (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN)**
  - **V**: unchanged
  - **C**: from **barrel shifter** if shift ≠ 0. Unchanged otherwise
- **Arithmetic instructions (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN)**
  - **V**: Signed overflow from **ALU**
  - **C**: Carry (bit 32 of result) from **ALU**
- **When PC is the destination register** (exception return)
  - CPSR is copied from SPSR. This includes all the flags.
  - No change in **user** or **system** modes

  Example: SUBS PC,LR,#4 @ return from IRQ

# Multiply

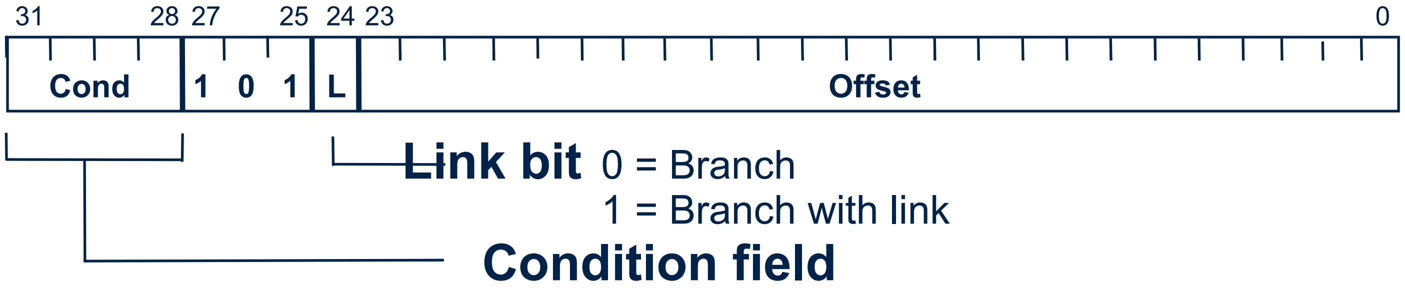
- **Syntax:**
  - MUL{<cond>}{S} Rd, Rm, Rs		Rd = Rm * Rs
  - MLA{<cond>}{S} Rd,Rm,Rs,Rn		Rd = (Rm * Rs) + Rn
  - [U|S]MULL{<cond>}{S} RdLo, RdHi, Rm, Rs		RdHi,RdLo := Rm*Rs
  - [U|S]MLAL{<cond>}{S} RdLo, RdHi, Rm, Rs		RdHi,RdLo:=(Rm*Rs)+RdHi,RdLo

- **Cycle time**
  - Basic MUL instruction
    - 2-5 cycles on ARM7TDMI
    - 1-3 cycles on StrongARM/XScale
    - 2 cycles on ARM9E/ARM102xE
  - +1 cycle for ARM9TDMI (over ARM7TDMI)
  - +1 cycle for accumulate (not on 9E though result delay is one cycle longer)
  - +1 cycle for “long”

- **Above are “general rules” - refer to the TRM for the core you are using for the exact details**

# Branch instructions

- **Branch :** `B{<cond>} label`
- **Branch with Link :** `BL{<cond>} subroutine_label`



31 28 27 25 24 23 0

| Cond | 1 0 1 | L | Offset |
|---|---|---|---|

**Link bit** 0 = Branch
1 = Branch with link

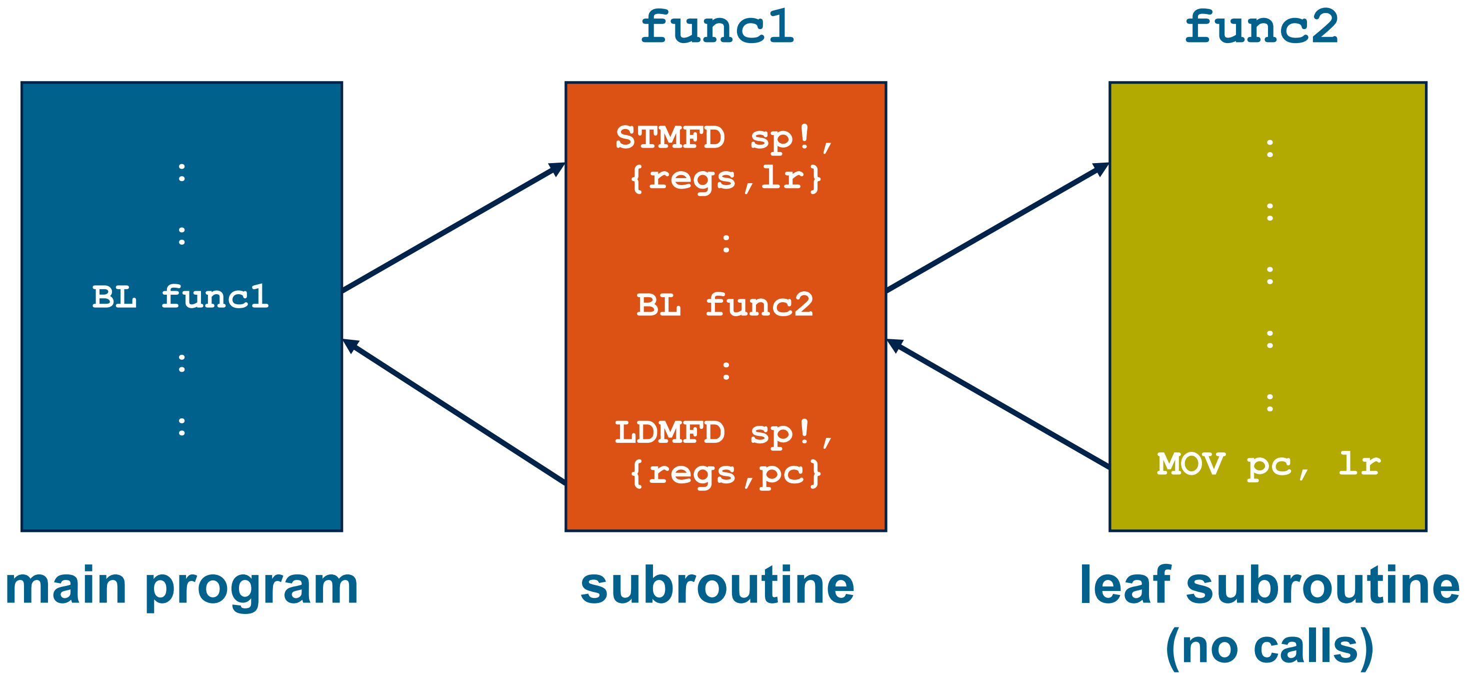
**Condition field**

- **The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC**
  - ± 32 Mbyte range
  - How to perform longer branches or absolute address branches?

    solution: LDR PC,…

# ARM Branches and Subroutines

- **BL <subroutine>**
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR
  - For non-leaf subroutines, LR will have to be stacked



func1

func2

```
:
:
BL  func1
:
:
```

```
STMFD sp!,
{regs,lr}
:
BL func2
:
LDMFD sp!,
{regs,pc}
```

```
:
:
:
:
:
MOV pc, lr
```

main program

subroutine

leaf subroutine
(no calls)

# Single register data transfer

| | | |
|---|---|---|
| `LDR` | `STR` | Word |
| `LDRB` | `STRB` | Byte |
| `LDRH` | `STRH` | Halfword |
| `LDRSB` | Signed byte load | |
| `LDRSH` | Signed halfword load | |

- **Memory system must support all access sizes**

- **Syntax:**
  - `LDR`{<cond>}{<size>} Rd, <address>
  - `STR`{<cond>}{<size>} Rd, <address>

e.g. `LDREQB`

ARM

# Address accessed

- **Address accessed by LDR/STR is specified by a base register plus an offset**
- **For word and unsigned byte accesses, offset can be**
  - An unsigned 12-bit immediate value (ie 0 - 4095 bytes).

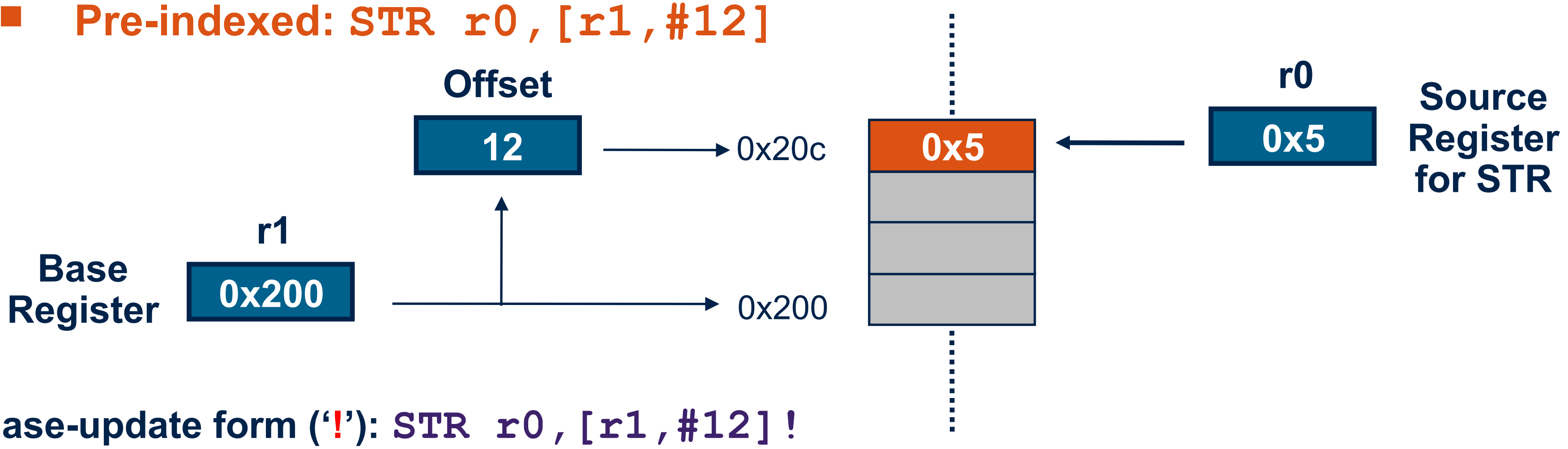
    ```
    LDR r0,[r1,#8]
    ```
  - A register, optionally shifted by an immediate value

    ```
    LDR r0,[r1,r2]
    LDR r0,[r1,r2,LSL#2]
    ```
- **This can be either added or subtracted from the base register:**

  ```
  LDR r0,[r1,#-8]
  LDR r0,[r1,-r2]
  LDR r0,[r1,-r2,LSL#2]
  ```
- **For halfword and signed halfword / byte, offset can be:**
  - An unsigned 8 bit immediate value (ie 0-255 bytes).
  - A register (unshifted).
- **Choice of *pre-indexed* or *post-indexed* addressing**

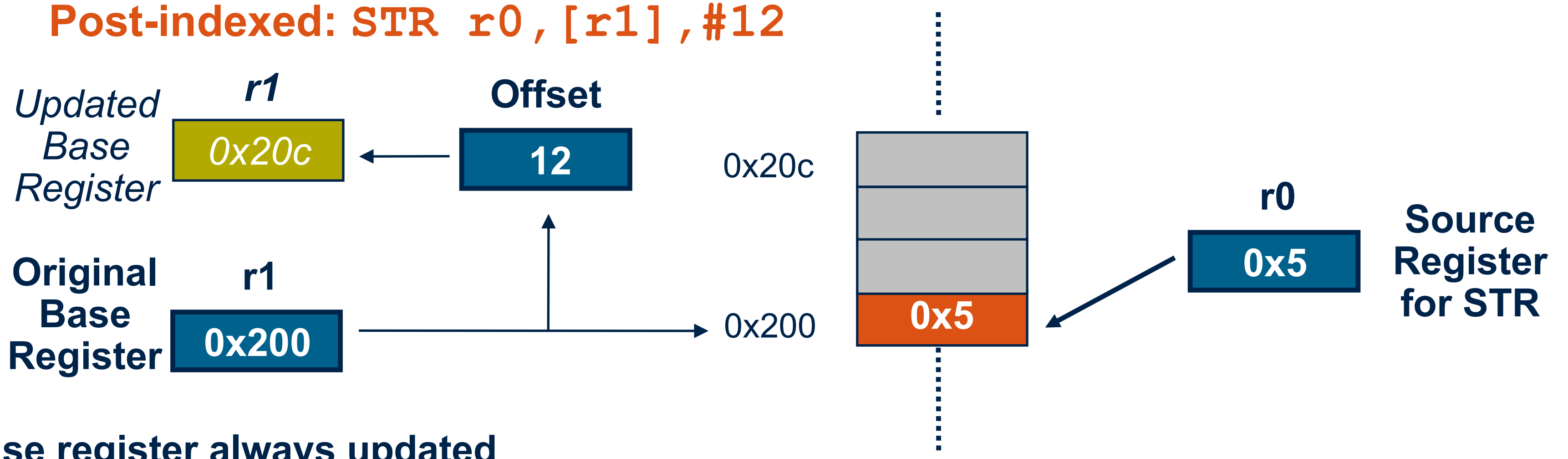
# Pre or Post Indexed Addressing?

- **Pre-indexed: `STR r0,[r1,#12]`**



Base-update form (‘!’): `STR r0,[r1,#12]!`

- **Post-indexed: `STR r0,[r1],#12`**



Base register always updated

# LDM / STM operation

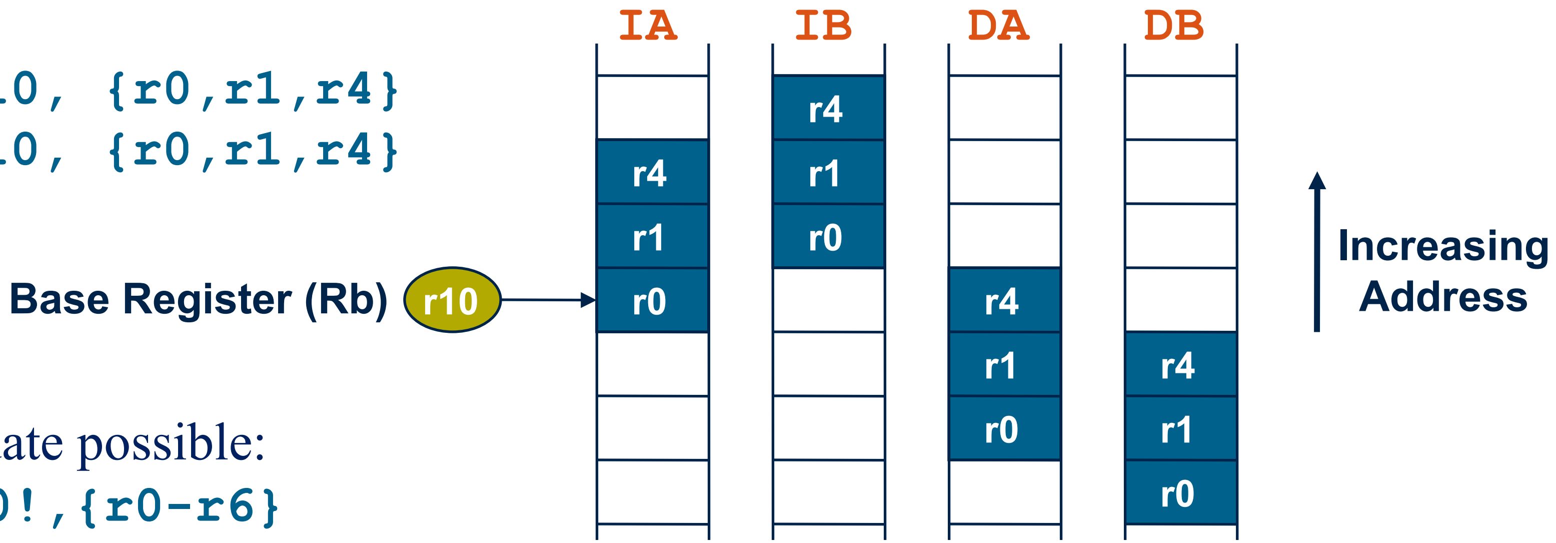
- **Load/Store Multiple Syntax:**

  `<LDM|STM>`{<cond>}<addressing_mode> Rb{!}, <register list>

- **4 addressing modes:**

| | |
|---|---|
| `LDMIA / STMIA` | increment after |
| `LDMIB / STMIB` | increment before |
| `LDMDA / STMDA` | decrement after |
| `LDMDB / STMDB` | decrement before |

```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```



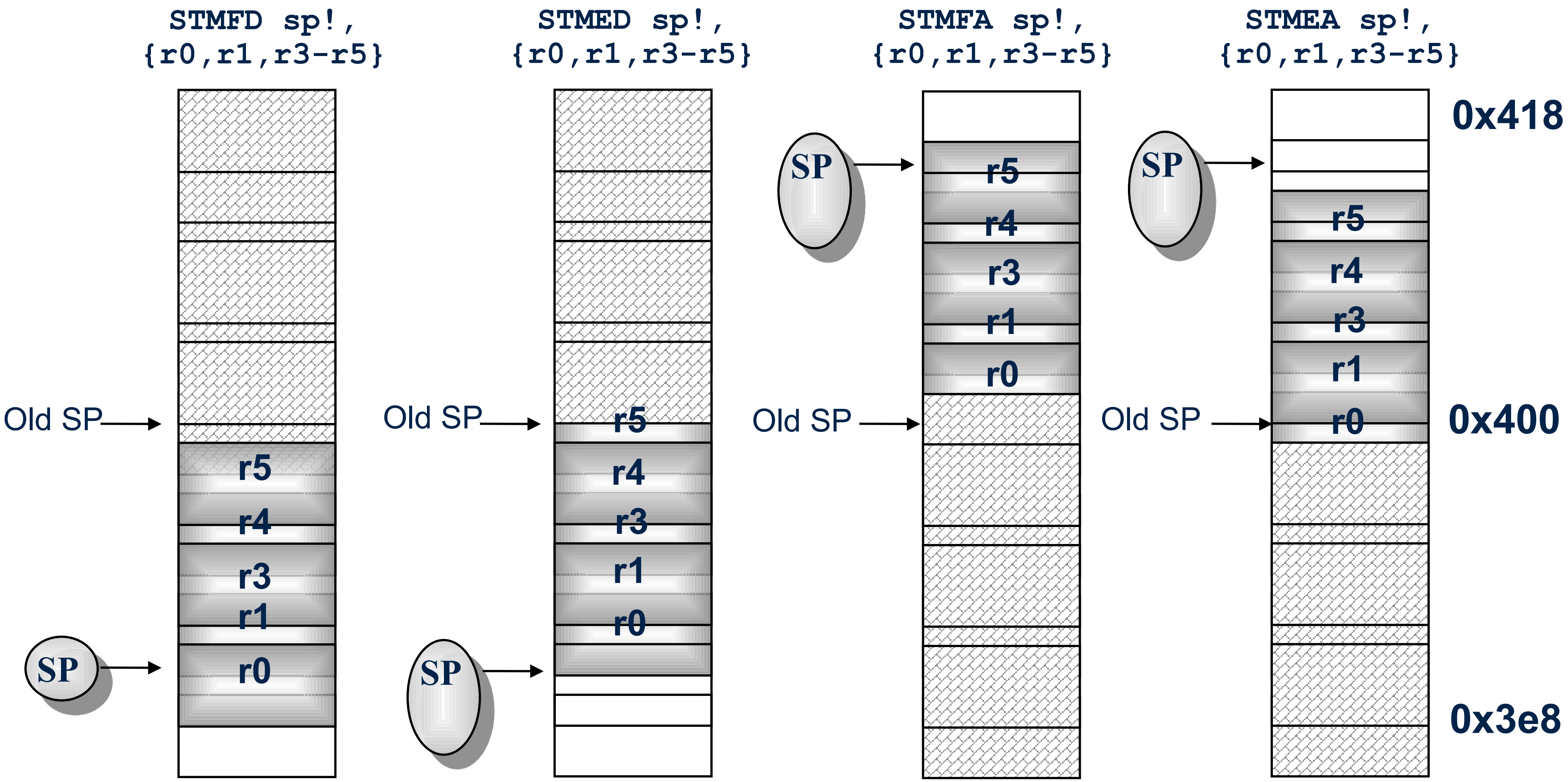
Base-update possible:

```
LDM r10!,{r0-r6}
```

# LDM/STM for Stack Operations

- **Traditionally, a stack grows down in memory, with the last "pushed" value at the lowest address. The ARM also supports ascending stacks, where the stack structure grows up through memory.**
- **The value of the stack pointer can either:**
  - Point to the last occupied address (Full stack)
    - and so needs pre-decrementing/incrementing (ie before the push)
  - Point to an unoccupied address (Empty stack)
    - and so needs post-decrementing/incrementing (ie after the push)
- **The stack type to be used is given by the postfix to the instruction:**
  - STMFD / LDMFD : Full Descending stack
  - STMFA / LDMFA : Full Ascending stack.
  - STMED / LDMED : Empty Descending stack
  - STMEA / LDMEA : Empty Ascending stack
- **Note: ARM Compilers will always use a Full descending stack.**

# Stack Examples



# LDM/STM Alias Names

- STMIA, STMIB, STMDA, STMDB are the same instructions as STMEA, STMFA, STMED, STMFD, respectively
- LDMIA, LDMIB, LDMDA, LDMDB are also the same instructions as LDMFD, LDMED, LDMFA, LDMEA, respectively
- The later names are useful when working with stacks

# LDM/STM: ^ modifier

- **The ^ modifier changes the behavior of LDM and STM. There are 2 cases:**

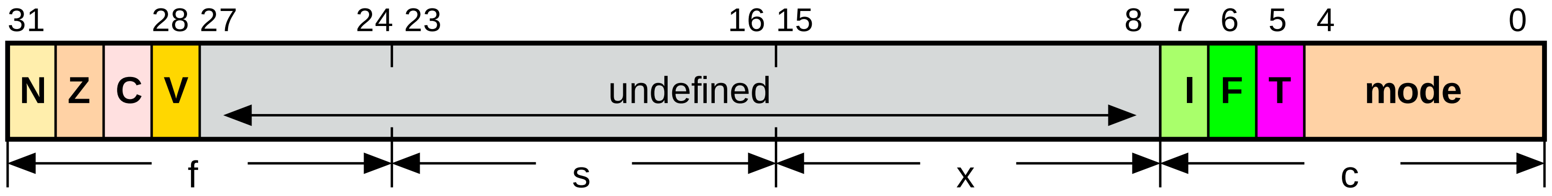
- **If the PC is not included in the register list:**
  - A '^' specifies a transfer to/from the user register bank
  - Used in exception handlers to inspect/modify the user mode registers

```
Example:  stmia  r0,{sp,lr}^  @ Transfer SP_user and LR_user to memory
          ldr    r1,[r0]      @ R1=SP_user
          ldr    r2,[r0,#4]   @ R2=LR_user
```

- **If the PC is included in the register list (LDM only):**
  - The SPSR is copied to CPSR
  - Appropriate for exception return

```
Example:  ldmfd  sp!, {r4-r7,pc}^    @ return from SWI
```

# PSR Transfer Instructions



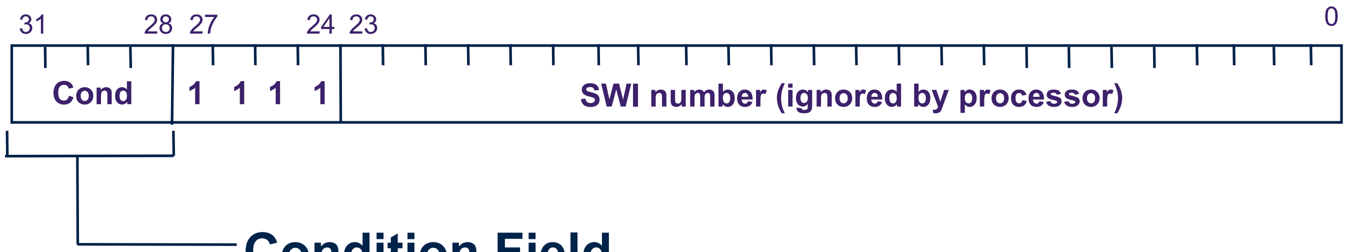
- MRS and MSR allow contents of CPSR / SPSR to be transferred to / from a general purpose register.
- Syntax:
  - `MRS{<cond>} Rd,<psr>              ; Rd = <psr>`
  - `MSR{<cond>} <psr[_fields]>,Rm ; <psr[_fields]> = Rm`

where

- `<psr> = CPSR or SPSR`
- `[_fields] = any combination of 'fsxc'`

- Also an immediate form
  - `MSR{<cond>} <psr_fields>,#Immediate`
- In User Mode, all bits can be read but only the condition flags (_f) can be written.

# Software Interrupt (SWI)



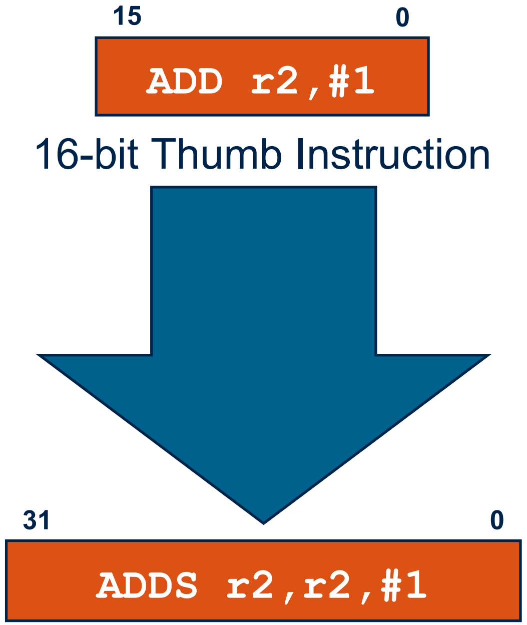
| 31 28 | 27 24 | 23 0 |
|---|---|---|
| Cond | 1 1 1 1 | SWI number (ignored by processor) |

Condition Field

- Causes an exception trap to the SWI hardware vector
- The SWI handler can examine the SWI number to decide what operation has been requested.
- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request (System Calls).
- Syntax:
  - `SWI{<cond>} #<SWI number>`

# Thumb State

- **Thumb is a 16-bit instruction set**
  - Optimized for code density from C code (~65% of ARM code size)
  - Improved performance from memory with a narrow data bus
  - Subset of the functionality of the ARM instruction set
- **Core has additional execution state - Thumb**
  - Switch between ARM and Thumb via the `BX Rn` instruction (Branch and eXchange). If Rn.0 is 1 (odd address) the processor will change to thumb state.



16-bit Thumb Instruction

32-bit ARM Instruction

**Thumb instruction set limitations:**

- Conditional execution only for branches
- Source and destination registers identical
- Only Low registers (R0-R7) used
- Constants are of limited size
- Inline barrel shifter not used
- No MSR, MRS instructions

# Atomic data swap

- Exchanges a word or byte between a register and a memory location
- This operation cannot be interrupted, not even by DMA
- Main use: Operating System semaphores
- Syntax:
  - `SWP {<cond>} Rd, Rm, [Rn]`
  - `SWPB{<cond>} Rd, Rm, [Rn]`

Rd=[Rn]; [Rn]=Rm (Rd and Rm can be the same)

# Exception / Interrupt Return

- **How to restore CPSR from SPCR?**
  - Data processing instruction with S-bit set (update status) and PC as the destination register:
    - **MOVS pc, lr**
    - **SUBS pc, lr, #4**
  - Load Multiple, restoring PC from a stack, and with the special qualifier ‘**^**’:
    - **LDMFD sp!, {r0-r12, pc}^**

- **Different return for each exception/interrupt:**

| | | | |
|---|---|---|---|
| SWI: | MOVS pc, lr | UNDEF: | MOVS pc, lr |
| FIQ: | SUBS pc, lr, #4 | IRQ: | SUBS pc, lr, #4 |
| Prefetch Abort: | SUBS pc, lr, #4 | Data Abort: | SUBS pc, lr, #8 |

# Coprocessors

- **Coprocessor instructions:**
  - Coprocessor data operation: **CDP**
  - Coprocessor Load/Store: **LDC**, **STC**
  - Coprocessor register transfer: **MRC**, **MCR**

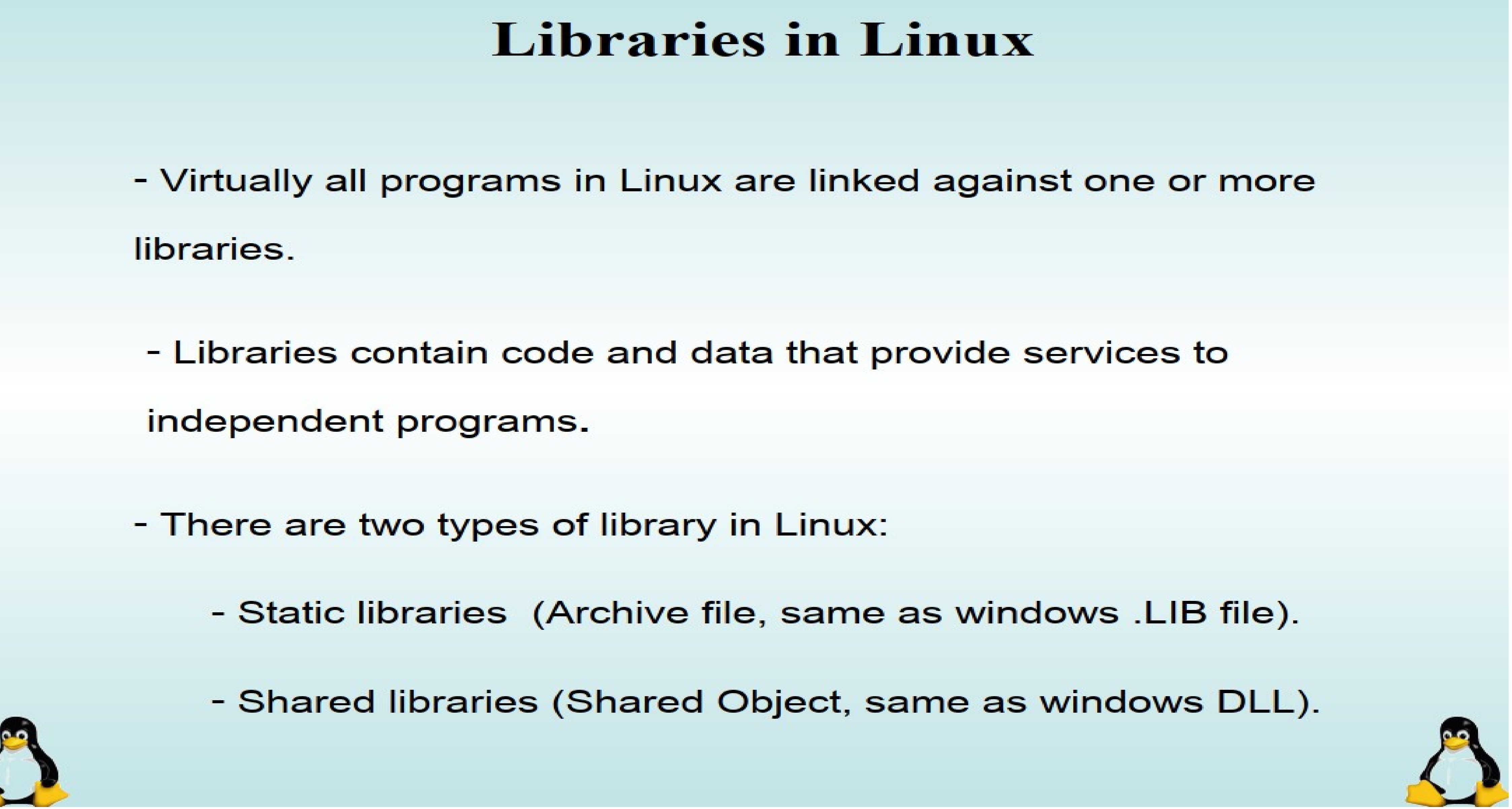
  (some coprocessors, like P14 and P15, only support MRC and MCR)
- **A 4-bit coprocessor number (P**xx**) has to be specified in these instructions.**
- **Result in UNDEF exceptions if coprocessor is missing**
- **The most common coprocessors:**
  - P15: System control (cache, MMU, …)
  - P14: Debug (Debug Communication Channel)
  - P1, P4, P10: Floating point (FPA, FPE, Maverick, VFP, …)
- **The assembler can translate the floating-point mnemonics into coprocessor instructions.**

# Arm Assembly – Statup Code

```
# qemu-system-arm -M versatilepb -m 128M -nographic -kernel test.bin
```

```
# qemu-system-arm -M versatilepb -m 128M -nographic -s -S -kernel test.bin
```

# Linux Libraries



## Libraries in Linux

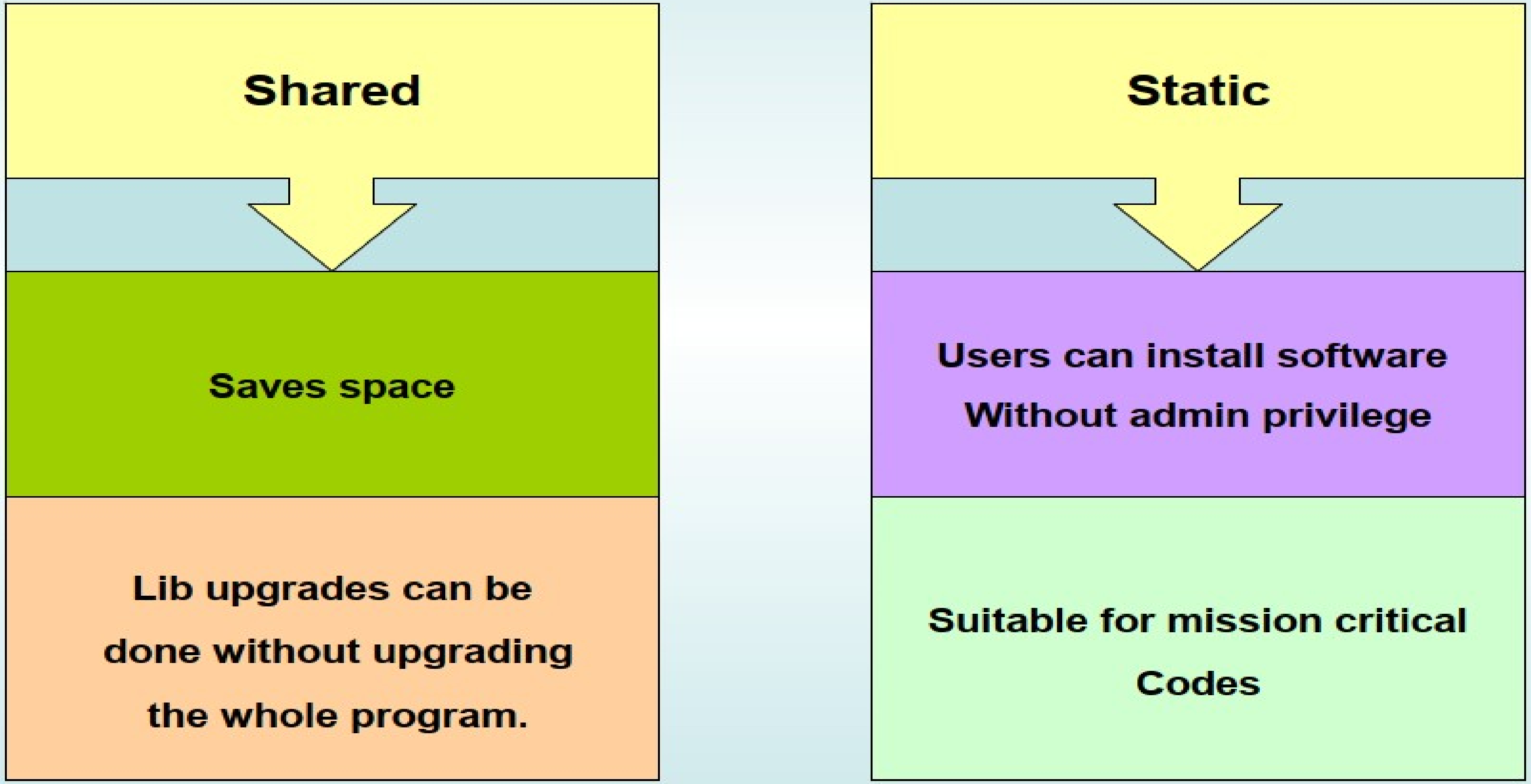
- Virtually all programs in Linux are linked against one or more libraries.

- Libraries contain code and data that provide services to independent programs.

- There are two types of library in Linux:
    - Static libraries  (Archive file, same as windows .LIB file).
    - Shared libraries (Shared Object, same as windows DLL).

# Linux Libraries

## Shared vs. Static



### Shared

Saves space

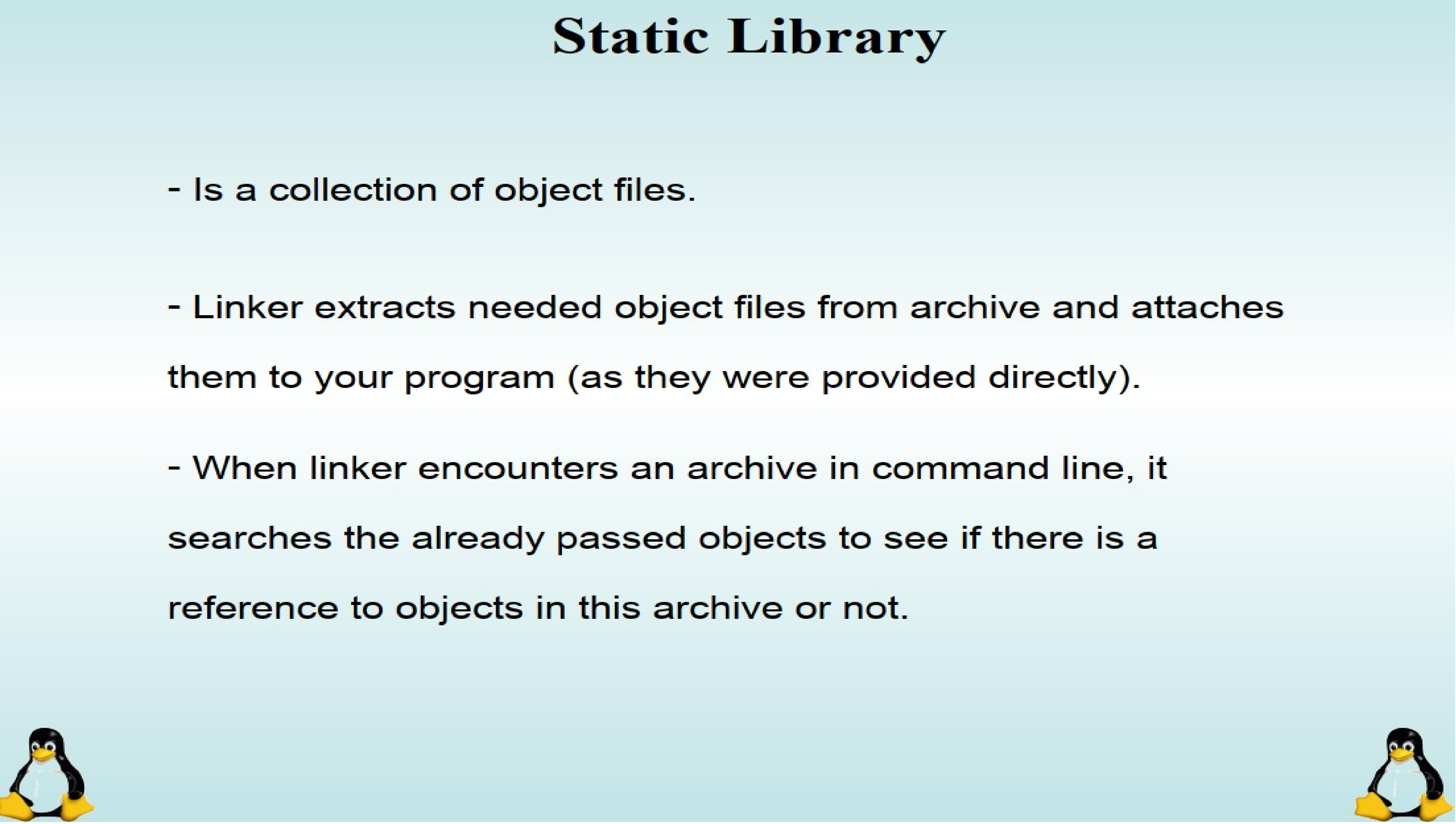
Lib upgrades can be done without upgrading the whole program.

### Static

Users can install software Without admin privilege

Suitable for mission critical Codes

# Static Libraries



## Static Library

- Is a collection of object files.

- Linker extracts needed object files from archive and attaches them to your program (as they were provided directly).

- When linker encounters an archive in command line, it searches the already passed objects to see if there is a reference to objects in this archive or not.

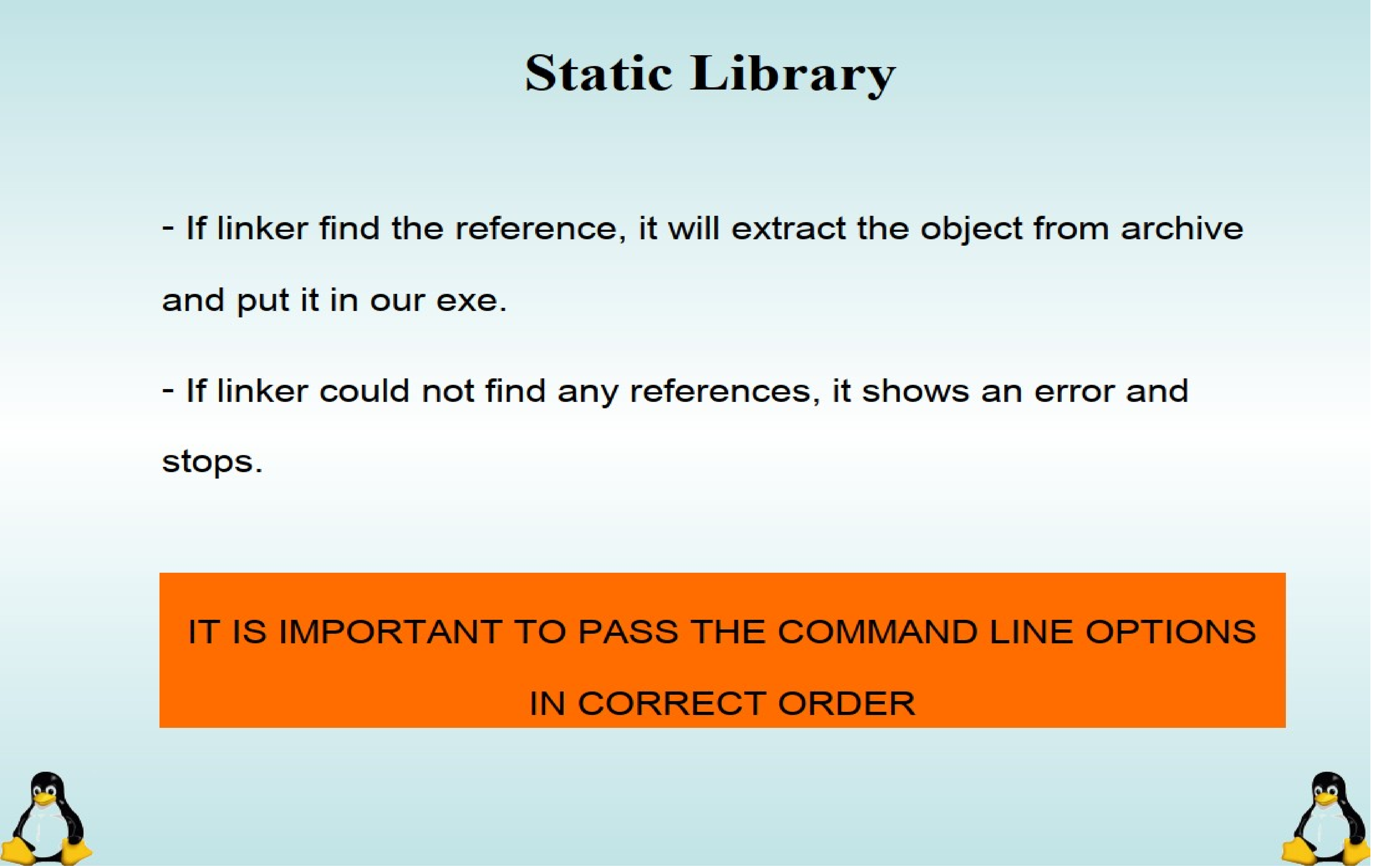
# Static Library

```
ar cr <libraryname> <list of object files>
```

**Example:**

```
ar cr libfuncs.a obj1.o obj2.o
```

# Static Libraries



## Static Library

- If linker find the reference, it will extract the object from archive and put it in our exe.

- If linker could not find any references, it shows an error and stops.

**IT IS IMPORTANT TO PASS THE COMMAND LINE OPTIONS IN CORRECT ORDER**

# Static Library

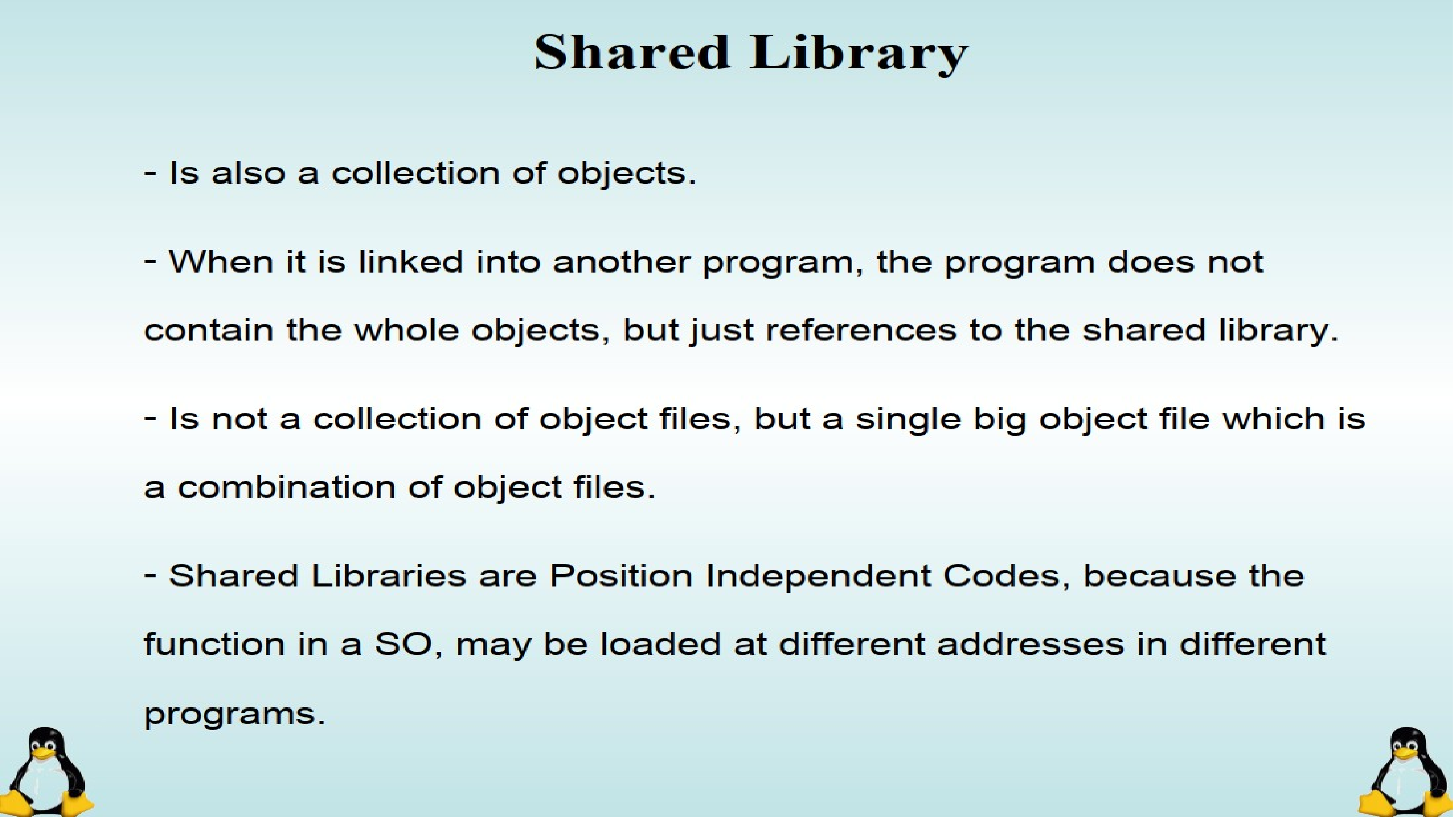
```
gcc <list of sources and object files> -L. –l<library name without lib prefix and .a suffix>
```

**Example:**

```
        gcc -o statictest statictest.c -L. -lfuncs  <correct way of calling>
```

```
       gcc -L. -lfuncs -o statictest statictest.c   <Incorrect way of calling will result in error>
```

# Shared Library



## Shared Library

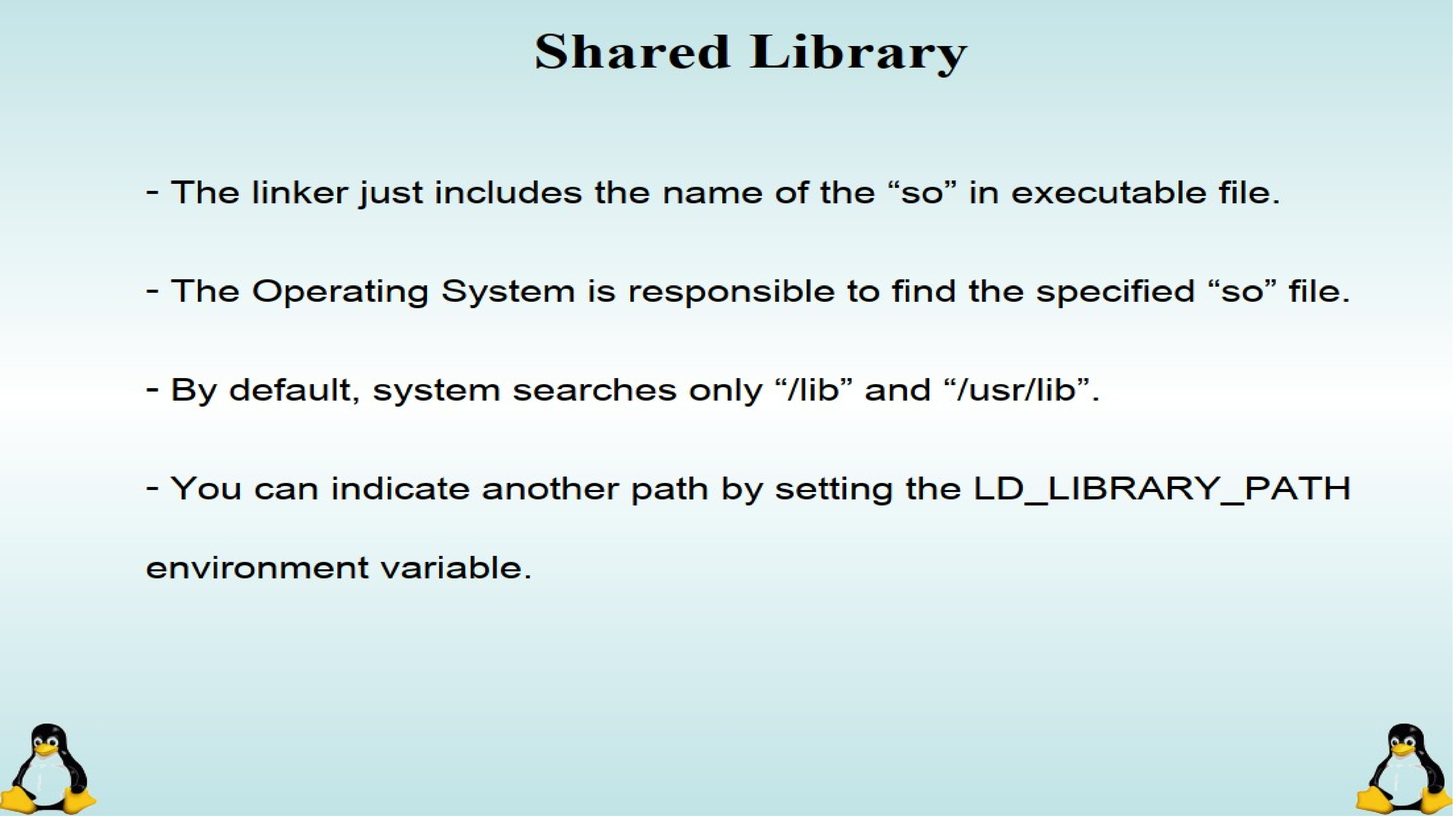
- Is also a collection of objects.

- When it is linked into another program, the program does not contain the whole objects, but just references to the shared library.

- Is not a collection of object files, but a single big object file which is a combination of object files.

- Shared Libraries are Position Independent Codes, because the function in a SO, may be loaded at different addresses in different programs.

# Shared Library



## Shared Library

- The linker just includes the name of the “so” in executable file.

- The Operating System is responsible to find the specified “so” file.

- By default, system searches only “/lib” and “/usr/lib”.

- You can indicate another path by setting the LD_LIBRARY_PATH environment variable.

# Shared Library

```
gcc -shared -fPIC -o <shared library name> <object files list>
```

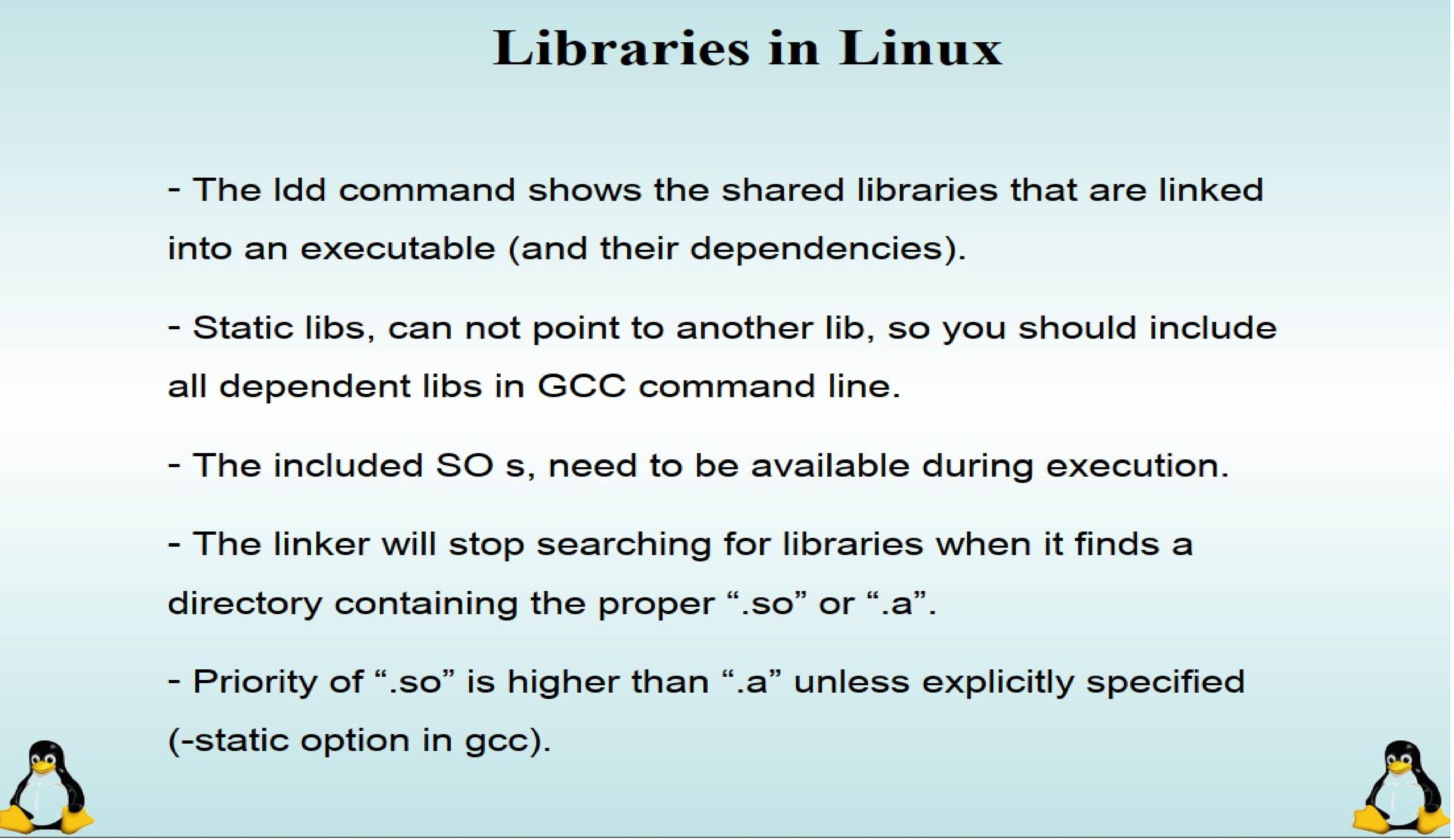
Note: object files should also be compiled with –fPIC option example [ gcc -c -fPIC -o obj1.o obj1.c ]

**Example:**

```
        gcc -shared -fPIC -o libfuncs.so obj1.o obj2.o
```

- LD_LIBRARY_PATH to set the shared library search path
- LD_DEBUG=<options>

# Shared Library



## Libraries in Linux

- The ldd command shows the shared libraries that are linked into an executable (and their dependencies).

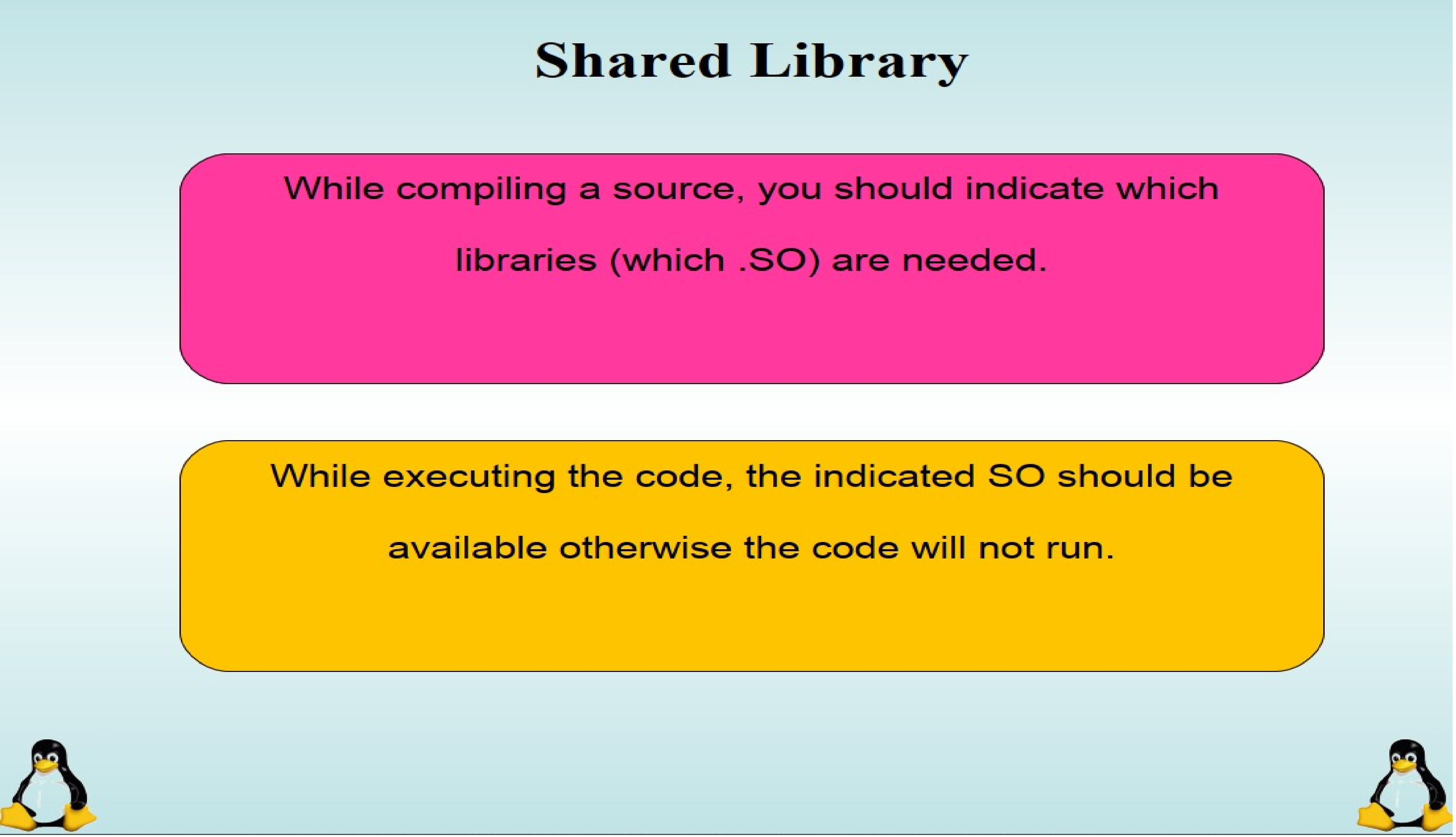
- Static libs, can not point to another lib, so you should include all dependent libs in GCC command line.

- The included SO s, need to be available during execution.

- The linker will stop searching for libraries when it finds a directory containing the proper “.so” or “.a”.

- Priority of “.so” is higher than “.a” unless explicitly specified (-static option in gcc).

# Shared Library

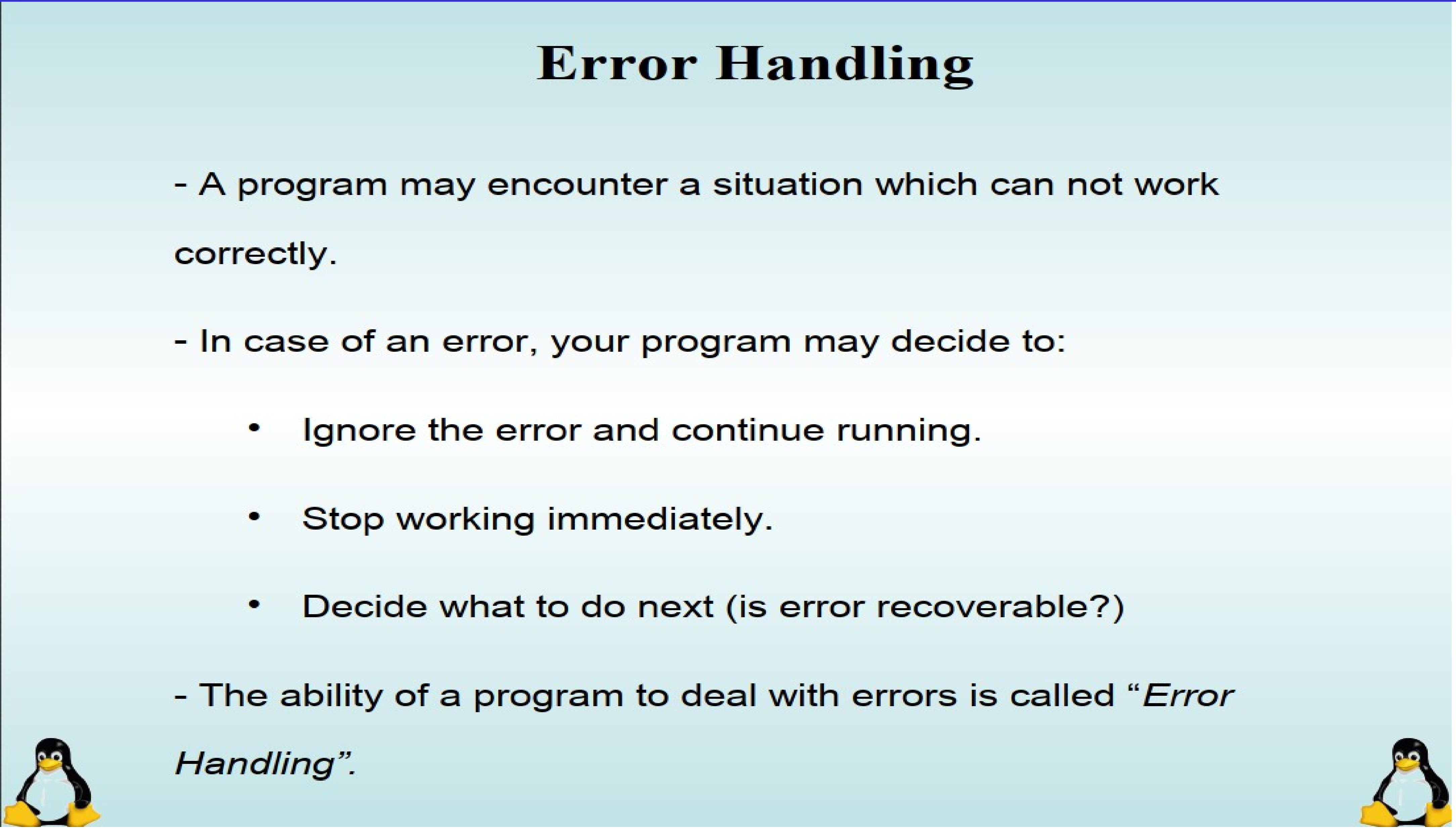


## Shared Library

While compiling a source, you should indicate which libraries (which .SO) are needed.

While executing the code, the indicated SO should be available otherwise the code will not run.

# Error Handling



## Error Handling

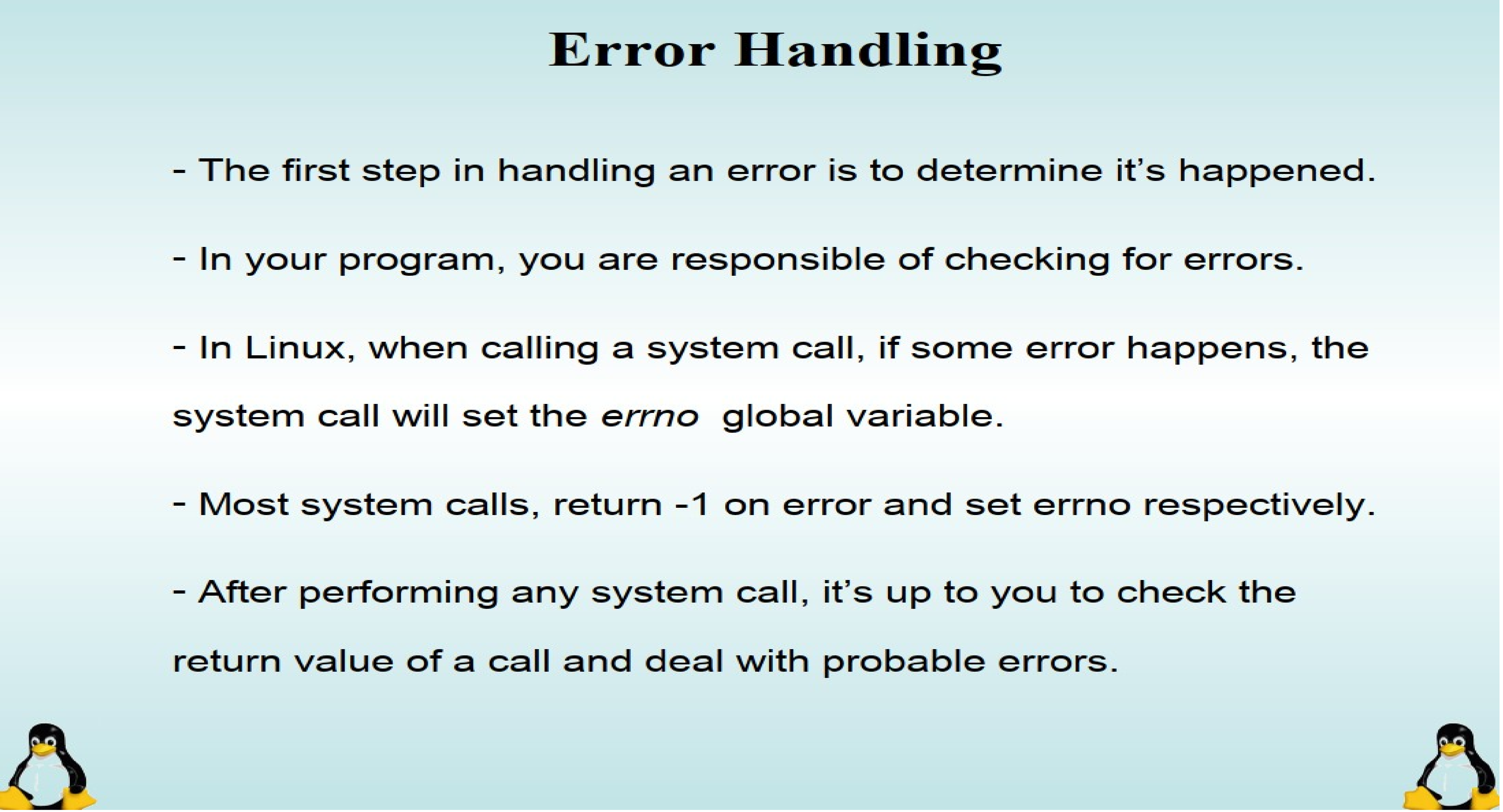
- A program may encounter a situation which can not work correctly.

- In case of an error, your program may decide to:

- Ignore the error and continue running.
- Stop working immediately.
- Decide what to do next (is error recoverable?)

- The ability of a program to deal with errors is called "*Error Handling*".

# Error Handling



## Error Handling

- The first step in handling an error is to determine it’s happened.

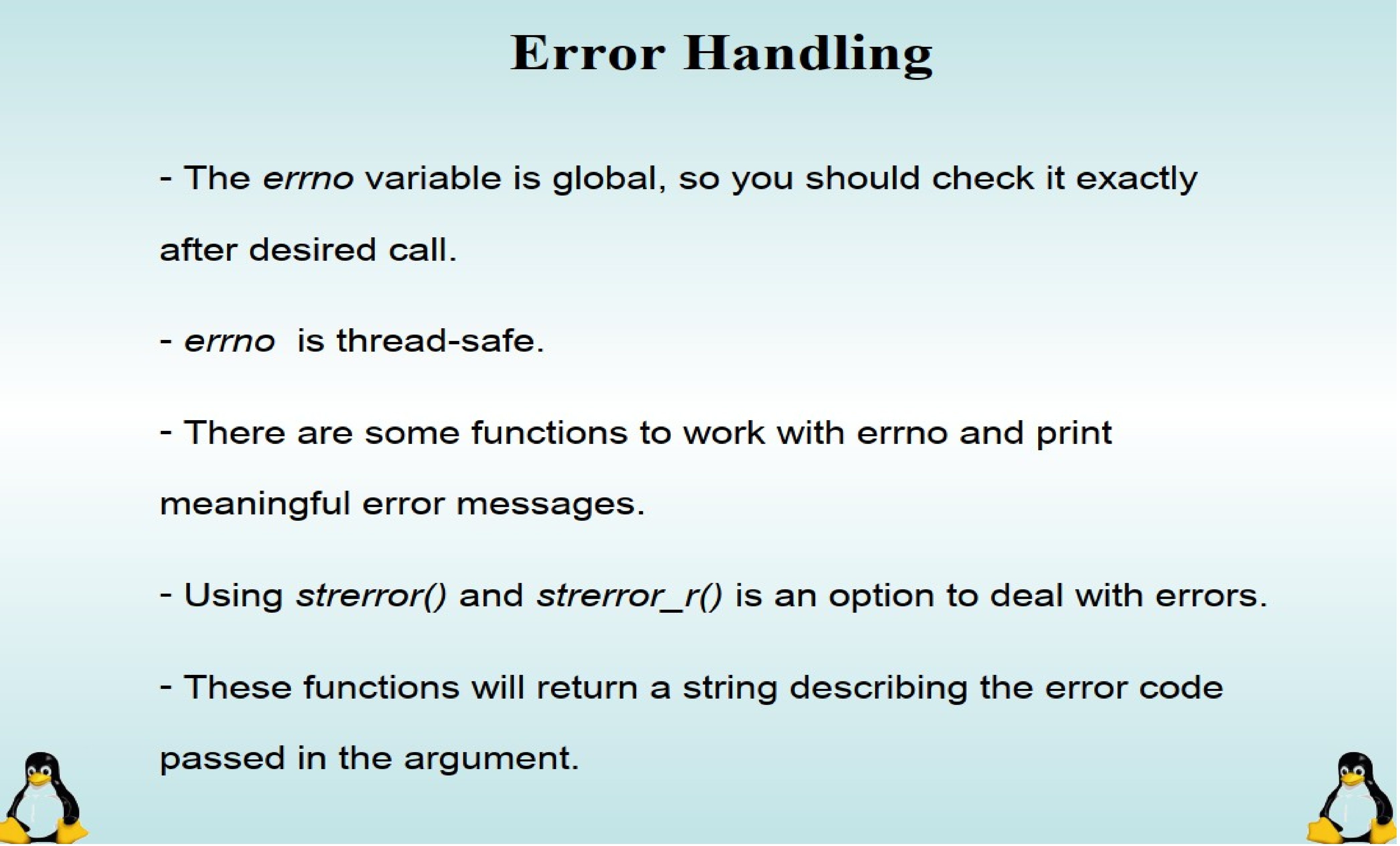
- In your program, you are responsible of checking for errors.

- In Linux, when calling a system call, if some error happens, the system call will set the *errno* global variable.

- Most system calls, return -1 on error and set errno respectively.

- After performing any system call, it’s up to you to check the return value of a call and deal with probable errors.

# Error Handling



## Error Handling

- The *errno* variable is global, so you should check it exactly after desired call.

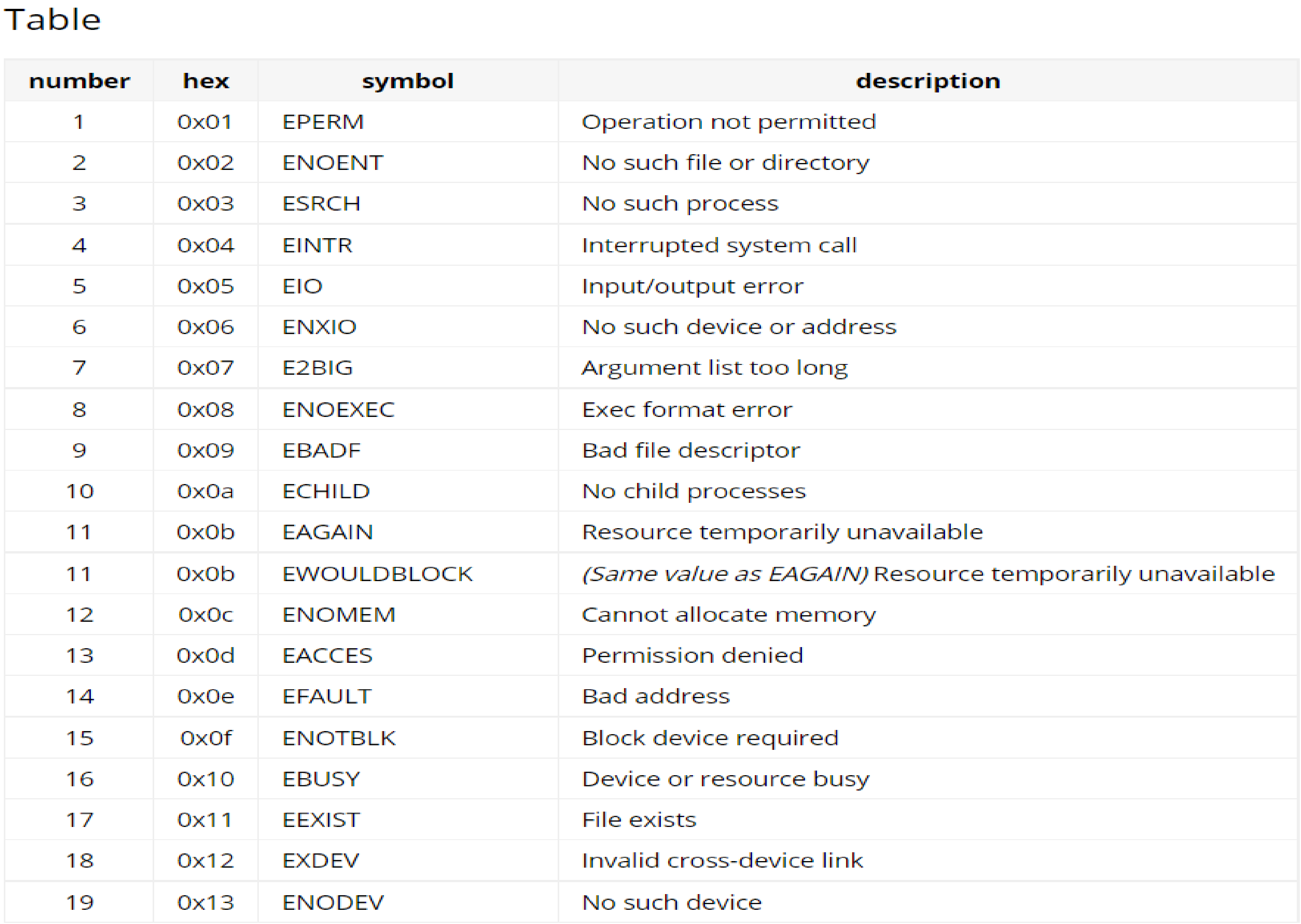
- *errno*  is thread-safe.

- There are some functions to work with errno and print meaningful error messages.

- Using *strerror()* and *strerror_r()* is an option to deal with errors.

- These functions will return a string describing the error code passed in the argument.

# Error Handling



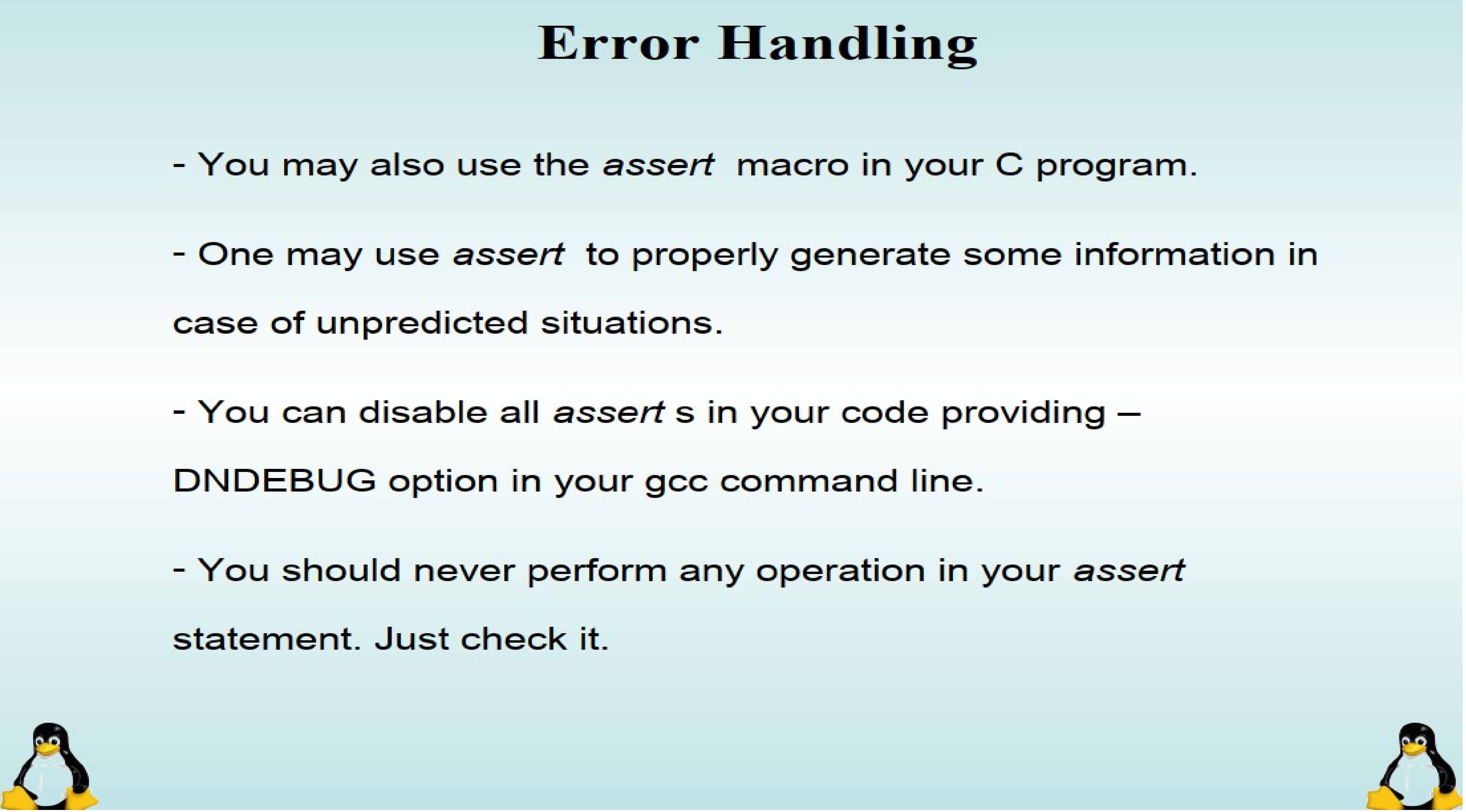
Table

| number | hex | symbol | description |
|---|---|---|---|
| 1 | 0x01 | EPERM | Operation not permitted |
| 2 | 0x02 | ENOENT | No such file or directory |
| 3 | 0x03 | ESRCH | No such process |
| 4 | 0x04 | EINTR | Interrupted system call |
| 5 | 0x05 | EIO | Input/output error |
| 6 | 0x06 | ENXIO | No such device or address |
| 7 | 0x07 | E2BIG | Argument list too long |
| 8 | 0x08 | ENOEXEC | Exec format error |
| 9 | 0x09 | EBADF | Bad file descriptor |
| 10 | 0x0a | ECHILD | No child processes |
| 11 | 0x0b | EAGAIN | Resource temporarily unavailable |
| 11 | 0x0b | EWOULDBLOCK | *(Same value as EAGAIN)* Resource temporarily unavailable |
| 12 | 0x0c | ENOMEM | Cannot allocate memory |
| 13 | 0x0d | EACCES | Permission denied |
| 14 | 0x0e | EFAULT | Bad address |
| 15 | 0x0f | ENOTBLK | Block device required |
| 16 | 0x10 | EBUSY | Device or resource busy |
| 17 | 0x11 | EEXIST | File exists |
| 18 | 0x12 | EXDEV | Invalid cross-device link |
| 19 | 0x13 | ENODEV | No such device |

# Error Handling

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/errnos.md

# Asserts



## Error Handling

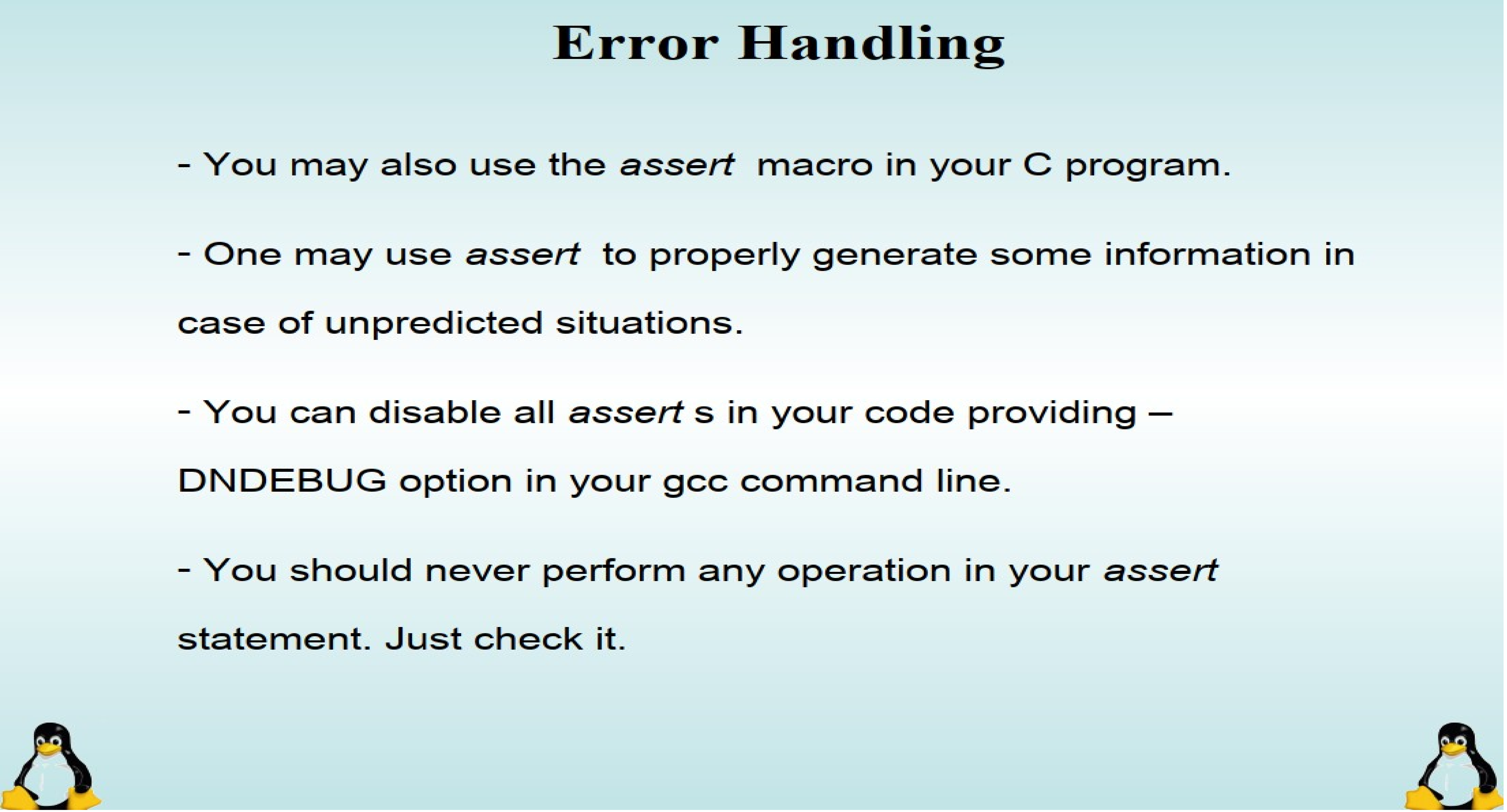
- You may also use the *assert* macro in your C program.

- One may use *assert* to properly generate some information in case of unpredicted situations.

- You can disable all *assert* s in your code providing –DNDEBUG option in your gcc command line.

- You should never perform any operation in your *assert* statement. Just check it.

# Asserts



## Error Handling

- You may also use the *assert* macro in your C program.

- One may use *assert* to properly generate some information in case of unpredicted situations.

- You can disable all *assert* s in your code providing – DNDEBUG option in your gcc command line.

- You should never perform any operation in your *assert* statement. Just check it.

# Asserts

**Syntax**

The syntax for the assert macro in the C Language is:

**void assert(int expression);**

❑**expression** : An expression that we expect to be true under normal circumstances.

# Thank you