ARQCP Course

Arquitetura de Computadores
Licenciatura em Engenharia Informática

2025/26

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

1. Computer
2. Vulnerabilities
3. Memory
4. C Data types
5. C Pointers
6. Integers
7. Bit-Level Operations
8. Logical Operations

1 Computer

2 Vulnerabilities

3 Memory

4 C Data types

5 C Pointers

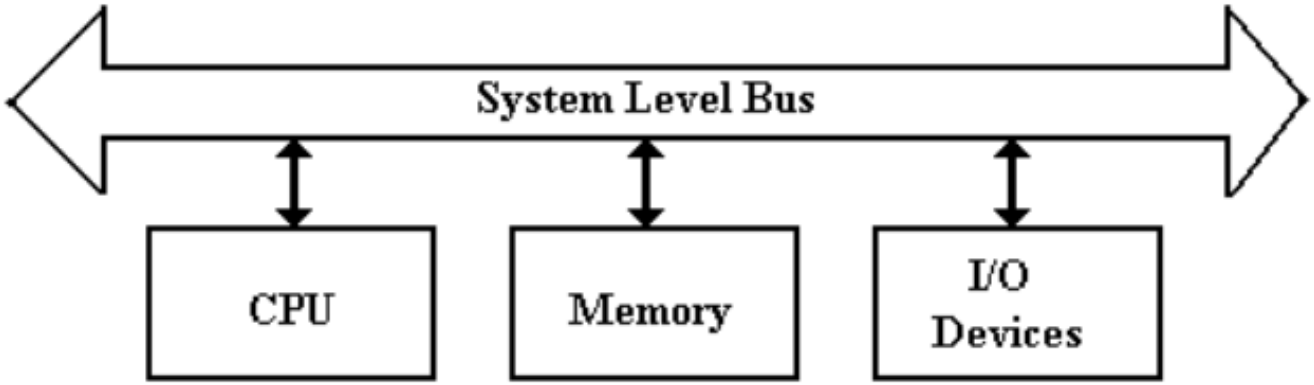
6 Integers

7 Bit-Level Operations

8 Logical Operations

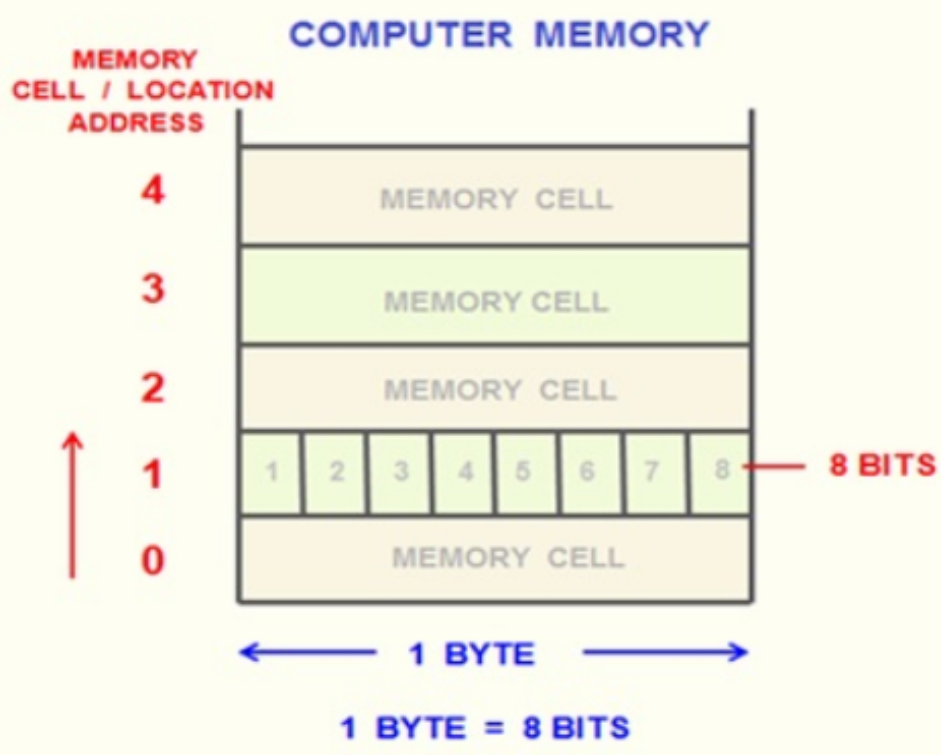
Computer

- The major components of a computer are a **memory unit**, a **central processing unit** (CPU), and **input-output units**.
  - The **memory unit stores programs and data**.
  - The **CPU performs arithmetic and other data-processing operations** as specified by the program.
  - The **program and data are transferred into memory**, typically, triggered by means of an input device such as a keyboard.
  - An output device, such as a display, receives the results of the computations, and the printed results are presented to the user.
- All these components are connected by a **system bus**
  - This is shared by all components.



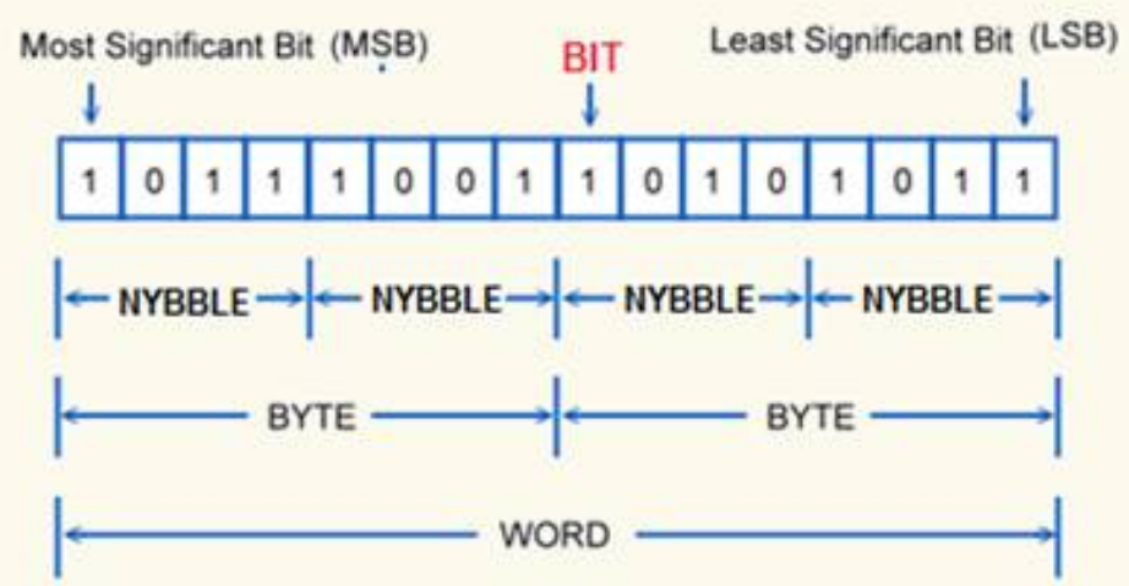
Memory (I)

- It is divided into large number of small parts called **memory cells**.
- Each memory cell is a group of **eight binary cells**.
  - A binary cell is a device that possesses **two stable states and is capable of storing one bit (0 or 1) of information**.
- Each cell (or **byte**) has a **unique address** which starts from zero to memory size minus one.



Memory (II)

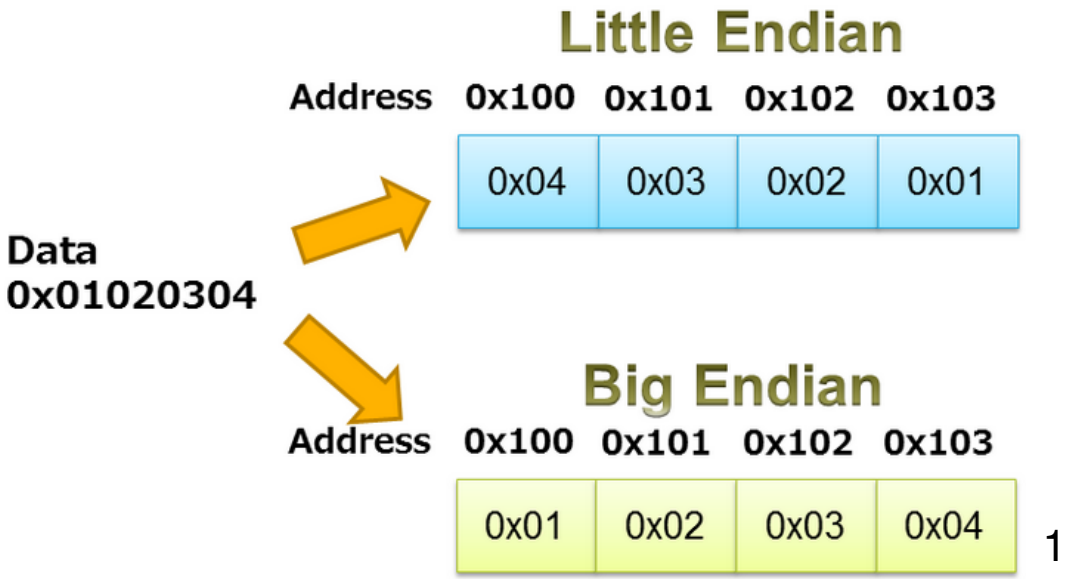
- A **word** is defined as specific number of bits which can be processed together by processor.
  - A word defines the amount of data (number of bits) data can be transferred between CPU and memory in a single operation.



- A **nibble** is a set of four bits.
- **Least Significant Bit** (LSB) is the low-order bit in a set of bits while **Most Significant Bit** (MSB) is the high-order bit.

Memory (III)

- Endianness is a term that describes the **order in which a sequence of bytes is stored in computer memory**.
- **Big-endian** is an order in which the "big end" (most significant value in the sequence) is stored first, at the lowest storage address.
- **Little-endian** is an order in which the "little end" (least significant value in the sequence) is stored first.

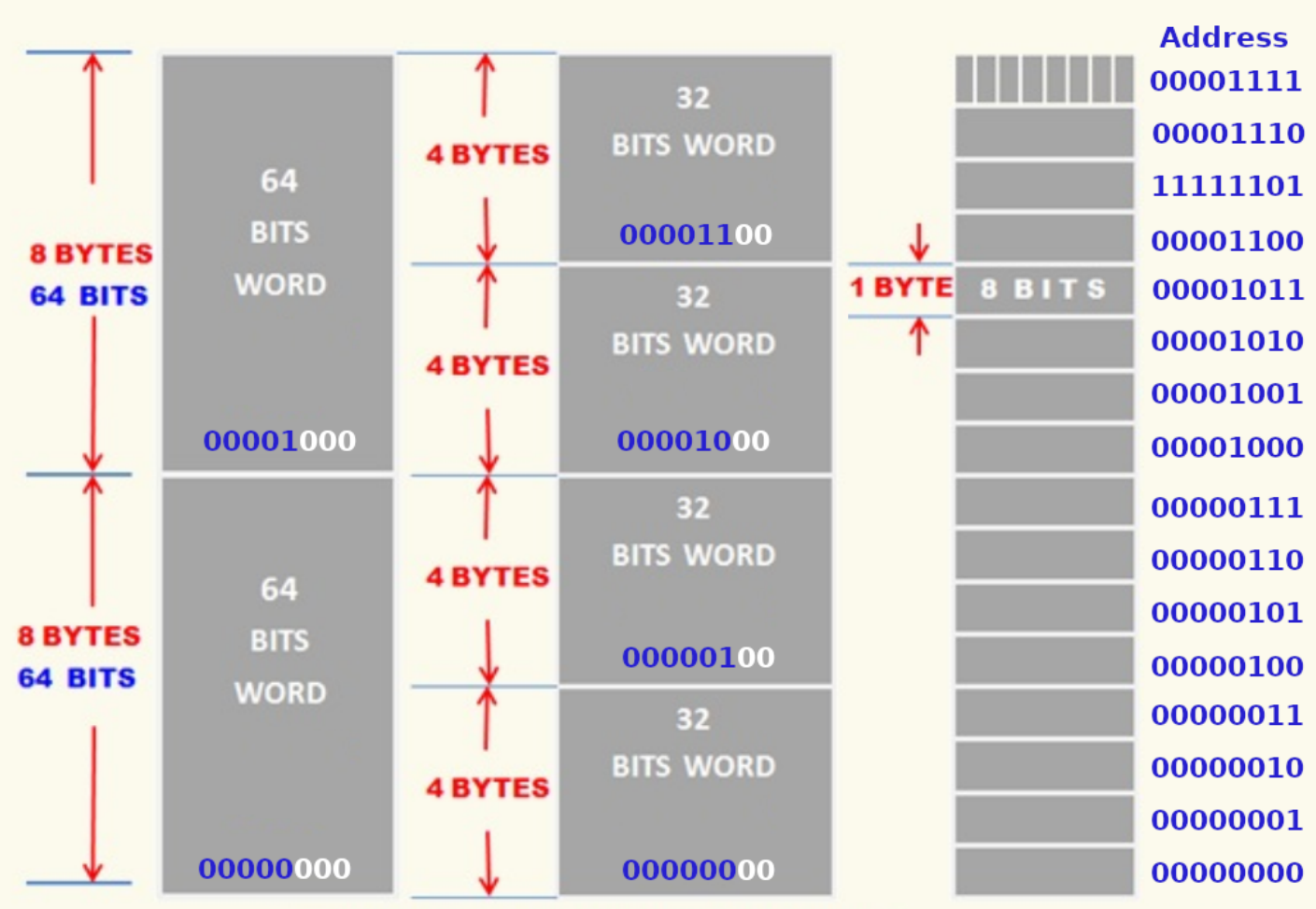


[1]

[1] Hexadecimal numbers are prefixed with `0x`

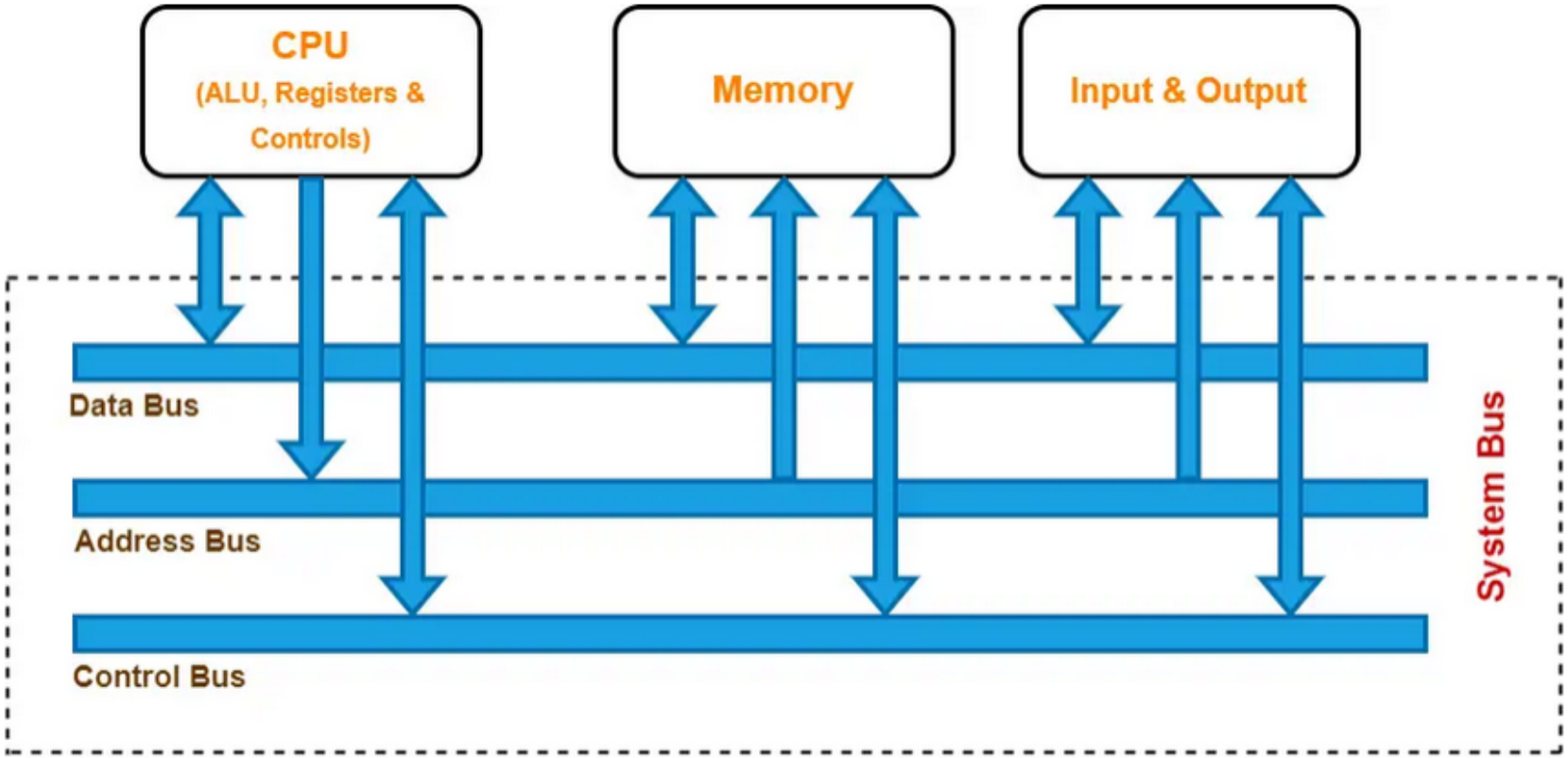
Memory (IV)

- **Word addressable** memory approach.



- The **word size depends on the architecture** (data bus and CPU registers)

System Bus (I)



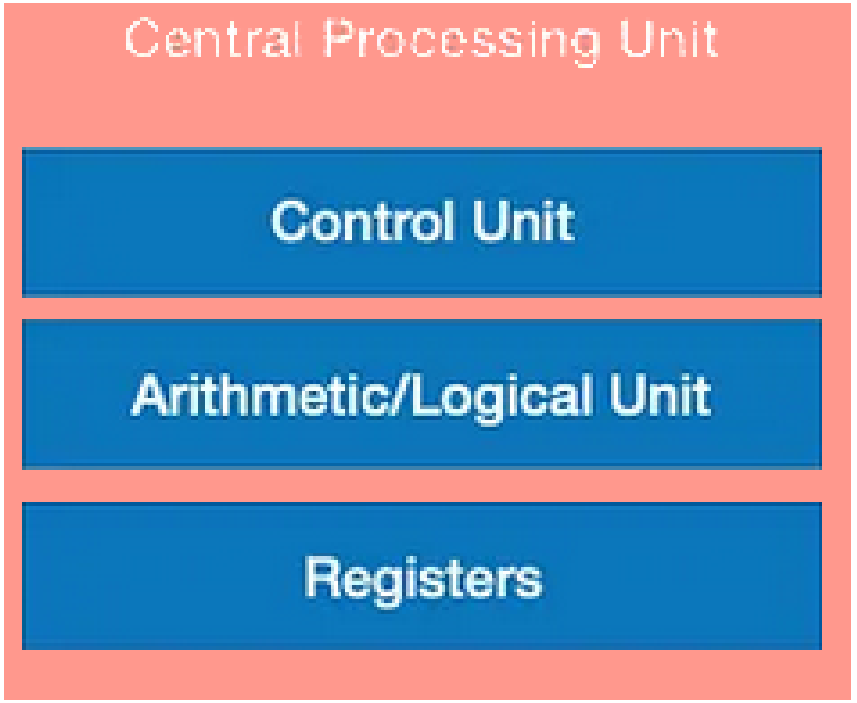
- Data Bus
  - It is used for transmitting the data/instruction
  - It is bi-directional.
  - The width of a data bus refers to the number of bits (electrical wires) that the bus can carry at a time, **word size**.

System Bus (II)

- Address Bus
  - It carries the source or destination address of data i.e. where to store or from where to retrieve the data.
  - It is uni-directional.
- Control Bus
  - It is used to transfer the control and timing signals from one component to the other component.
    - Memory read – Data from memory address location to be placed on data bus.
    - Memory write – Data from data bus to be placed on memory address location.
    - I/O Read – Data from I/O address location to be placed on data bus.
    - I/O Write – Data from data bus to be placed on I/O address location.
    - Other control signals, such interrupts
  - It is bi-directional.

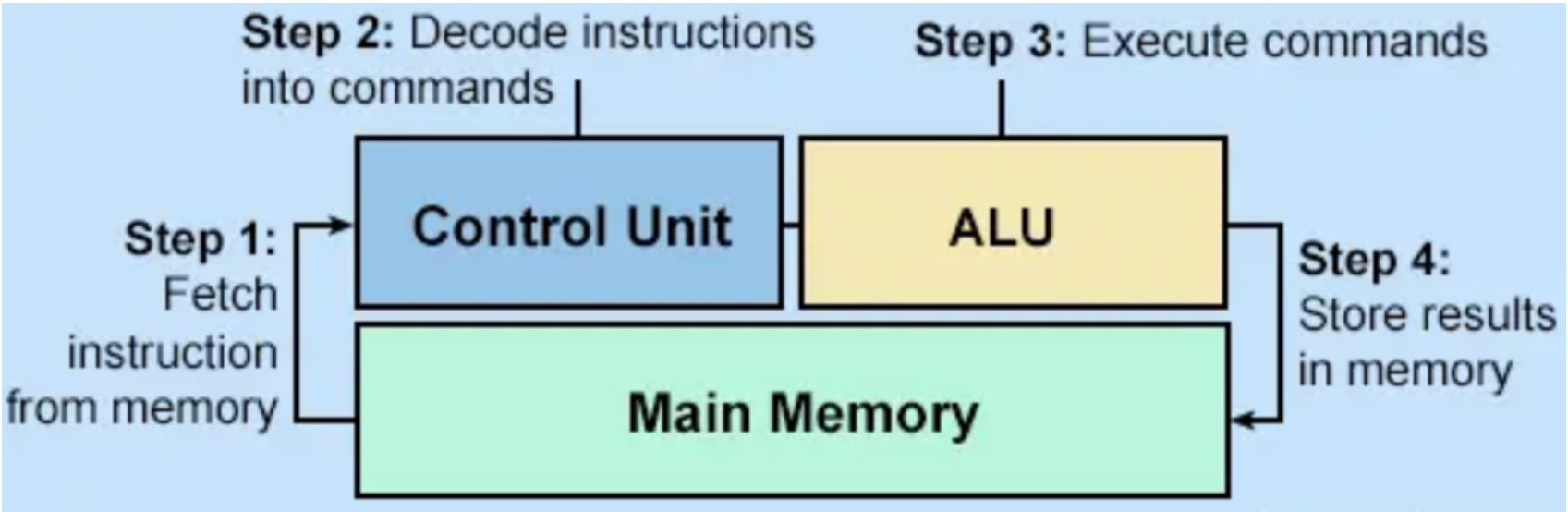
CPU (I)

- Arithmetic Logic Unit (ALU)
  - It performs **logical and mathematical operations**.
- Registers
  - Each CPU contains a set of registers (that are high speed memory chunks)
  - CPU **registers hold data during processing**.



- Control Unit
  - It **directs the operation of other units** by providing **timing and control signals** to synchronize their operations.
  - It directs the flow of data between the CPU and other components, especially the **data flow between memory and CPU**.

- Control Unit (Cont.)
  - It **decodes and translates** the program machine code (bit patterns) instructions during the program execution.



Bytes Units

Full Form	Units	Bytes
1 Byte	8 bits	
1 kilobyte(KB)	1024 byte	2^{10} bytes
1 Megabyte(MB)	1024 KB	2^{20} bytes
1 Gigabyte (GB)	1024 MB	2^{30} bytes
1 Terabyte(TB)	1024 GB	2^{40} bytes
1 Petabyte(PB)	1024 TB	2^{50} bytes
1 Exabyte(EB)	1024 PB	2^{60} bytes

- A kilobyte (KB) is 1,024 bytes, not one thousand bytes as might be expected, because computers use binary (**base two**) math, instead of a decimal (base ten) system.

1 Computer

2 Vulnerabilities

3 Memory

4 C Data types

5 C Pointers

6 Integers

7 Bit-Level Operations

8 Logical Operations

Vulnerabilities

- *A vulnerability is a hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows an attacker to cause harm to the stakeholders of an application. Stakeholders include the application owner, application users, and other entities that rely on the application* [2].
- Software vulnerabilities are when **applications have errors or bugs** in them.
- A **defect in software** or the surrounding processes that could result in the compromise of system assets.
- Attackers look at buggy software as **an opportunity to attack the system making use of these flaws**.

[2] https://owasp.org/www-community/vulnerabilities/

Vulnerabilities Catalogs

- Common Weakness Enumeration (CWE)[3]
  - CWE is a community-developed list of software and hardware weakness types.
- Common Vulnerabilities and Exposures (CVE) [4]
  - Identify, define, and catalog publicly disclosed cybersecurity vulnerabilities.
- Common Vulnerability Scoring System (CVSS) [5]
  - CVSS provides a way to capture the principal characteristics of a vulnerability and produce a numerical score reflecting its severity.
- The Open Web Application Security Project [6]
  - OWASP is a nonprofit foundation that works to improve the security of software.

---

[3] https://cwe.mitre.org/index.html
[4] https://www.cve.org/
[5] https://www.first.org/cvss/
[6] https://owasp.org/

1 Computer

2 Vulnerabilities

3 **Memory**

4 C Data types

5 C Pointers

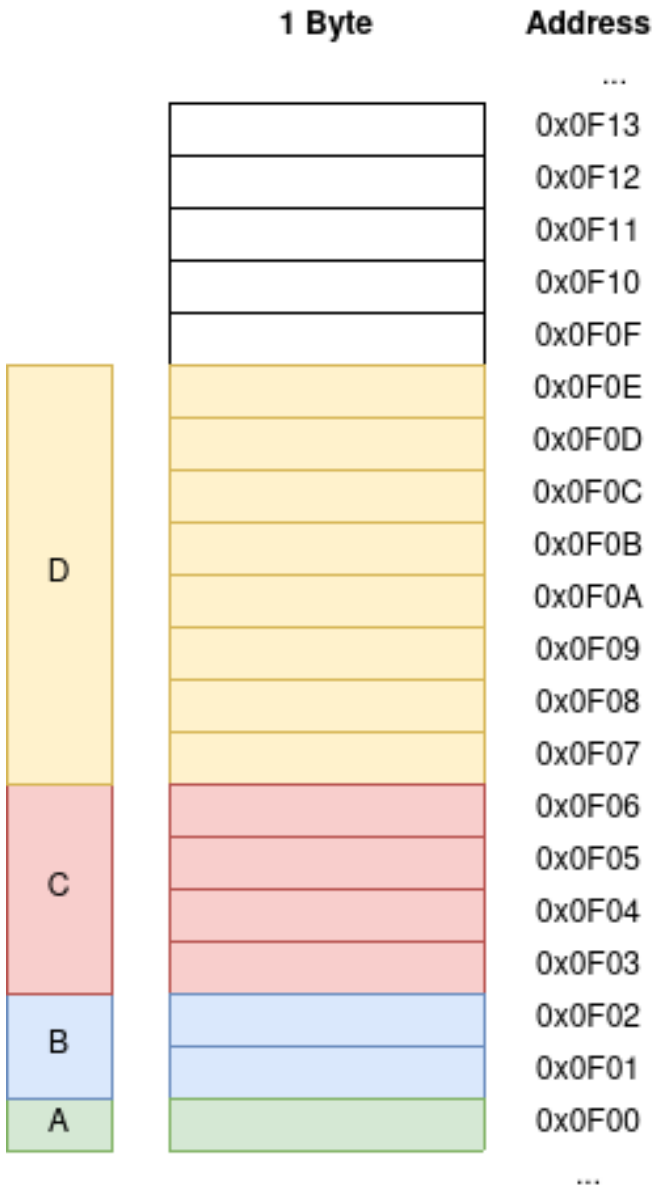
6 Integers

7 Bit-Level Operations

8 Logical Operations

Data Size & Address

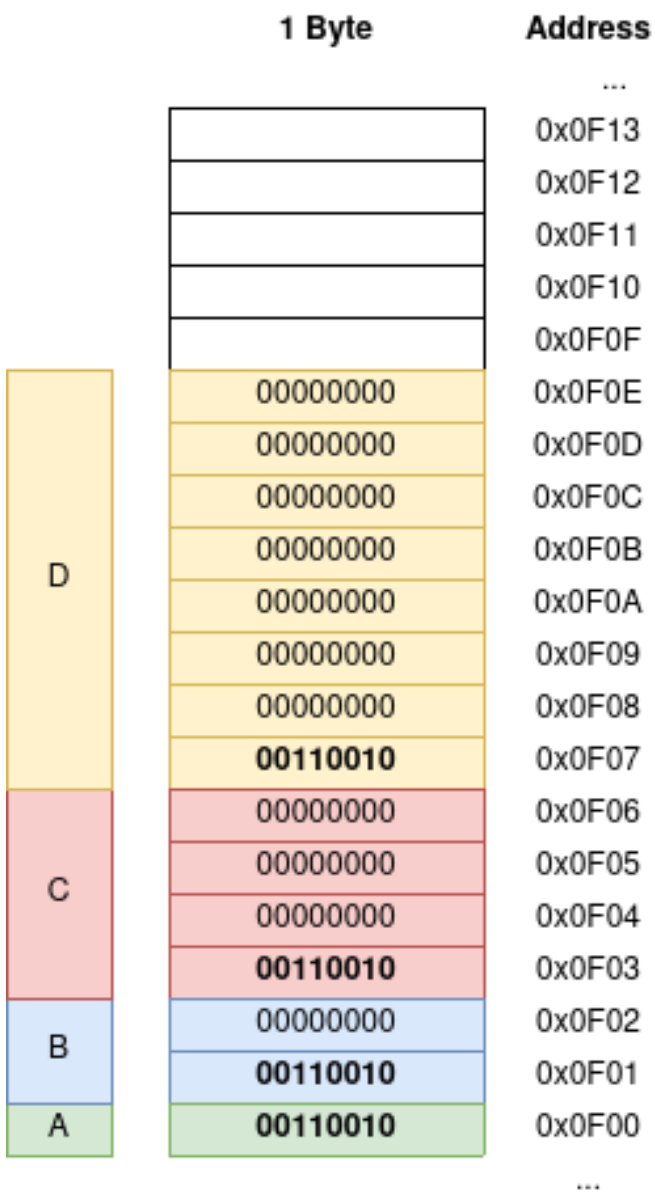
- CPUs have instructions for manipulating **single bytes**, as well as data represented as **2-, 4-, and 8-byte** quantities.
  - However, they provide ways to test and manipulate single bits.
- Data **is encoding using a fixed-size amount of bytes**.
- Memory in a computer is **a set of containers for encoding data**.
  - Each container size **must be in accordance with CPU instructions**.
  - Each container **is addressable** from the address of the first byte (lowest).



Encoding Data (I)

$D = C = B = A = \quad 50_{(10)} \mid 110010_{(2)}$

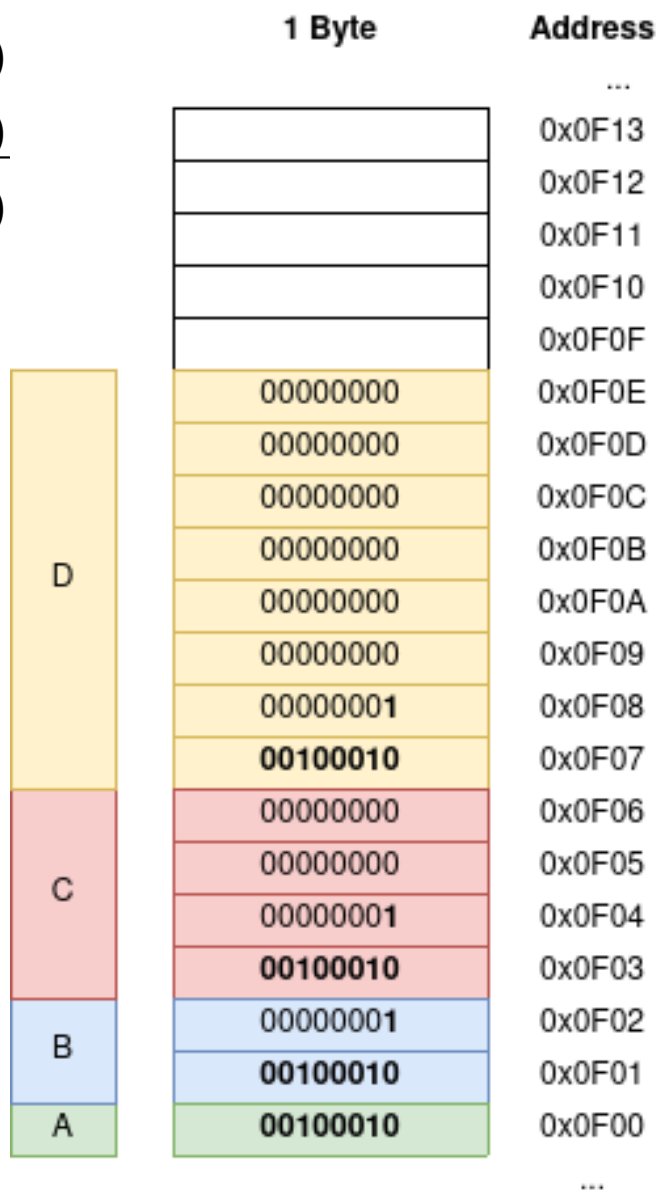
- This is a write operation.
- The goal is to write $50_{(10)}$ value in all containers.
- Given the size of each container, computer performs as follows:
  - To write in A, starting by the address of the first byte, it uses a single byte instruction to write the value.
  - To write in B, starting by the address of the first byte, it uses a 2-bytes instruction to write the value.
  - ...



Encoding Data (II)

$$
\begin{array}{rrr}
 & 50_{(10)} & 110010_{(2)} \\
+ & 240_{(10)} & 11110000_{(2)} \\
\hline
D = C = B = A = & 290_{(10)} & 100100010_{(2)}
\end{array}
$$

- Which is the value of A?
  - $00100010_{(2)} = 34(10)$
- Since computers use a limited number of bits to encode a data
  - Hence, some operations can **overflow/underflow** when the results are too large to be represented
- Binary arithmetic operations explanation can be found at:
  - Moodle web page
  - `https://www.youtube.com/watch?v=-f6fjBhu8eA`



CWE-190: Integer Overflow or Wraparound

- An **integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation**.
  - When this occurs, the value may wrap to become a very small or negative number.
- While this may be intended behavior in circumstances that rely on wrapping, **it can have security consequences if the wrap is unexpected**.
  - This is especially the case **if the integer overflow can be triggered using user-supplied inputs**.
  - This becomes security-critical **when the result is used to control looping**, **make a security decision**, or **determine the offset or size in behaviors such as memory allocation, copying, concatenation**, etc.

2024 CWE Top 25

Rank	ID	Name
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
2	CWE-787	Out-of-bounds Write
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
4	CWE-352	Cross-Site Request Forgery (CSRF)
5	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
6	CWE-125	Out-of-bounds Read
7	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
8	CWE-416	Use After Free
9	CWE-862	Missing Authorization
10	CWE-434	Unrestricted Upload of File with Dangerous Type
11	CWE-94	Improper Control of Generation of Code ('Code Injection')
12	CWE-20	Improper Input Validation
13	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')
14	CWE-287	Improper Authentication
15	CWE-269	Improper Privilege Management
16	CWE-502	Deserialization of Untrusted Data
17	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor
18	CWE-863	Incorrect Authorization
19	CWE-918	Server-Side Request Forgery (SSRF)
20	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
21	CWE-476	NULL Pointer Dereference
22	CWE-798	Use of Hard-coded Credentials
23	CWE-190	Integer Overflow or Wraparound
24	CWE-400	Uncontrolled Resource Consumption
25	CWE-306	Missing Authentication for Critical Function

1 Computer

2 Vulnerabilities

3 Memory

4 C Data types

5 C Pointers

6 Integers

7 Bit-Level Operations

8 Logical Operations

C data types

- A C data type is basically **a group (a container) of contiguous bytes**.

C data type [7]	Number of Bytes			
	RISC-V32	RISC-V64	Intel IA32	x86-64
`char`	1	1	1	1
`short` [`int`]	2	2	2	2
`int`	4	4	4	4
`long` [`int`]	4	8	4	8
`long long` [`int`]	8	8	8	8
`float`	4	4	4	4
`double`	8	8	8	8
`long double`	16	16	10/12	10/16
type$\star$ [8]	4	8	4	8

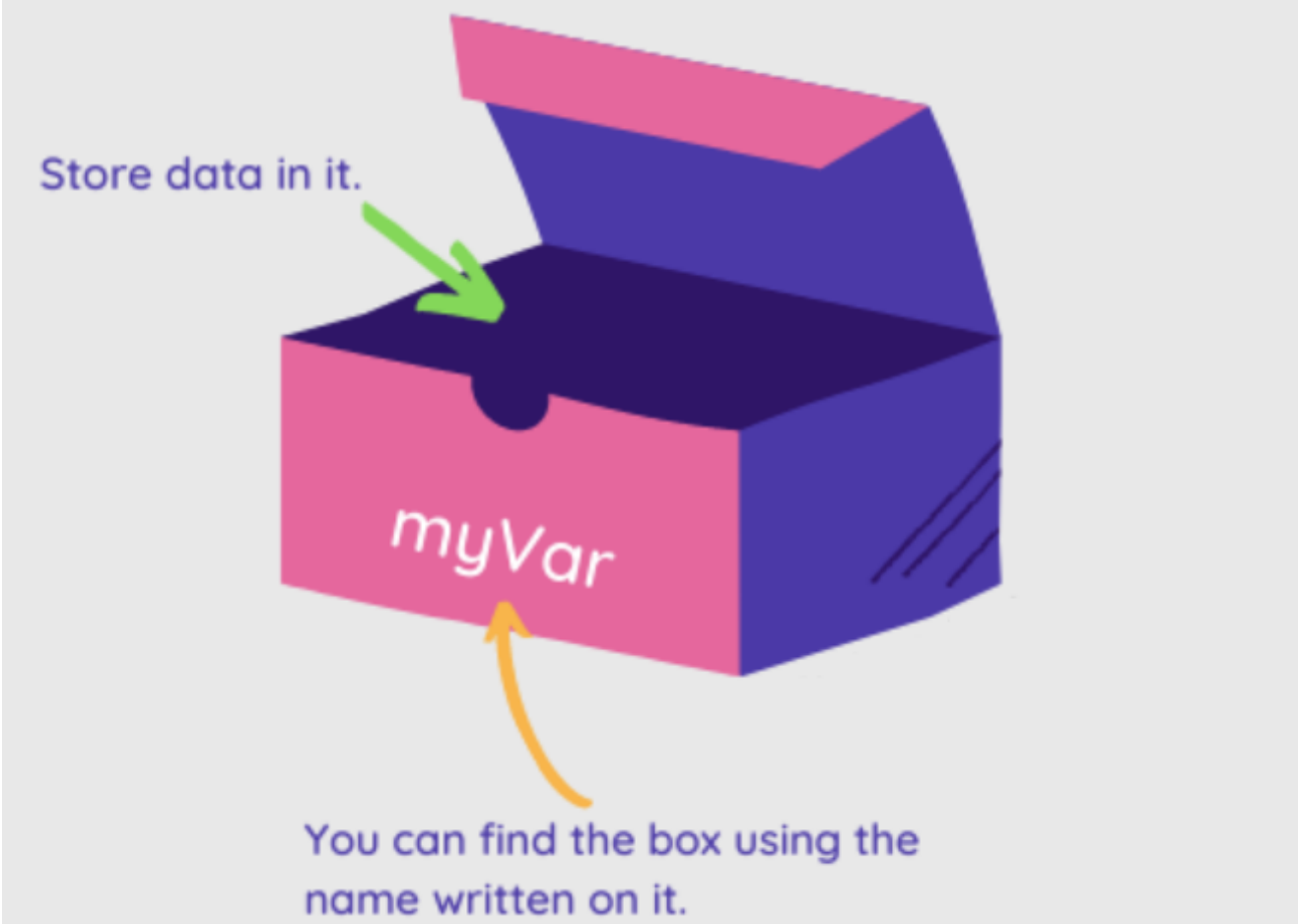
- The integer data types `char`, `short`, `int`, `long`, `long long` can be used to store **non-negative** values when declared as `unsigned` (for instance, `unsigned char`, `unsigned int`, ...).

[7] Text in square brackets is optional.

[8] Pointer

Variables

- Variables are used **to manage data**.
- A variable is a **symbolic name for (or reference to) data**.



- The data stored in a variable **can be changed during the course of the program**, which is why it is named a **variable**.

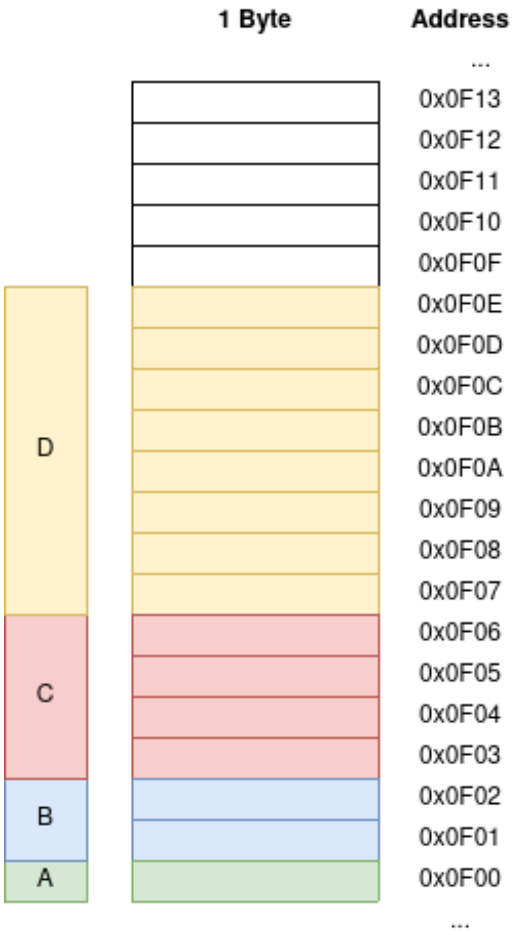
Declaring a Variable

- Variables are **containers for storing data values**.
- Each container **is addressable from the address of the first byte (lowest)**.
- Which is the output of following code?

```
printf("%p\n", &A);
printf("%p\n", &B);
printf("%p\n", &C);
printf("%p\n", &D);
```

  - We can get the **memory address of a variable with the reference operator `&`**.

```
char A;
short int B;
int C
long long D;
```



1 Computer

2 Vulnerabilities

3 Memory

4 C Data types

5 C Pointers

6 Integers

7 Bit-Level Operations

8 Logical Operations

Declaration

- A C pointer is also a **C data type as the other types**
  - So it is also **a group of contiguous bytes**.
- Declaring a C pointer variable

```
type* var-name;
```

  - Where `type` is any valid C data type, such as `int`, `char`, or other.
    - The `type` **should be of the variable the pointer will be pointing**.
  - The asterisk `*` is used to declare a pointer
  - The `var-name` is the name of the pointer variable.

```
int* ip; /* pointer to an integer */
double* dp; /* pointer to a double */
float* fp; /* pointer to a float */
char* cp /* pointer to a character */
```

- The size (number of bytes) of a pointer variable is **the same regardless of its type**.

```
printf("%d\n", sizeof(ip));
printf("%d\n", sizeof(dp));
printf("%d\n", sizeof(fp));
printf("%d\n", sizeof(cp));
```

Uninitialized and Null

- A pointer declared **without initialization**, i.e., it is pointing to "somewhere" which **could be an invalid memory location**.

```
int* ip; /* pointer to an integer */
double* dp; /* pointer to a double */
float* fp; /* pointer to a float */
char* cp /* pointer to a character */
```

- **It is always a good practice to assign a `NULL` value to a pointer variable in case you do not have an exact address to be assigned.**
  - This is done at the time of variable declaration.
  - A pointer that is assigned `NULL` is called a **null pointer**.

```
int* ip = NULL;
double* dp = NULL;
float* fp = NULL;
char* cp = NULL;
```

- The `NULL` pointer is a constant with a value of zero defined in several standard libraries.

Referencing

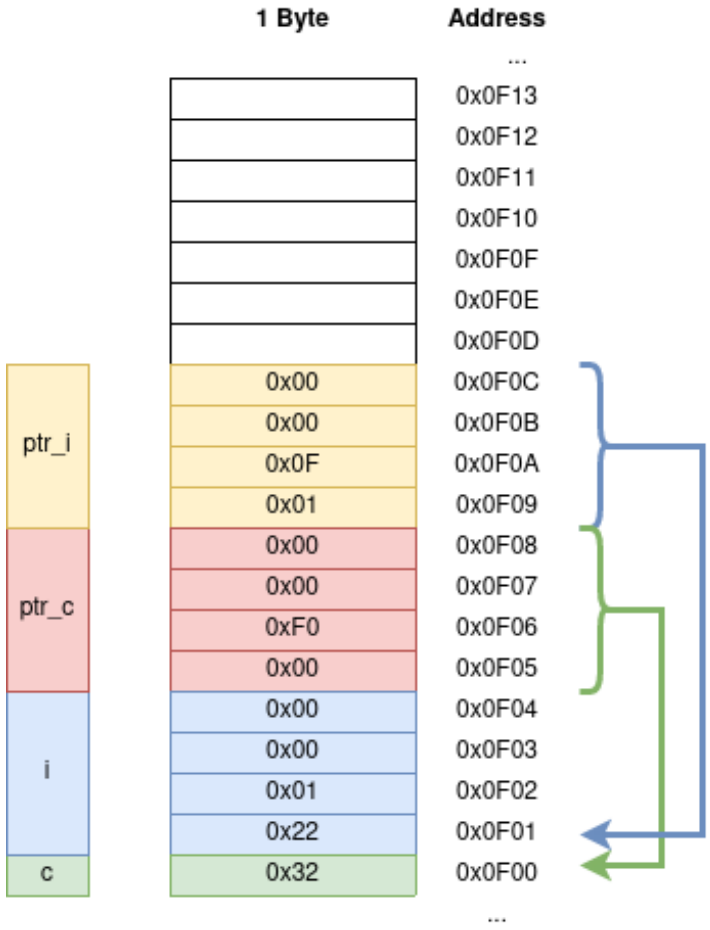
- Making a pointer variable to point other variables by providing address of that variable to the pointer is known as **referencing of pointer** (or initialization of pointer).

```
pointer_variable = &normal_variable;
```

- A pointer is a variable **that stores the memory address of another variable as its value**.
  - The value of a pointer variable is an address
- Which is the output of following code?

```
printf("%p, %p, %p, %p\n", &c, &i, &ptr_c, &ptr_i);
printf("%x, %x, %p, %p\n", c, i, ptr_c, ptr_i);
```

```
char c = 50; //0x32
int i = 290; //0x122
char* ptr_c = &c;
int* ptr_i = &i;
```



Dereferencing

- The operator $\star$ used in front of the name of the pointer variable is known as pointer or **dereferencing or indirection operator**.

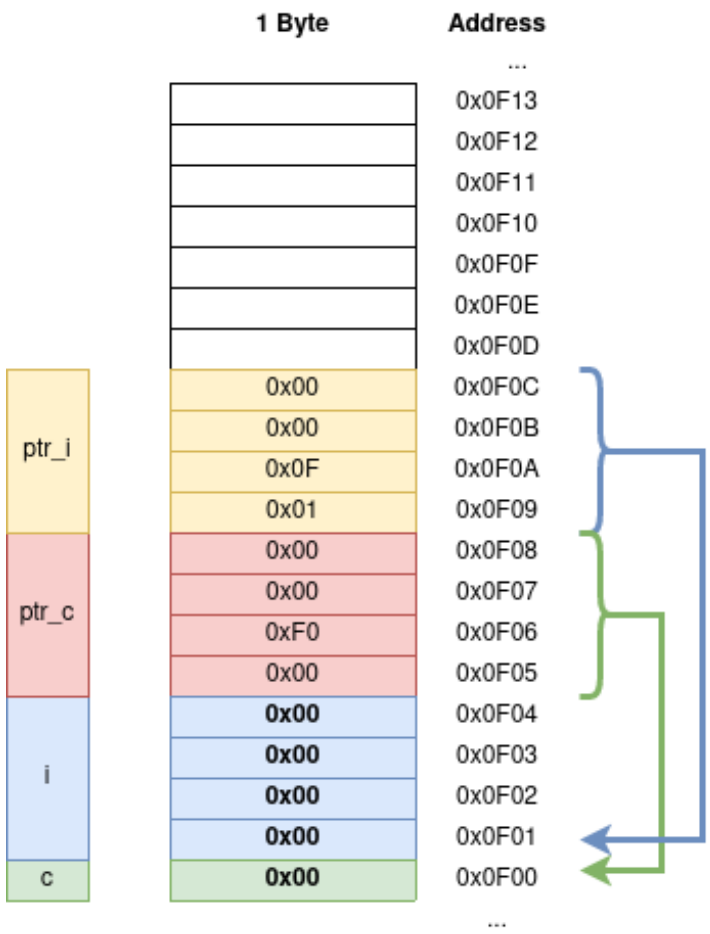
```
*pointer_variable
```

  - It allows **to manage the content of such pointed variable**

- Which is the output of following code?

```
printf("%x, %x, %x, %x\n", c, i, *ptr_c, *ptr_i);
```

```
*ptr_c = 0x00;
*ptr_i = 0x00;
```



Arithmetic

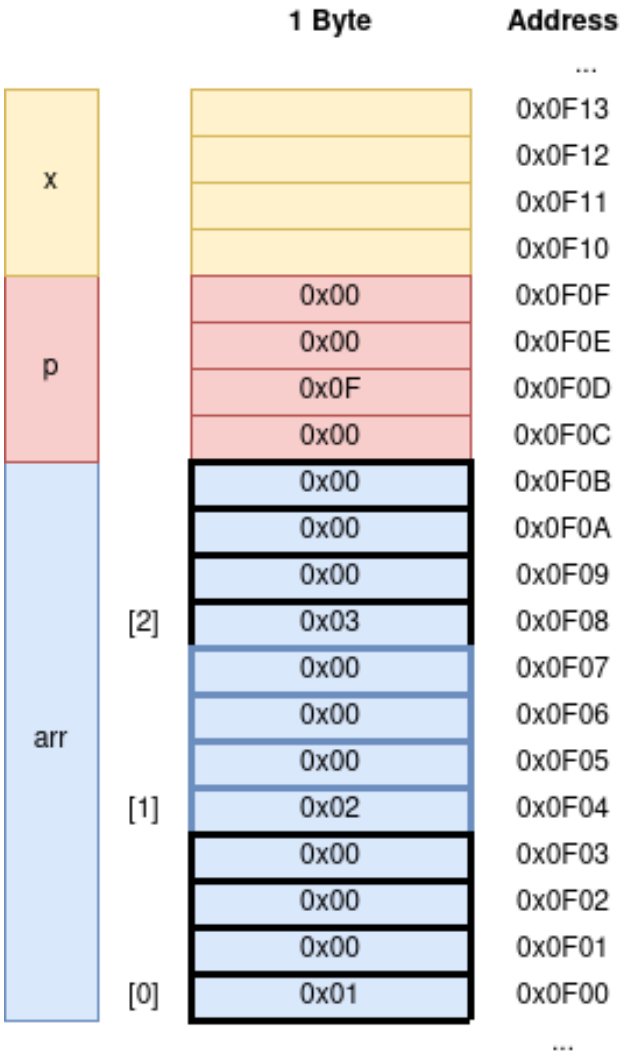
- A pointer in C is an address, which is a **numeric value (unsigned)**.
  - So you can perform **arithmetic operations on a pointer** just as you can on a numeric value.
    - There are four arithmetic operators that can be used on pointers: `++`, `--`, `+`, and `-`
- What is the `x` value?

```
int* p = arr;
p++;
int x = *(p + 1);
```

- What is the `x` value?

```
int* p = arr + 2;
p--;
int x = *(p - 1);
```

```
int arr[3] = {1,2,3};
```



Comparison

- Pointers may be compared by using relational operators, such as >, >=, <, <=, ==, !=.

```
int arr[5] = {1,2,3,4,5};
int* ptr1 = arr;
int* ptr2 = &arr[0];
if (ptr1 == ptr2) {
  printf("The value of the rrray name and the address of first element are equal.");
}
int* ptr = arr;
while(ptr < arr + 5){
  printf("%d\n", *ptr);
  ptr++;
}
```

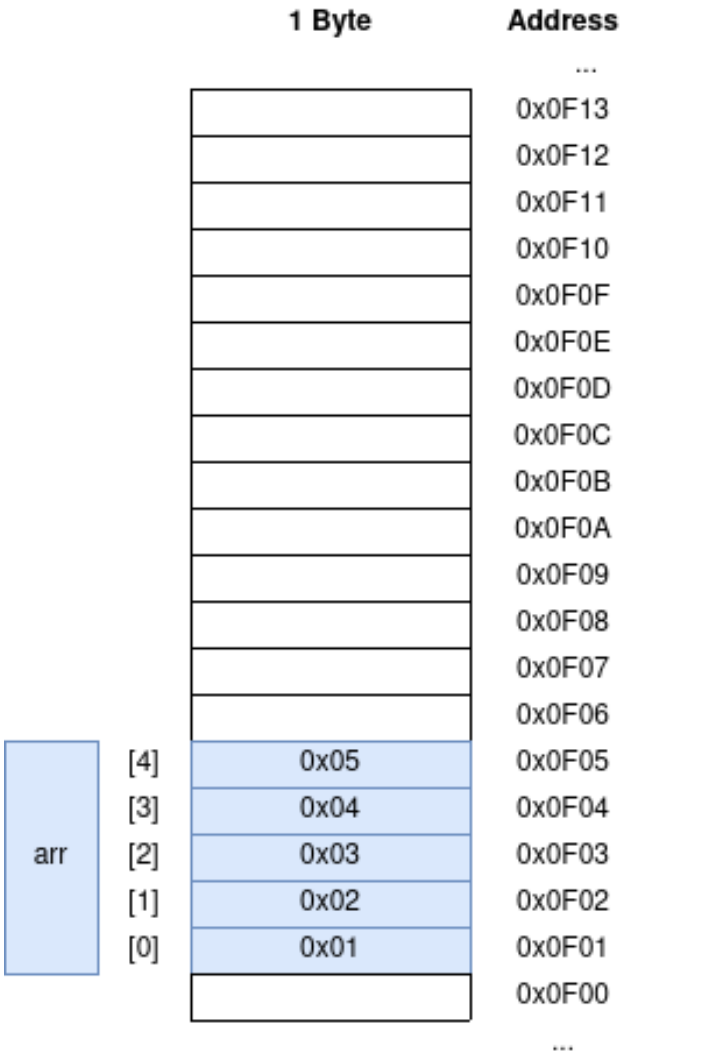
C Arrays (1-Dimensional)

- The **name of an array** (base address), is actually **a pointer to the first element of the array**.
- The memory address of the first element is the same as the name of the array .
- Which is the output of following code?

```
printf("%p, %p, %p\n", arr, &arr[0], &arr[4]);
```

- **To find the address of an element in an array** the following formula is used:
  - Address of array[i] = base address of array + (sizeof(data type) * i)

```
char arr[5];
```



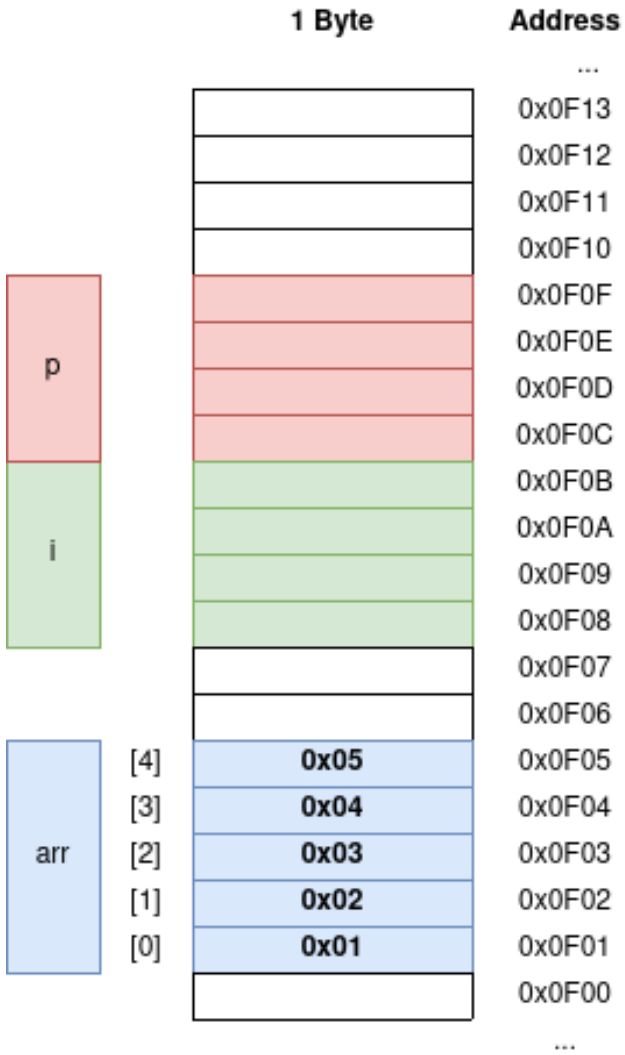
Pointers and Arrays

- Traversal is the process in which we visit every element of the data structure.
- For C array traversal,
  - Using **an index** variable.

```
int i;
for (i = 0; i < 5; i++) {
  arr[i] = i + 1;
}
```

  - Using **pointers**.

```
char* p;
for (p = arr; p < arr + 5; p++) {
  *p = (p - arr) + 1;
}
```



Why does it need a type?

- **Memory operations**
  - A pointer holds the address of the first byte (of a set of bytes).
  - So, from that address, **it needs to how many bytes will be involved in an operation** (read, write and so on).
    - If it is a `char *` pointer, any read or write operation will involved one single byte.
    - If it is a `int *` pointer, any read or write operation will involved four bytes.
- **Arithmetic operations**

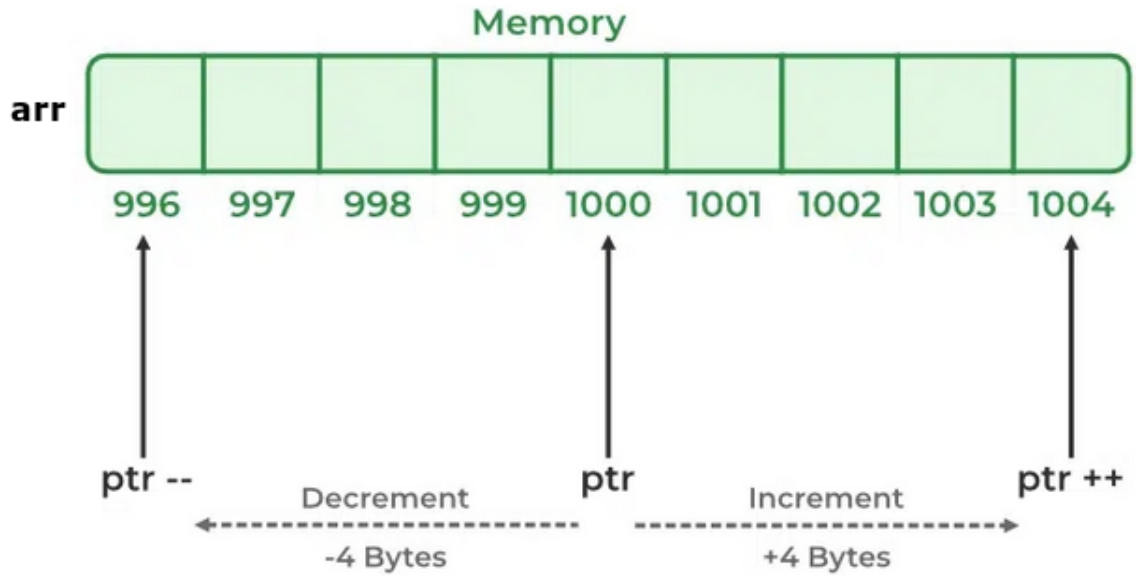
```
int arr[5];
int* ptr = arr;
ptr++;
printf(" ptr = %p, arr= %p\n", ptr,arr);
printf(" ptr - arr = %d\n", ptr - arr);
```

```
ptr = 0x3ffff19c, arr= 0x3ffff198
ptr - arr = 1
```

  - Is this (0x3ffff19c-0x3ffff198 = 1) correct?

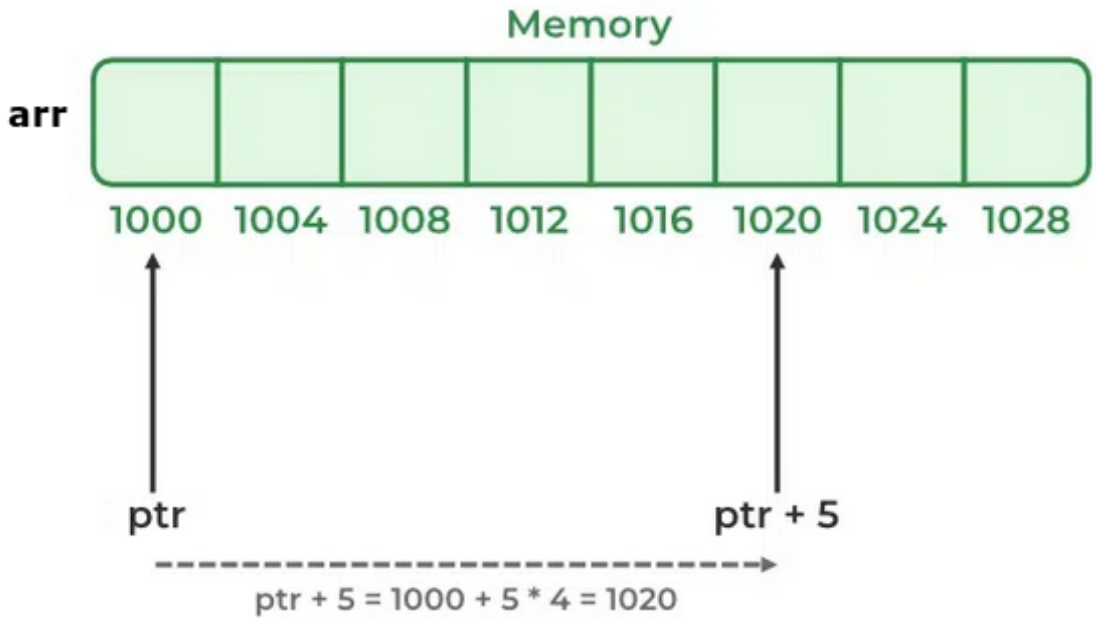
Increment/Decrement

- **Increment**
  - When a pointer is incremented, it actually **increments by the number equal to the size of the data type for which it is a pointer**.
- **Decrement**
  - When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.



Addition/Subtraction of Integer

- **Addition**
  - When a pointer is added with an integer value, **the value is first multiplied by the size of the data type and then added to the pointer**.



- **Subtraction**
  - When a pointer is subtracted with an integer value, the value is first multiplied by the size of the data type and then subtracted from the pointer similar to addition.

1 Computer

2 Vulnerabilities

3 Memory

4 C Data types

5 C Pointers

6 Integers

7 Bit-Level Operations

8 Logical Operations

Integer

- Integers represent a growing and underestimated source of vulnerabilities in C and C++ programs.
- Integer **range checking has not been systematically applied** in the development of most C and C++ software.
- Integers in C and C++ are either signed (default) or unsigned (when declared as `unsigned`).
- **Signed integers are used to represent positive and negative values**, the range of which depends on the number of bits allocated to the type and the encoding technique.
  - On a computer using two's complement arithmetic, a signed integer ranges from -2^{n-1} through $2^{n-1}-1$ (where n is the number of bits).
    - The most significant bit indicates sign,0 for non-negative and 1 for negative
- **Unsigned integer values range from zero to a maximum** that depends on the size of the type.
  - This maximum value can be calculated as $2^n - 1$ (where n is the number of bits)

Encoding

- Values for `short` (16 bits)

	Decimal	Hexadecimal	Binary
Min	(-2^{15}) -32768	0x80 00	10000000 00000000
-1	-1	0xFF FF	11111111 11111111
0	0	0x00 00	00000000 00000000
Max	$(2^{15} - 1)$ 32767	0x7F FF	01111111 11111111

- Values for `unsigned short` (16 bits)

	Decimal	Hexadecimal	Binary
Min	0	0x00 00	00000000 00000000
Max	$(2^{16} - 1)$ 65535	0xFF FF	11111111 11111111

- The limits for integer types are defined in the C standard header file `<limits.h>`.
  - `INT_MIN` = -2147483648, defines the minimum value for an `int`.
  - `INT_MAX` = +2147483647, defines the maximum value for an `int`.
  - `UINT_MAX` = 4294967295, defines the maximum value for an `unsigned int`.

Type casting

- Typecasting **is a way of converting a variable or data from one data type to another data type**.
- There are 2 types of casting
  - **Implicit casting** (Compiler does this)
    - The implicit type conversion takes place when more than one data type is present in an expression.
    - It is done by the compiler itself it is also called **automatic type conversion**.
    - If the operands are of two different data types, then **an operand having lower data type is automatically converted into a higher data type** (is also called as **type promotion**).
  - **Explicit casting** (Programmer does this)
    - Explicit type conversion refers to the type conversion performed by a programmer by modifying the data type of an expression using the type cast operator.

```
(datatype) expression
```

- Data will **be truncated when the higher data type is converted into a lower**.

Explicit type casting

```
...
int sum = 17, count = 5;
double mean;
mean = sum / count;
printf("Value of mean: %f\n", mean);
```

- Both the operands in the division expression are of `int` type
- Output: Value of mean: 3.000000

```
...
int sum = 17, count = 5;
double mean;
mean = (double) sum / count;
printf("Value of mean: %f\n", mean);
```

- Cast operator has precedence over division, so the value of `sum` is first converted to type `double` and finally it gets divided by `count` yielding a `double` value.
- Output: Value of mean: 3.400000

Implicit type casting (promotion)

```
int i = 17;
char c = 'c'; /* ascii value is 99 */
int sum;
sum = i + c;
printf("Value of sum : %d\n", sum);
```

- The compiler is doing integer promotion of `char c`.
- Output: Value of sum: 116

```
int i = 17;
char c = 'c'; /* ascii value is 99 */
float sum;
sum = i + c;
printf("Value of sum : %f\n", sum);
```

- Output: Value of sum: 116.000000

Conversions Between Signed and Unsigned (I)

- C allows casting between different numeric data types.
  - For example, suppose variable `x` is declared as `int` and `u` as `unsigned int`.

```
int x = ...;
unsigned int u = ...;
u = (unsigned) x;
x = (int) u;
```

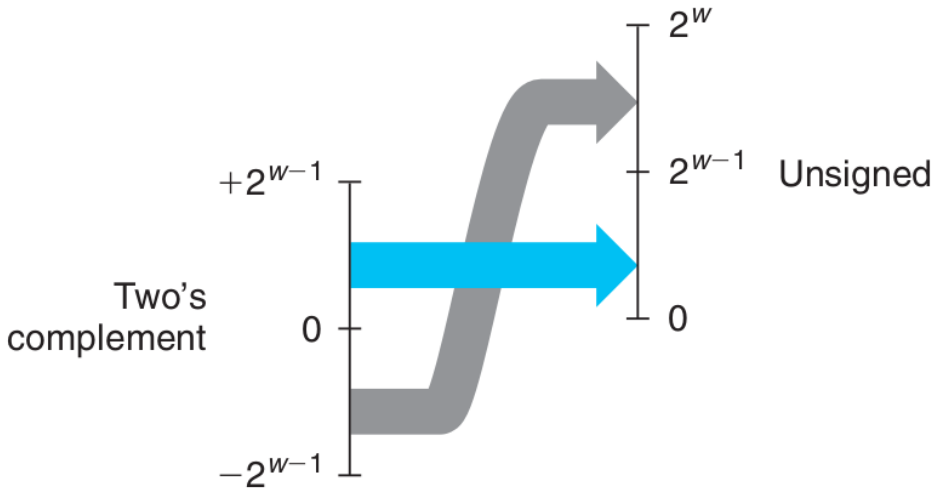
  - The expression `u = (unsigned) x` converts the value of `x` to an unsigned value, and `x = (int) u` converts the value of `u` to a signed integer.
- The general rule for conversions between signed and unsigned numbers with the same word size – **the numeric values might change, but the bit patterns do not**.

```
short v = -12345;
unsigned short uv = (unsigned short) v;
printf("v = %d, uv = %u\n", v, uv);
```

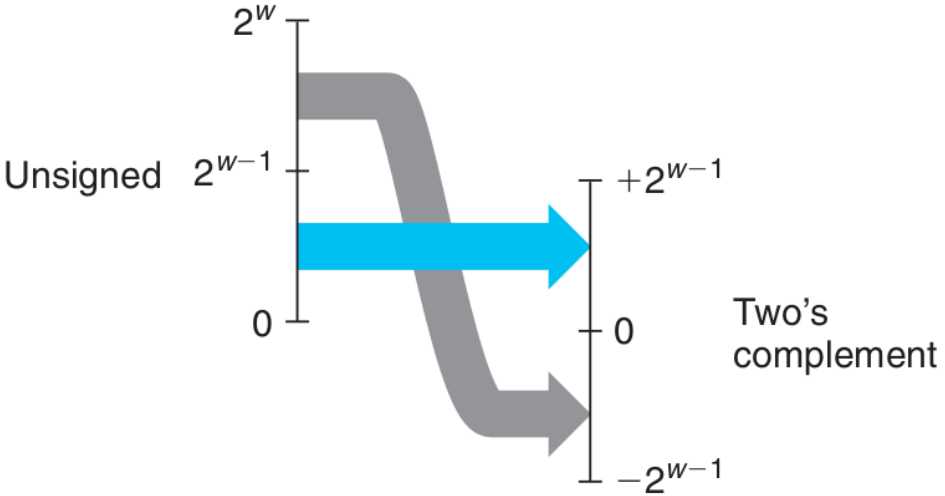
```
v = -12345, uv = 53191
```

Conversions Between Signed and Unsigned (II)

- Conversion from two's complement to unsigned (where w is the number of bits).



- Conversion from unsigned to two's complement.



Conversions Between Signed and Unsigned (III)

- Generally, most numbers are **signed by default**.
  - When declaring a constant such as `12345` or `0x1A2B`, the value is considered signed.
  - Adding character `U` or `u` as a suffix creates an unsigned constant, e.g., `12345U` or `0x1A2Bu`.
  - Suffix indicates the type.
    - `int`:- No suffix are required because integer constant are by default assigned as `int` data type.
    - `unsigned int`: `u` or `U`.
    - `long int`: `l` or `L`.
    - `unsigned long int`: `ul` or `UL`.
    - `long long int`: `ll` or `LL`.
    - `unsigned long long int`: `ull` or `ULL`.
- When an operation is performed where one operand **is signed and the other is unsigned**, C implicitly **casts the signed argument to unsigned** and performs the operations assuming the numbers are nonnegative.
  - Including comparison operations `<`, `>`, `==`, `<=`, `>=`

Expanding the Bit Representation of a Number (I)

- To convert an **unsigned number to a larger data type**, it **adds leading zeros to the representation**
  - This operation is known as **zero extension**.
- For converting a **two's- complement number to a larger data type** it copies of the most significant bit to the representation.
  - This operation is known as **sign extension**.

```
short x = 15213;
int ix = (int)x; /* expand to 32-bit value */
short y = -15213;
int iy = (int)y; /* expand to 32-bit value */
```

	Decimal	Hexadecimal	Binary
x	15213	0x3B 6D	00111011 01101101
ix	15213	0x00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	0xC4 93	11000100 10010011
iy	-15213	0xFF FF C4 93	11111111 11111111 11000100 10010011

Expanding the Bit Representation of a Number (II)

- First, change the **size**
  - zero extension, for unsigned number
  - sign extension, for signed number
- Second, converts from **signed to unsigned**.
- This convention is required by the C standards.

```
short sx = -12345;
unsigned short ux = sx;
unsigned int uy = sx;
```

	Decimal	Hexadecimal	Binary
sx	-12345	0xCF C7	11001111 11000111
ux	53191	0xCF C7	11001111 11000111
uy	4294954951	0xFF FF CF C7	11111111 11111111 11001111 11000111

Truncating Numbers (I)

- During datatype conversion, if operand on the right hand side is of higher rank type (more bytes) then it becomes the type of left hand side operand which is of lower rank (less bytes).

```
int x = 15213;
short sx = (short)x;
```

  - This kind of type conversion from higher rank to lower rank is called **Demotion**.
- **Demotion represents a potential problem with truncation or unpredictable results often occurring**.
  - How do you fit an integer value of 456 into a `char` variable?
  - How do you fit the floating-point value of 45656.453 into an `int` variable?
  - Most **compilers give a warning if it detects demotion** happening.
  - A **compiler warning does not stop the compilation process**.
    - It does warn the programmer to check to see if the demotion is reasonable.

Truncating Numbers (II)

- When truncating a w-bit number to a k-bit number ($w > k$), we **drop the high-order $w - k$ bits**.
- Reinterpret the number.
- **Truncating a number can alter its value**.

```
int x = 53191;
short sx = (short) x;
int y = sx;
```

	Decimal	Hexadecimal	Binary
x	53191	0x00 00 CF C7	00000000 00000000 11001111 11000111
sx	-12345	0xCF C7	11001111 11000111
y	-12345	0xFF FF CF C7	11111111 11111111 11001111 11000111

- For an unsigned number x, the result of truncating it to k bits is equivalent to computing $x \bmod 2^k$.

```
short x = 300;          //0b00000001 00101100
char y = (char) x;//300 % 256 = 44 0b00101100
```

1 Computer

2 Vulnerabilities

3 Memory

4 C Data types

5 C Pointers

6 Integers

7 **Bit-Level Operations**

8 Logical Operations

Bitwise operators (I)

- C supports bitwise boolean operations on a set of bits (bit vector) such as `char`, `int` and so on.

Operator	Meaning
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise exclusive OR operator
~	Binary One's Complement Operator is a unary operator
< <	Left shift operator
> >	Right shift operator

Bitwise operators (II)

- The bitwise logical operators **work on the data bit by bit**, starting from the LSB, which is the rightmost bit, working towards the MSB , which is the leftmost bit.
- Shifting is called to be **Logical** when **performed on unsigned integers** in either direction, either right or left.
  - In Logical Shift, during shifting either side, **empty slots are padded with Zeros**.
- Shifting is called to be **Arithmetic** when **performed on signed integers** in either direction, either right or left.
  - An arithmetic shift right replicates the sign bit in MSB, instead of filling in with 0's as in the logical shift right .
  - The arithmetic shift left operation is identical to logical shift left.

1 Computer

2 Vulnerabilities

3 Memory

4 C Data types

5 C Pointers

6 Integers

7 Bit-Level Operations

8 Logical Operations

Logical Operations

- C also provides a set of logical operators `||`, `&&`, and `!`, which correspond to the **Or**, **And**, and **Not** operations of logic.
  - These can easily be confused with the bit-level operations, but their function is quite different.
- The logical operations treat any nonzero argument as representing **True** and argument 0 as representing **False**.
  - They return either `1` or `0`, indicating a result of either **True** or **False**, respectively.
- An important distinction between the logical operators versus their bit-level counterparts is that the **logical operators do not evaluate their second argument if the result of the expression can be determined by evaluating the first argument**.
  - Thus, for example, the expression `a && 5/a` will never cause a division by zero, and the expression `p && *p++` will never cause the dereferencing of a null pointer.

Logical *vs.* Bit-level Operations

And

```
char A = 40;   char B = 30;  char C;
```

C = A && B; (Logical)	C = A & B; (Bitwise)

C=1;

```
A  b00101000
&
B  b00011110
-----------
C= b00001000
```

Bitwise operation

Not

```
char A = 40;   char B = 30;  char C;
```

C = ! A;	C = ~A;

C=0;

```
A  b00101000
C= ~A b11010111
C= -41
```

Bitwise operation

Or

```
char A = 40;   char B = 30;  char C;
```

C = A || B;	C = A | B;

C=1;

```
A  b00101000
|
B  b00011110
-----------
C= b00111110
C= 62
```

Bitwise operation