POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

Pointer Data Type

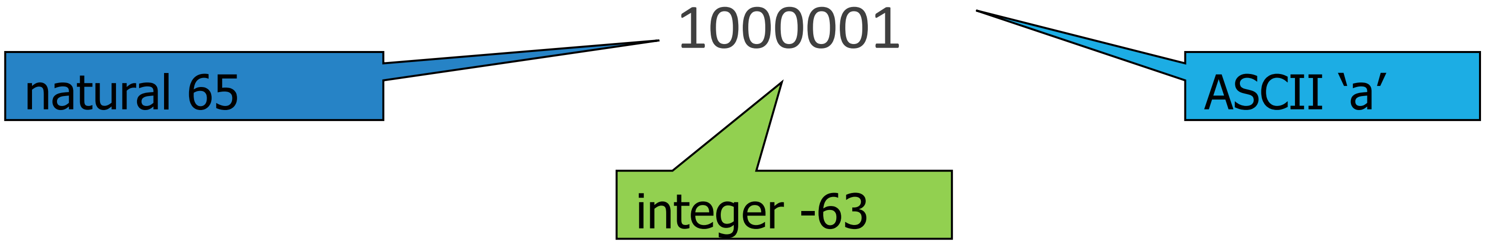
POINTERS AND DYNAMIC DATA STRUCTURES: MEMORY ALLOCATION AND MODULARITY IN C LANGUAGE

Data in the central memory

Data stored as sequences of 1s and 0s encoding symbols of finite sets

- natural, integer, rational, characters

The sequence has meaning only if associated with the corresponding encoding :



1000001

natural 65

integer -63

ASCII ‘a’

The memory model

- **RAM Memory**: matrix of bits with *n* rows and *m* columns.

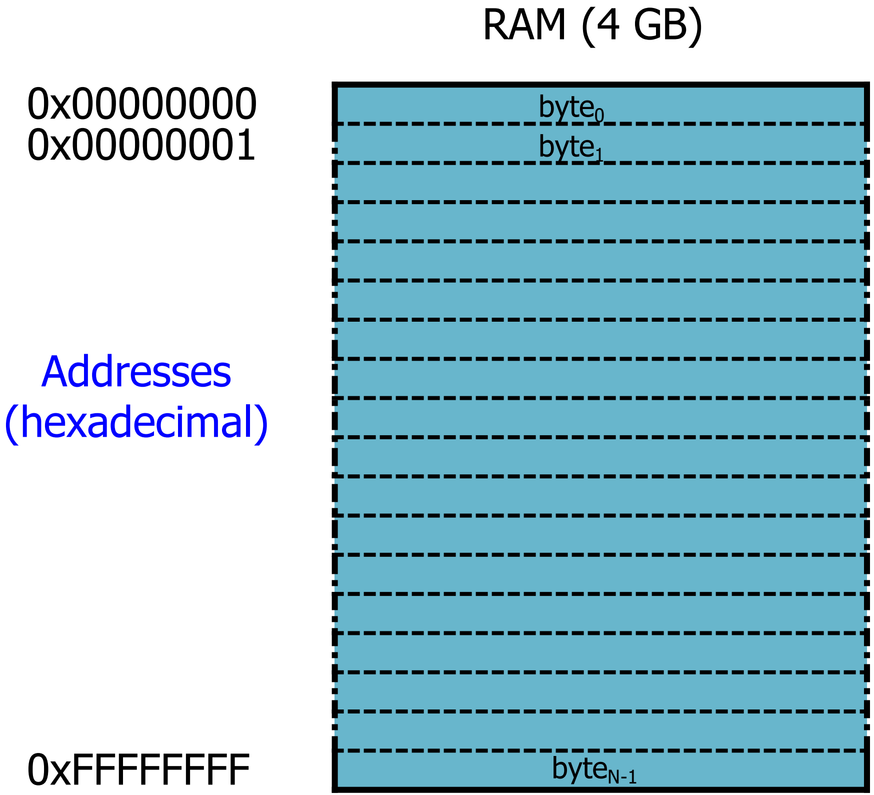
Example. 128 bit matrix:

- 32 rows x 4 columns
- 16 rows x 8 columns
- 8 rows x 16 columns

In general:

- *n* is a power of 2
- *m* is a multiple of 8 (1 byte = 8 bit)

Example of RAM



RAM (4 GB)

0x00000000 $byte_0$

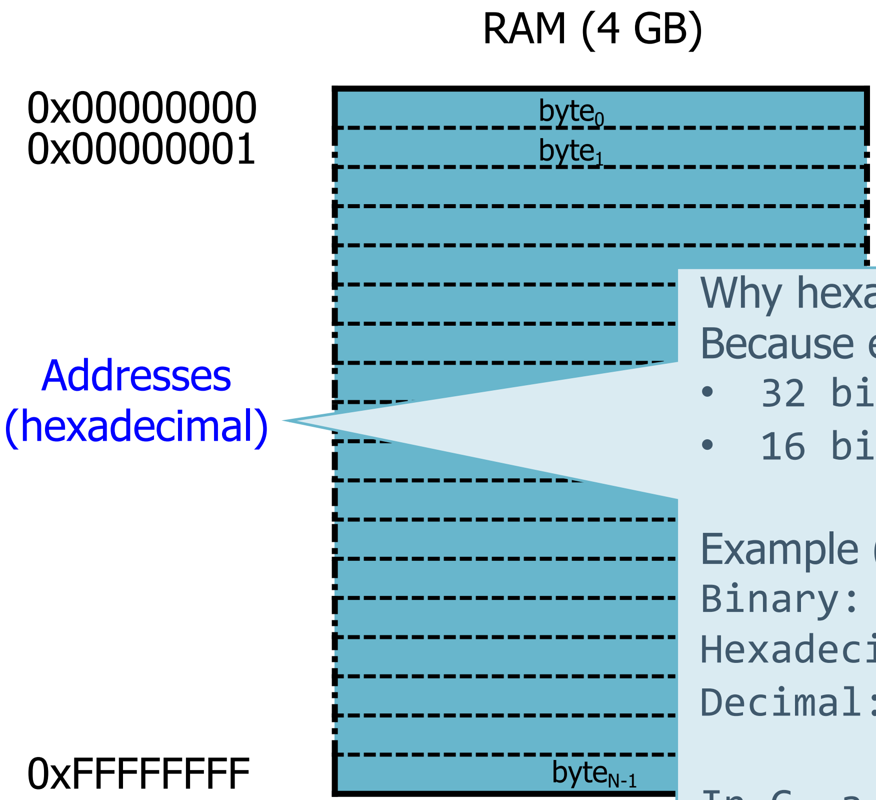
0x00000001 $byte_1$

Addresses (hexadecimal)

0xFFFFFFFF $byte_{N-1}$

- Memory of 4 GB (N=2^{32} byte)
- byte-addressable
- cells and words of 1 byte

Example of RAM



- Memory of 4 GB (N=2^{32} byte)
- byte-addressable
- cells and words of 1 byte

Why hexadecimal?
Because each digit corresponds to 4 bit

- 32 bit -> 8 hexadecimal digits
- 16 bit -> 4 hexadecimal digits

Example (16 bit):

```
Binary: 0110 0000 1010 0101
Hexadecimal: 60A5
Decimal: 24741
```

In C, a hexadecimal constant begins with 0x

Cell and word

- **Cell:**
  - Smallest group of k bits that are accessed unitarily
  - in general $k = 8 \Rightarrow 1$ byte
  - cell of 1 byte ⇒ byte-addressable memory
  - identified by an address: N cells ⇒ addresses in a 0 to N-1 range
  - address: string of $\lceil \log_2 N \rceil$ bit
- **Word:**
  - dimension of data handled by the processor
  - group of cells (in general it takes 4 or 8 byte)
  - for efficiency, read/write operations are made on words, non on cells
  - it can be in 1 row or several adjacent rows
  - rarely RAM is word-addressable, usually always byte-addressable

Big/Little Endian

Words on more than one cell:

- **Big Endian** (i.e., left to right):
  - The Most Significant Byte takes the lowest memory address
  - The Least Significant Byte takes the highest memory address
- **Little Endian** (i.e., right to left):
  - The Most Significant Byte takes the highest memory address
  - The Least Significant Byte takes the lowest memory address

- Memory of 4 GB
- byte-addressable
- cells of 1 byte

- words of 4 byte
- datum 0x123A5FF4 at address 0x3A7F2018

	0	1	2	3
0x00000000	byte$_0$	byte$_1$	byte$_2$	byte$_3$
0x00000004	byte$_4$	byte$_5$	byte$_6$	byte$_7$
		…		
0x3A7F2018	12	3A	5F	F4
		…		
0xFFFFFFFC	byte$_{N-4}$	byte$_{N-3}$	byte$_{N-2}$	byte$_{N-1}$

Big Endian

	0	1	2	3
0x00000000	byte$_3$	byte$_2$	byte$_1$	byte$_0$
0x00000004	byte$_7$	byte$_6$	byte$_5$	byte$_4$
		…		
0x3A7F2018	F4	5F	3A	12
		…		
0xFFFFFFFC	byte$_{N-1}$	byte$_{N-2}$	byte$_{N-3}$	byte$_{N-4}$

Little Endian

- Memory o
- byte-addre
- cells of 1 byte

One row represents 4 byte: lowest addresses on the left, highest on the right

datum 0x123A5FF4 at
address 0x3A7F2018

	0	1	2	3
0x00000000	byte$_0$	byte$_1$	byte$_2$	byte$_3$
0x00000004	byte$_4$	byte$_5$	byte$_6$	byte$_7$
		…		
0x3A7F2018	12	3A	5F	F4
		…		
0xFFFFFFFC	byte$_{N-4}$	byte$_{N-3}$	byte$_{N-2}$	byte$_{N-1}$

Big Endian

0	1	2	3
byte$_3$	byte$_2$	byte$_1$	byte$_0$
byte$_7$	byte$_6$	byte$_5$	byte$_4$
	…		
F4	5F	3A	12
	…		
byte$_{N-1}$	byte$_{N-2}$	byte$_{N-3}$	byte$_{N-4}$

Little Endian

- Memory
- byte-addr t
- cells of 1 byte addre /F2018

One row represents 4 byte: lowest addresses on the left, highest on the right

	0	1	2	3
0x00000000	byte$_0$	byte$_1$	byte$_2$	byte$_3$
0x00000004	byte$_4$	byte$_5$	byte$_6$	byte$_7$
		…		
0x3A7F2018	12	3A	5F	F4
		…		
0xFFFFFFFC	byte$_{N-4}$	byte$_{N-3}$	byte$_{N-2}$	byte$_{N-1}$

Big Endian

	0	1	2	3
0x00000000	byte$_3$	byte$_2$	byte$_1$	byte$_0$
0x00000004	byte$_7$	byte$_6$	byte$_5$	byte$_4$
		…		
0x3A7F2018	F4	5F	3A	12
		…		
0xFFFFFFFC	byte$_{N-1}$	byte$_{N-2}$	byte$_{N-3}$	byte$_{N-4}$

Little Endian

Most significant byte: 12

- cells of 1 byte

- words of 4 byte
- datum 0x123A5FF4 at address 0x3A7F2018

	0	1	2	3
0x00000000	byte$_0$	byte$_1$	byte$_2$	byte$_3$
0x00000004	byte$_4$	byte$_5$	byte$_6$	byte$_7$
	…			
0x3A7F2018	12	3A	5F	F4
	…			
0xFFFFFFFC	byte$_{N-4}$	byte$_{N-3}$	byte$_{N-2}$	byte$_{N-1}$

Big Endian

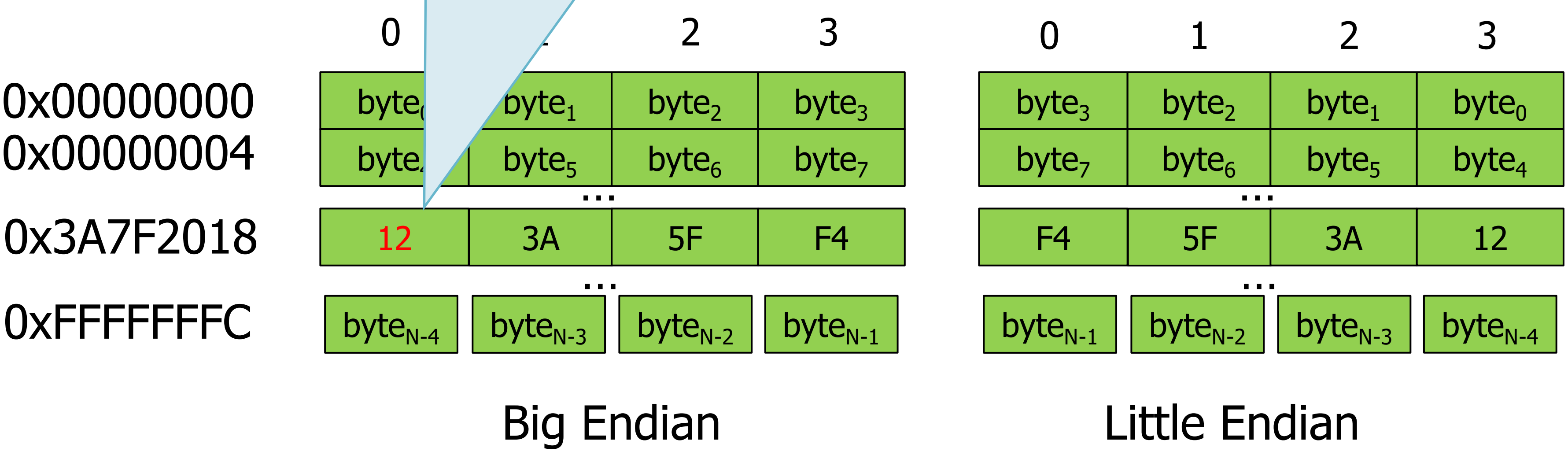
	0	1	2	3
0x00000000	byte$_3$	byte$_2$	byte$_1$	byte$_0$
0x00000004	byte$_7$	byte$_6$	byte$_5$	byte$_4$
	…			
0x3A7F2018	F4	5F	3A	12
	…			
0xFFFFFFFC	byte$_{N-1}$	byte$_{N-2}$	byte$_{N-3}$	byte$_{N-4}$

Little Endian

- Memory of 4 GB
- byte addressable

- words of 4 byte
- datum 0x123A5FF4 at address 0x3A7F2018

Most significant byte: 12



	0	1	2	3
0x00000000	byte$_0$	byte$_1$	byte$_2$	byte$_3$
0x00000004	byte$_4$	byte$_5$	byte$_6$	byte$_7$
	…			
0x3A7F2018	12	3A	5F	F4
	…			
0xFFFFFFFC	byte$_{N-4}$	byte$_{N-3}$	byte$_{N-2}$	byte$_{N-1}$

Big Endian

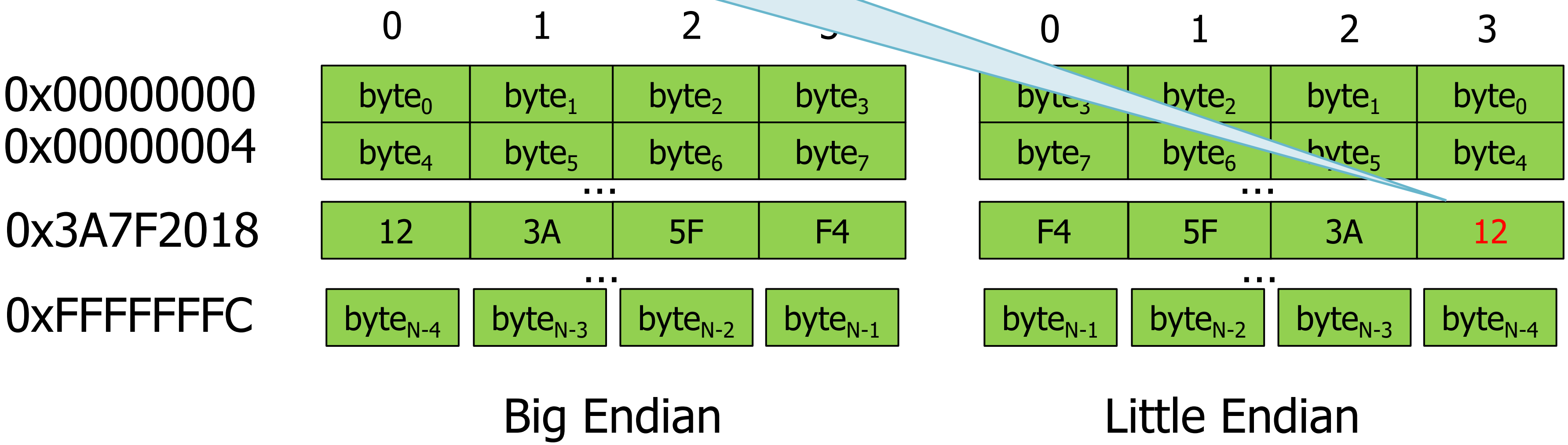
0	1	2	3
byte$_3$	byte$_2$	byte$_1$	byte$_0$
byte$_7$	byte$_6$	byte$_5$	byte$_4$
…			
F4	5F	3A	12
…			
byte$_{N-1}$	byte$_{N-2}$	byte$_{N-3}$	byte$_{N-4}$

Little Endian

- Memory of 4 GB

- words of 4 byte
- datum 0x123A5FF4 at address 0x3A7F2018

Most significant byte: 12



Big Endian

	0	1	2	3
0x00000000	byte$_0$	byte$_1$	byte$_2$	byte$_3$
0x00000004	byte$_4$	byte$_5$	byte$_6$	byte$_7$
		…		
0x3A7F2018	12	3A	5F	F4
		…		
0xFFFFFFFC	byte$_{N-4}$	byte$_{N-3}$	byte$_{N-2}$	byte$_{N-1}$

Little Endian

	0	1	2	3
0x00000000	byte$_3$	byte$_2$	byte$_1$	byte$_0$
0x00000004	byte$_7$	byte$_6$	byte$_5$	byte$_4$
		…		
0x3A7F2018	F4	5F	3A	12
		…		
0xFFFFFFFC	byte$_{N-1}$	byte$_{N-2}$	byte$_{N-3}$	byte$_{N-4}$

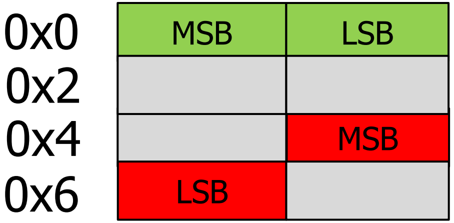
Memory alignment

“alignment”: putting the data in memory at an initial address equal to some multiple of the word size, to increase efficiency.

Example:

Memory of 8 cells of 1 byte and word of 2 byte, Big Endian

aligned, **not aligned**



0x0	🟩 MSB	🟩 LSB
0x2		
0x4		🟥 MSB
0x6	🟥 LSB	

- Even in byte-addressable memories read/write operations are done at the word level
  - In principle, it is not necessary to address and read/write a byte, but to address and read/write a word
  - It is possible to address a single byte and read/write a word at a time
- **Data with dimension >= words are always aligned**, for efficiency reasons

Alignment of structures in C

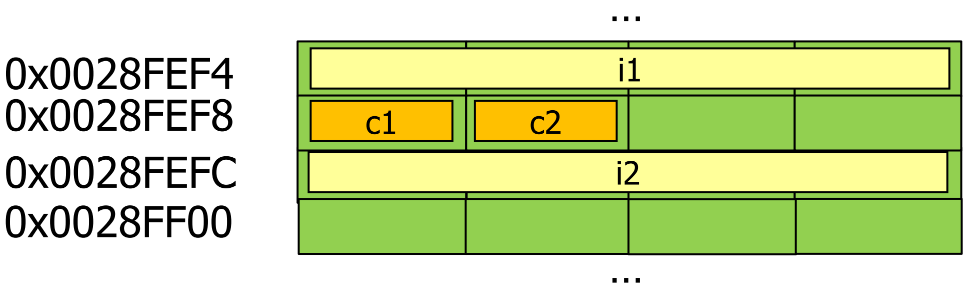
- Fields are stored in the same order they are declared
- Fields are aligned in memory along a boundary that is a multiple of their own size. That is:
    - *1-byte fields are not aligned*
    - *2-byte fields are aligned to multiples of two*
    - *4-byte fields are aligned to multiples of four*
    - *etc.*

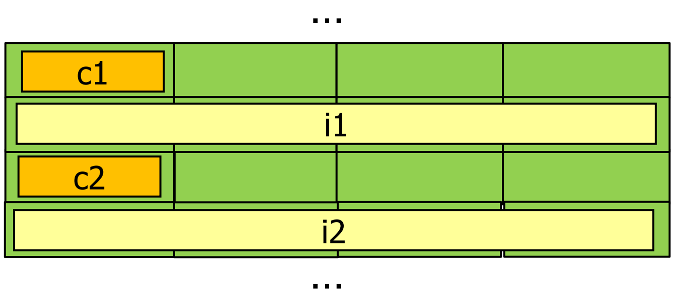
Example:

- memory of 4 GB byte-addressable with cells of 1 byte and words of 4 byte
- 2 `struct` with same fields in different order

```
typedef struct item1_s {
    int i1;
    char c1, c2;
    int i2;
} Item1;
```

```
typedef struct item2_s {
    char c1;
    int i1;
    char c2;
    int i2;
} Item2;
```



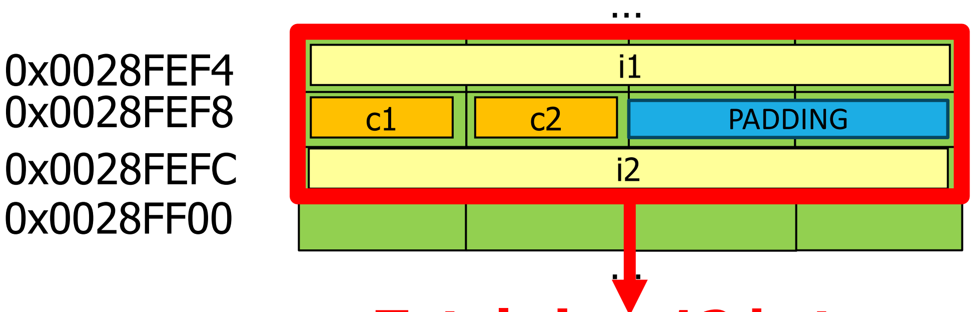


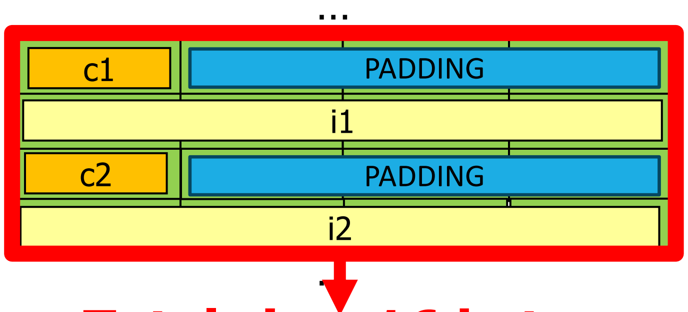
Example:

- memory of 4 GB byte-addressable with cells of 1 byte and words of 4 byte
- 2 `struct` with same fields in different order

```
typedef struct item1_s {
  int i1;
  char c1, c2;
  int i2;
} Item1;
```

```
typedef struct item2_s {
  char c1;
  int i1;
  char c2;
  int i2;
} Item2;
```





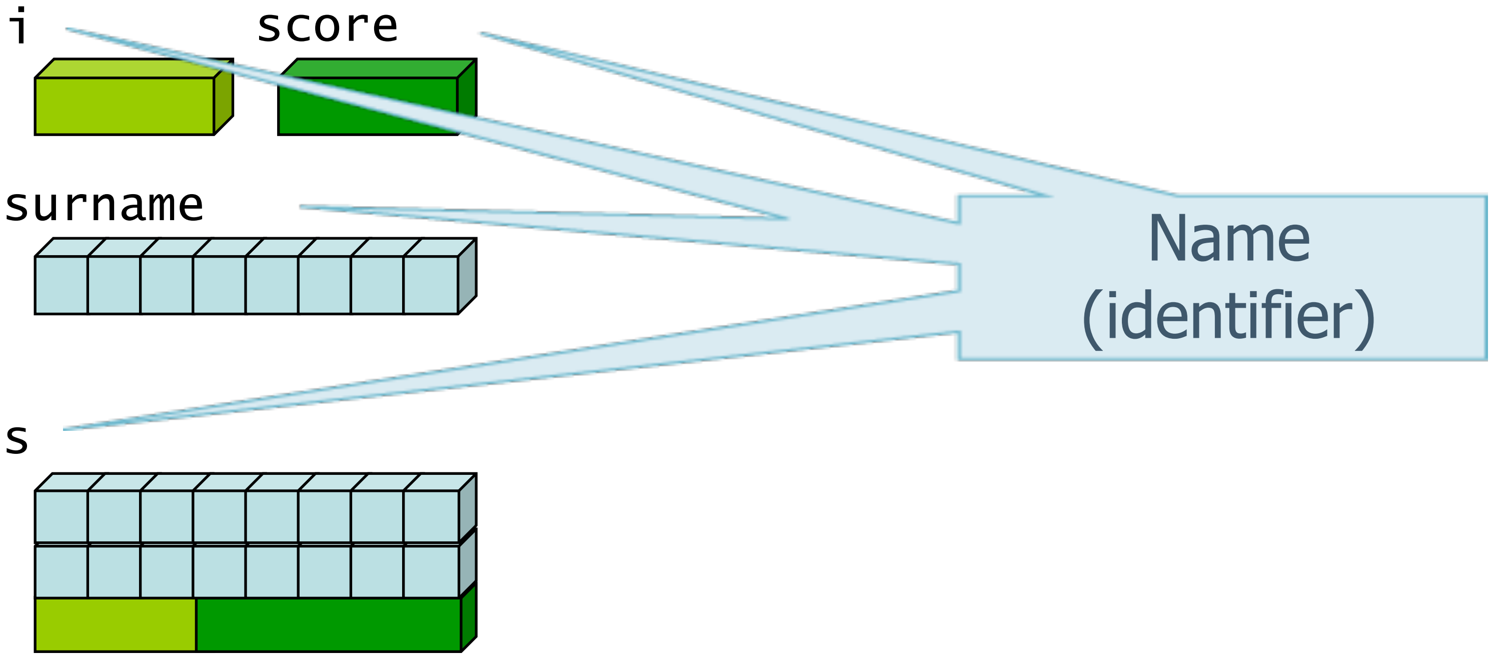
Total size: 12 byte

Total size: 16 byte

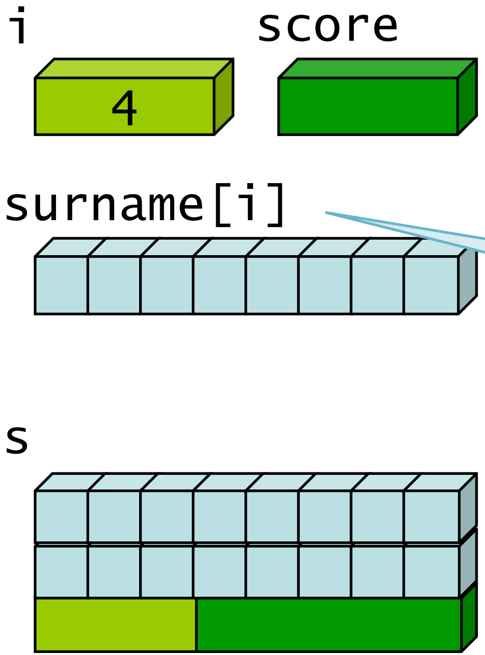
Variables

- Data in memory are stored in **containers** *(byte, words, word groups)* characterized by:
  - name (univocal identifier)
  - type
- If values can vary with time, the containers are called **variables** *(otherwise, **constant**)*
- Compiler/linker (and loader) **allocate** the variables at certain addresses, taking up 1 or more words. They maintain an identifier-address-type correspondence table.

How to identify a variable?

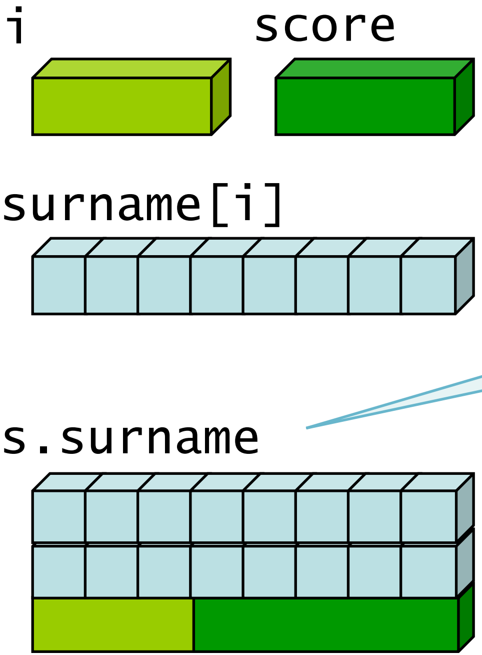


How to identify a variable?



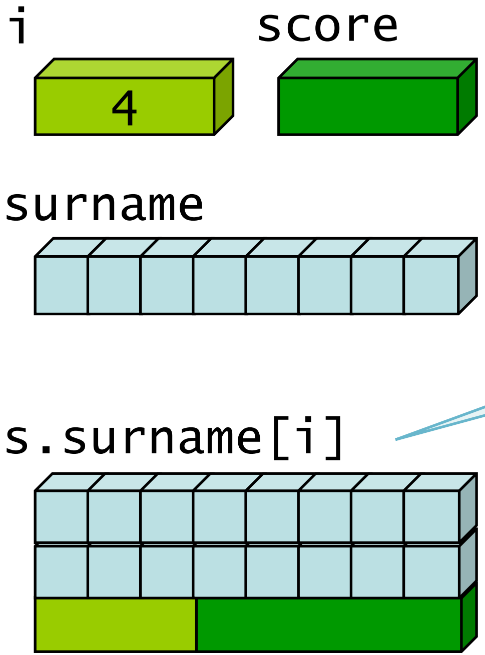
Name + index
(array)

How to identify a variable?



Sequence of names
(variable.field)

How to identify a variable?



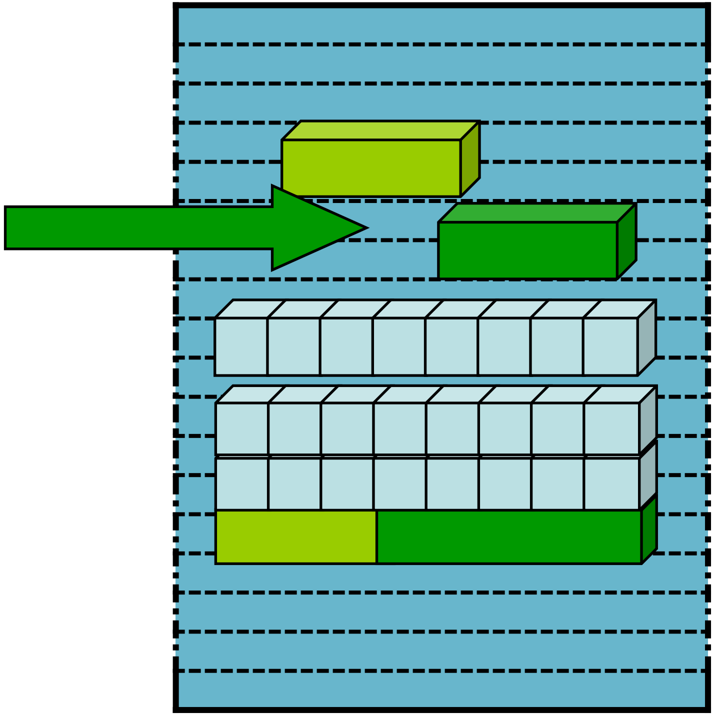
Sequence of names + index

The pointer

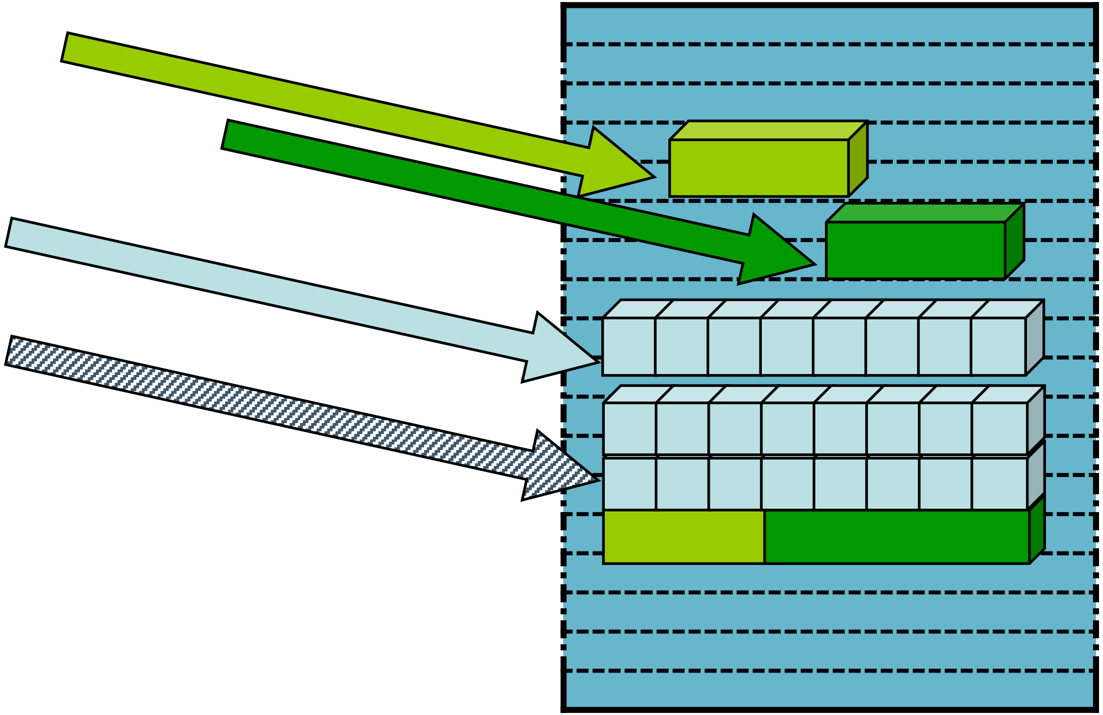
Way to access data in memory, alternative to variables.

It is a datum that contains a reference to another datum in memory, providing following necessary information:

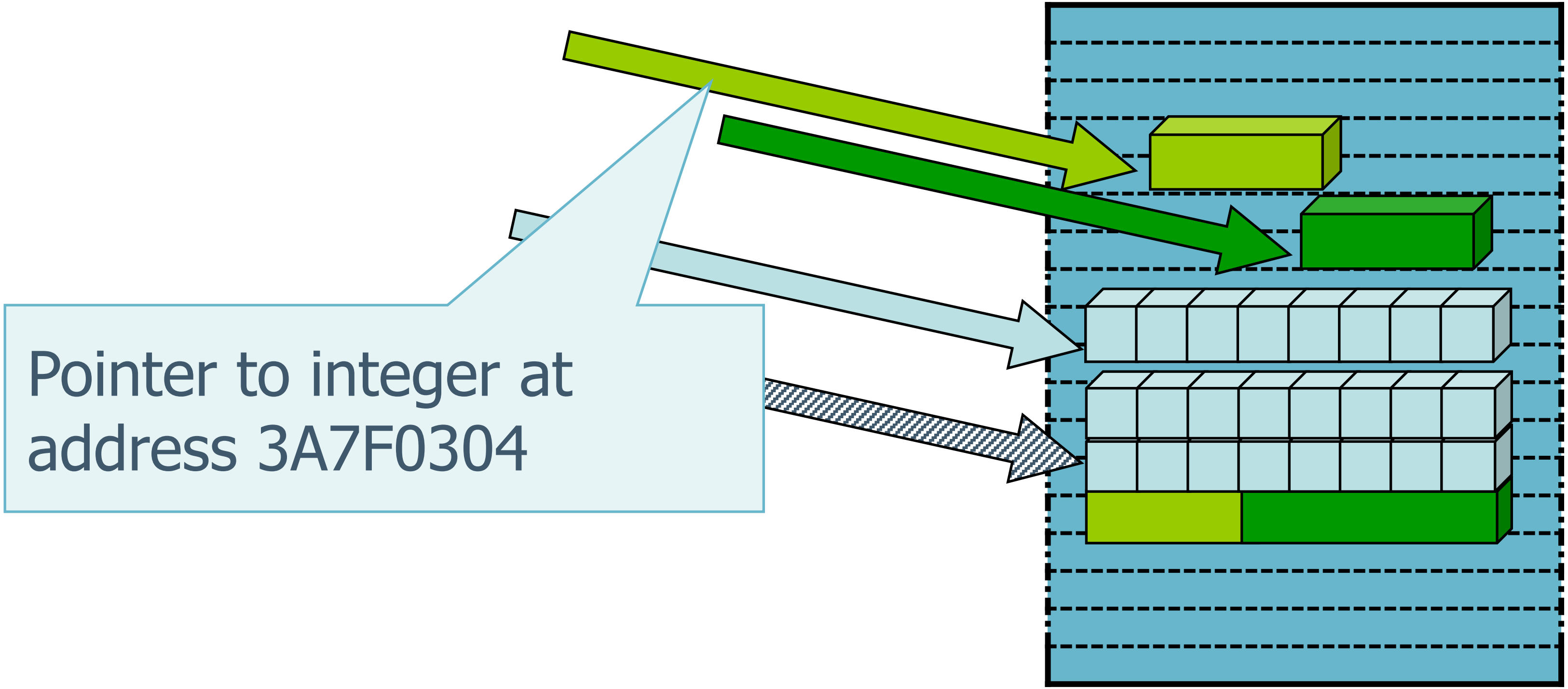
- Where the referenced datum is stored (address)
- How the referenced datum is encoded (type)



The pointer

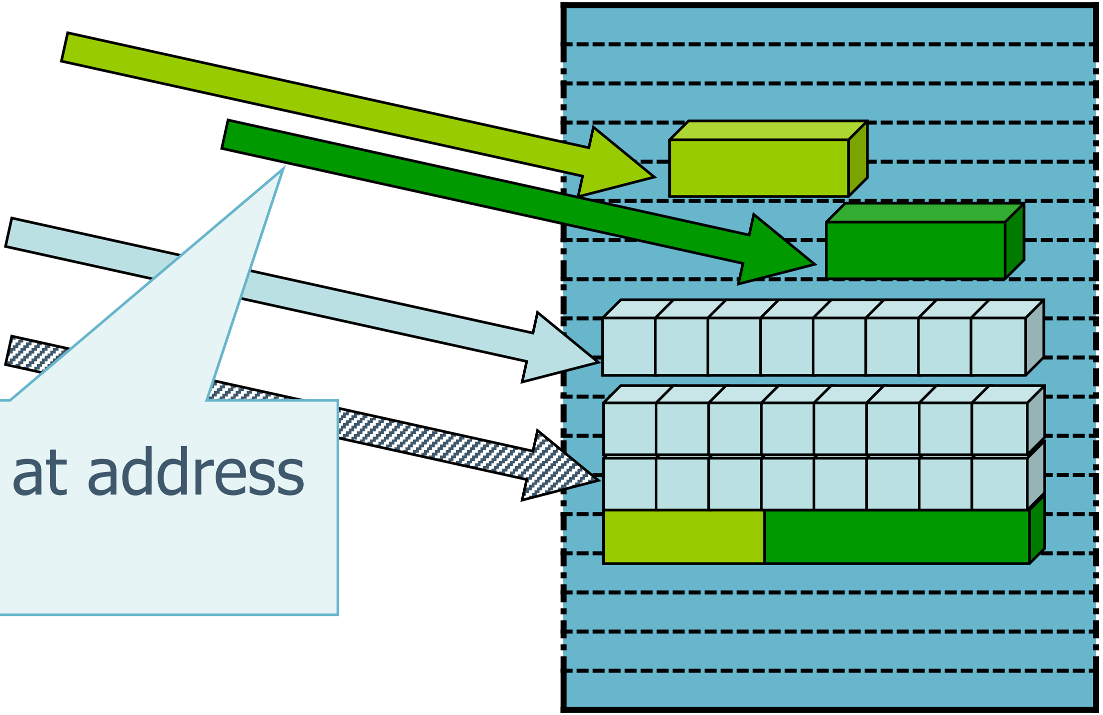


The pointer



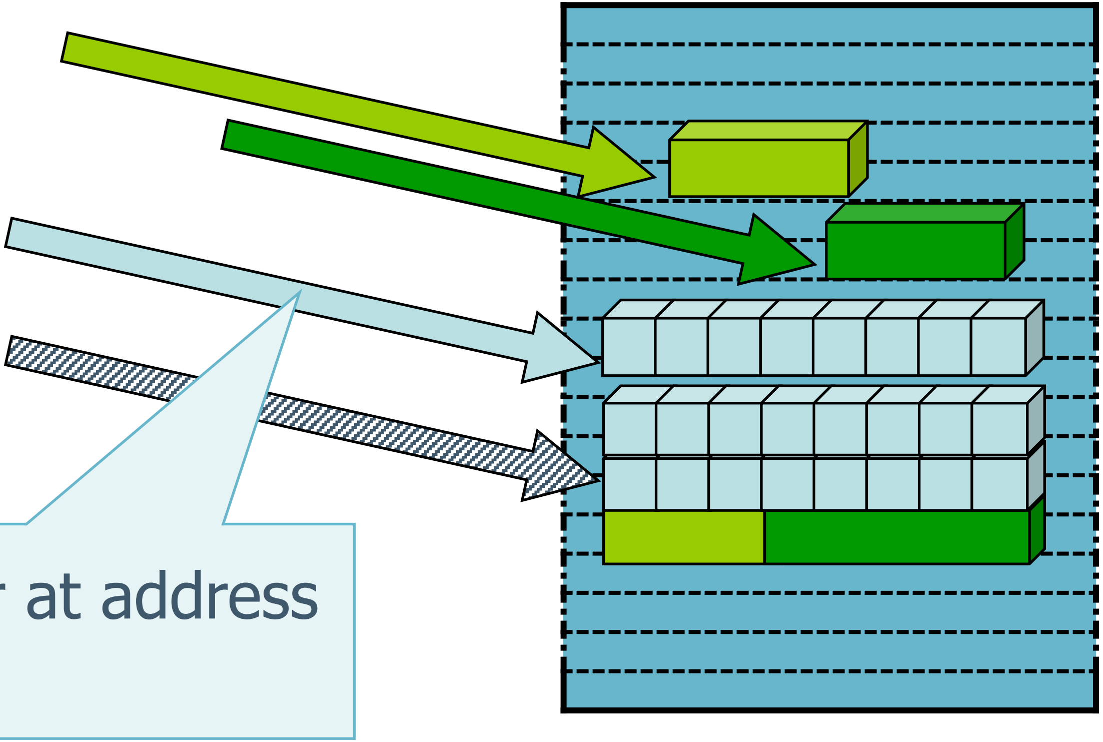
Pointer to integer at address 3A7F0304

The pointer



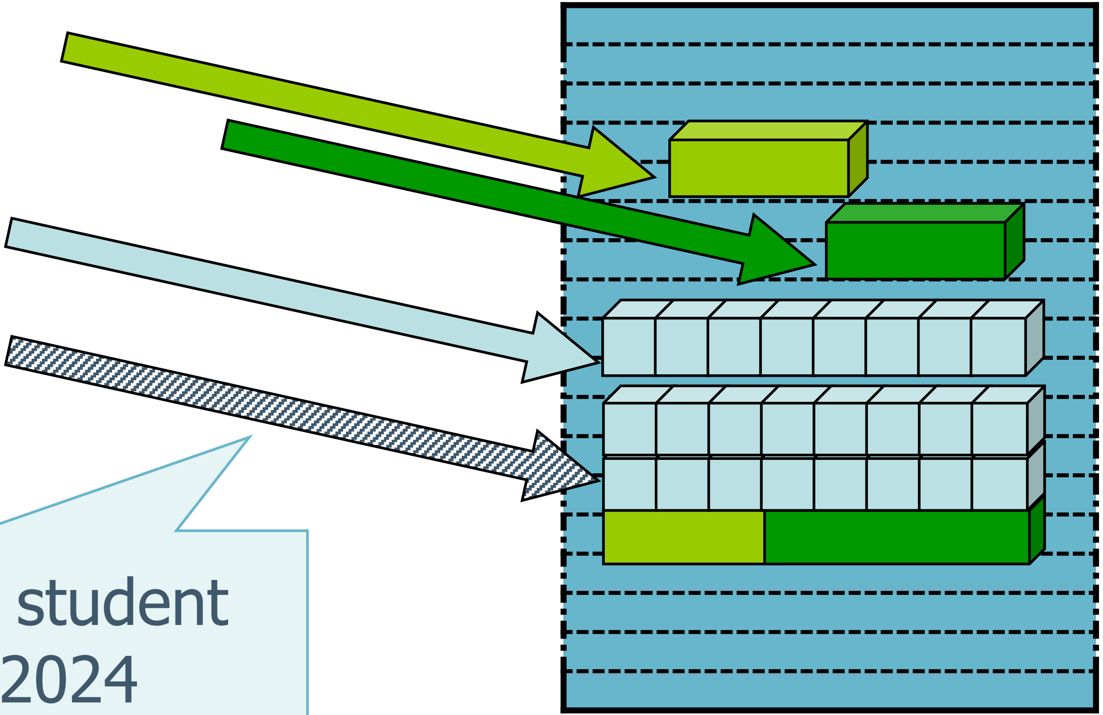
Pointer to float at address 3A7F2016

The pointer



Pointer to char at address 3A7F2020

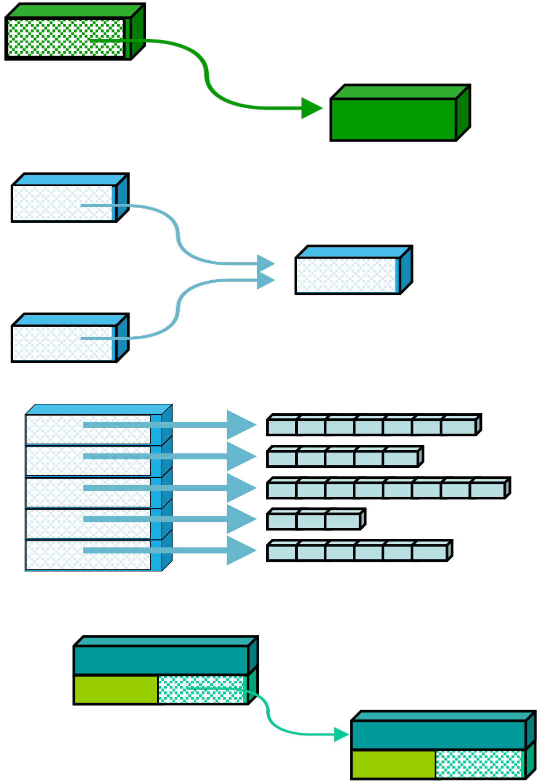
The pointer



Pointer to struct student at address 3A7F2024

The pointer

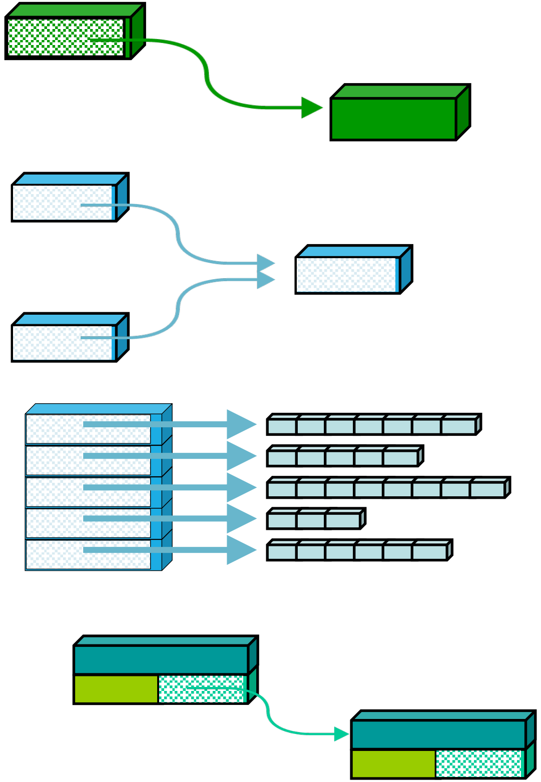
Unlike identifiers (which cannot be modified), pointers are manipulatable information (they can be calculated, modified, assigned)



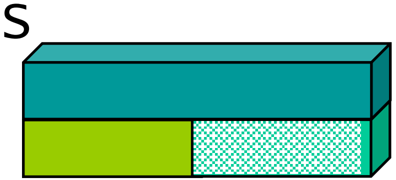
The pointer

Unlike identifiers (which cannot be modified), pointers are manipulatable information (they can be calculated, modified, assigned)

Novelty: a pointer is itself a datum, that points to another datum!

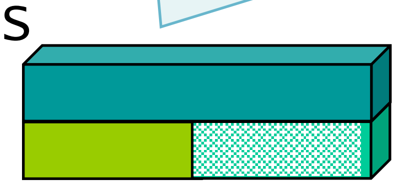


Operators: reference

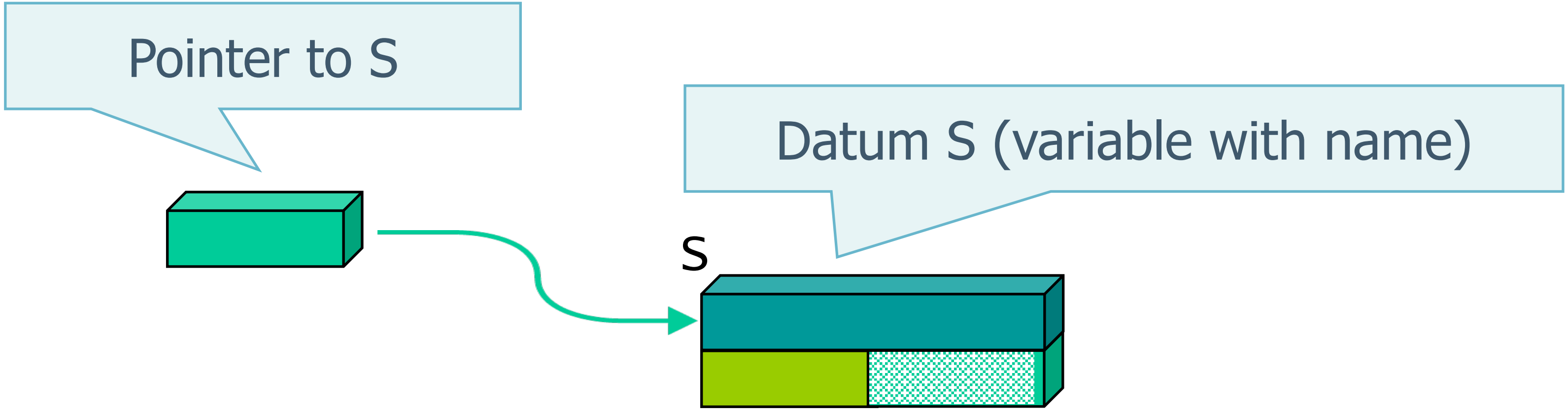


Operators: reference

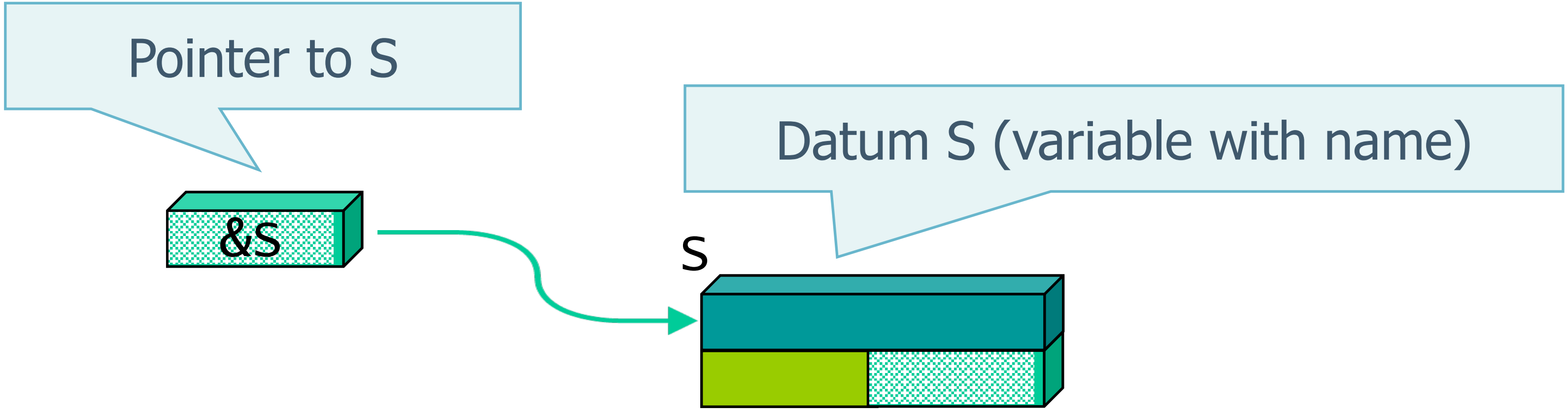
Datum S (variable with name)



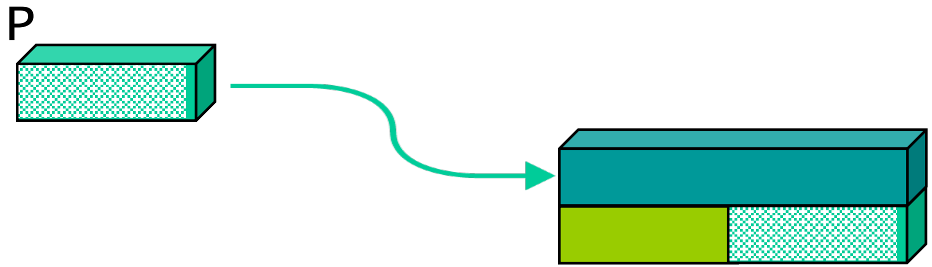
Operators: reference



Operators: reference

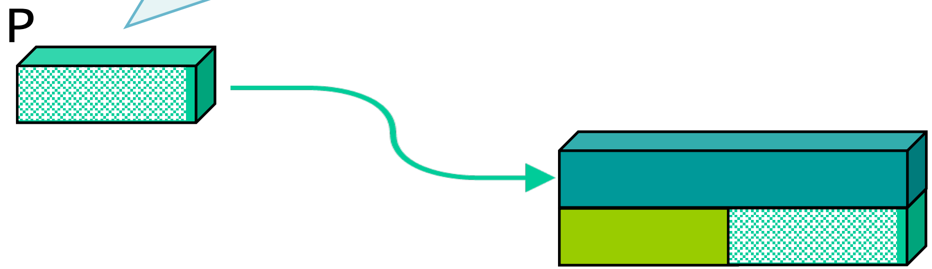


Operators: dereference

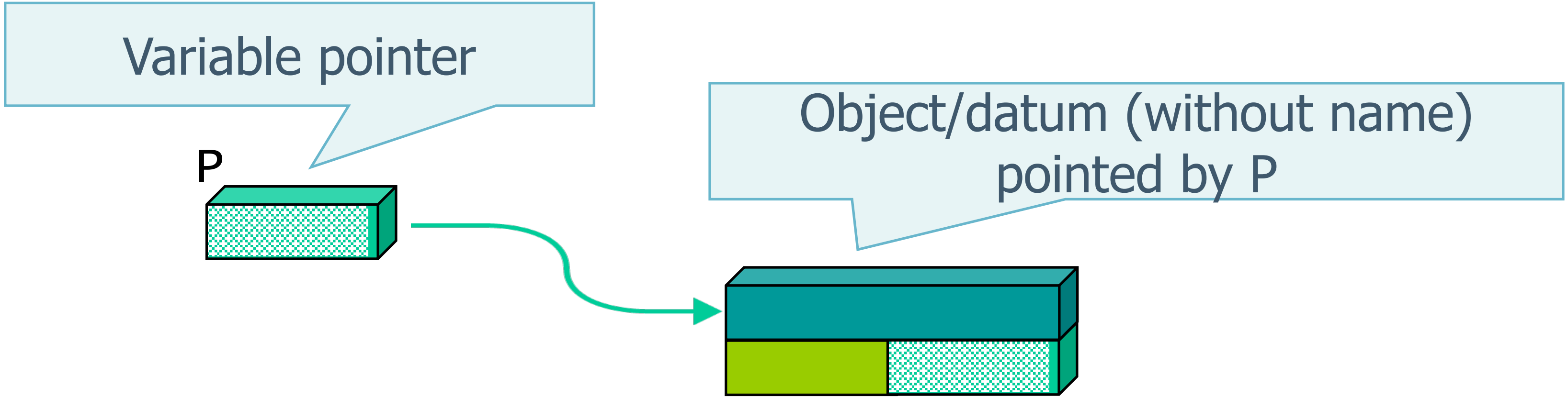


Operators: dereference

Variable pointer



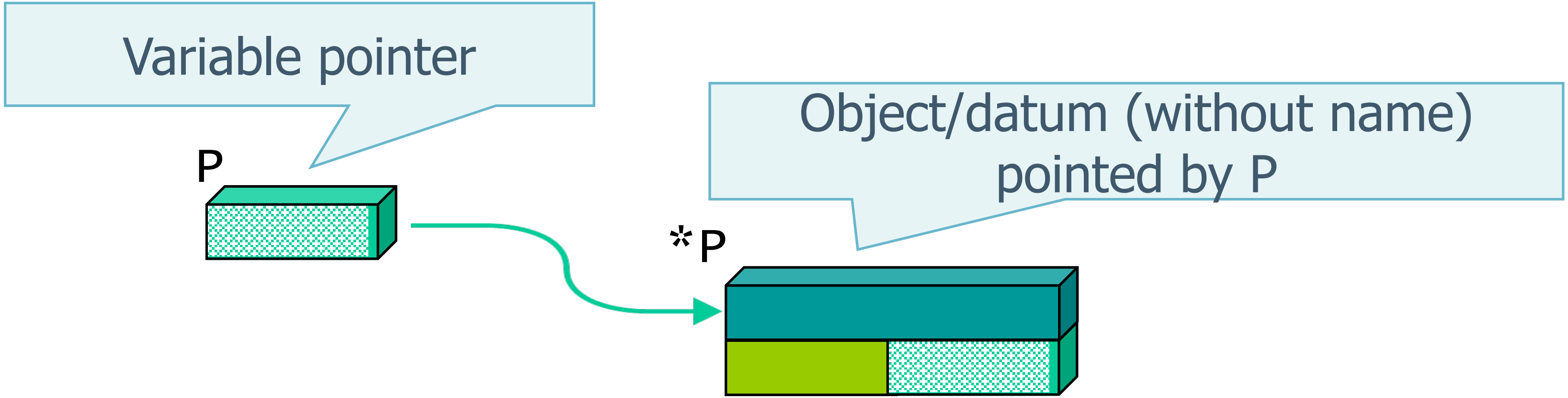
Operators: dereference



Variable pointer

Object/datum (without name) pointed by P

Operators: dereference



Variable pointer

Object/datum (without name) pointed by P

* and &

- * and & symbols are used, in definitions and usage of pointers, in a pre-fixed format to indicate
  - *… : datum pointed by …
  - &… : pointer to …
- The deferencing * and referencing & are dual operators.

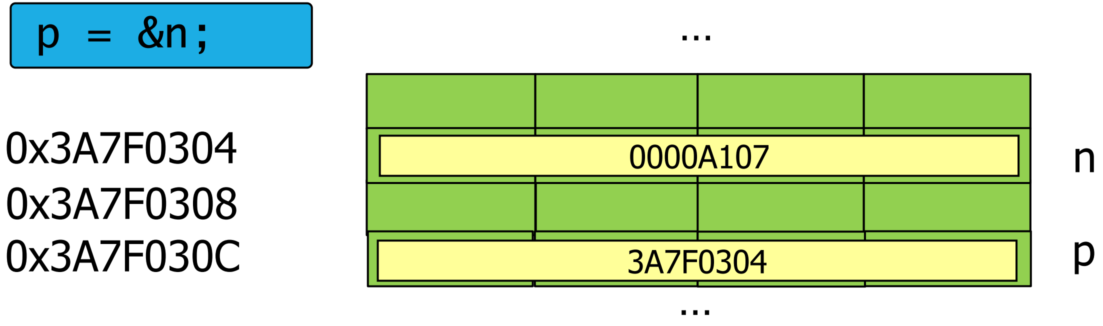
Example

- Integer variable n = 41223 (=0x0000A107) at address 0x3A7F0304
- Variable pointer to integer p (already declared) at address 0x3A7F030C
- 4 GB memory, byte-addressable, 1 byte cells, 4 byte words

Example

- Integer variable n = 41223 (=0x0000A107) at address 0x3A7F0304
- Variable pointer to integer p (already declared) at address 0x3A7F030C
- 4 GB memory, byte-addressable, 1 byte cells, 4 byte words

```
p = &n;
```



...

0x3A7F0304 | 0000A107 | n

0x3A7F0308

0x3A7F030C | 3A7F0304 | p

...

```
printf("n: %d\n", n);
printf("n: %d\n", *p);
```

are equivalent

```
scanf("%d", &n);
scanf("%d", p);
```

are equivalent

Declaration

- A declaration of a variable pointer requires the reference to an elementary data type (the one of the pointed datum)

```
int *px;
char *p0, *p1;
struct student *pstud;
FILE *fp;
```

Declaration

- A declaration of a variable pointer requires the reference to an elementary data type (the one of the pointed datum)

Variable px di of type "pointer to integer"

```
int *px;
char *p0, *p1;
struct student *pstud;
FILE *fp;
```

Declaration

- A declaration of a variable pointer requires the reference to an elementary data type (the one of the pointed datum)

Variables p0 and p1 of type "pointers to char"

```
int *px;
char *p0, *p1;
struct student *pstud;
FILE *fp;
```

Declaration

- A declaration of a variable pointer requires the reference to an elementary data type (the one of the pointed datum)

Variable pstud of type “pointer to struct student”

```
int *px;
char *p0, *p1;
struct student *pstud;
FILE *fp;
```

Declaration

- A declaration of a variable pointer requires the reference to an elementary data type (the one of the pointed datum)

Variable fp of type "pointer to FILE"

```
int *px;
char *p0, *p1,
struct student *pstud;
FILE *fp;
```

The declaration

```
int *px;
```

can be interpreted in two ways:

1) `*px` (datum pointed by `px`) will be (!) of type int

   NB: the variable `px`, at the moment of declaration, DOES NOT contain a datum (i.e., a pointer) yet. The pointed datum does not exist yet, it will exist only after the first assigment!

2) `int *` (type pointer to int) is the type of the variable `px`

The declaration of a pointer can be done in two different ways, based on how the spaces are placed:

a) <base type>   *<identifier>;
   asterisk is next to the identifier

```
int *px;
```

b) <base type> *   <identifier>;
   spaces between asterisk and identifier

```
int *   px;
```

or

```
int*   px;
```

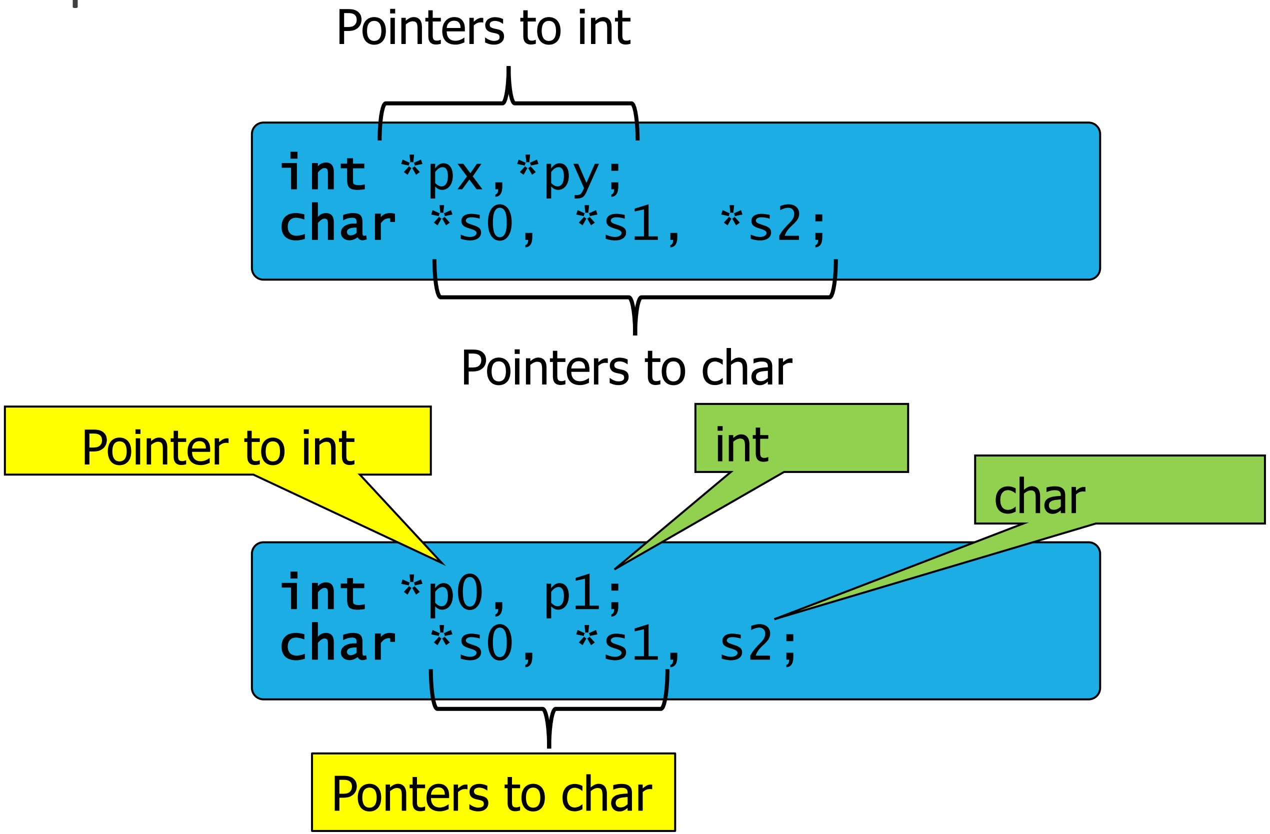
Factored declaration

- The declaration of more than a pointer variable of the same base type in the same instruction follows the (a) strategy:

  <base type> *<id_1>, *<id_2>…, *<id_n>;

- The base type is specified only once, but we need to prefix the asterisk to each declared variable

Example



Pointers to int

```
int *px,*py;
char *s0, *s1, *s2;
```

Pointers to char

Pointer to int

int

char

```
int *p0, p1;
char *s0, *s1, s2;
```

Ponters to char

Declaration with initialization

- You can assign a value to a variable pointer contextually to its declaration
  - Examples:

```
int x=0;
int *p = &x;
char *s = NULL;
```

Declaration with initialization

- You can assign a value to a variable pointer contextually to its declaration
  - Examples:

```
int x=0;
int *p = &x;
char *s = NULL;
```

or (equivalent declarations)

```
int x, *p = &x;
char *s = NULL;
```

The NULL constant

- The value actually assigned to a variable pointer is a memory address
- There is a constant that can be used as a “null pointer” (that is, the “zero” equivalent of pointer data types). This constant corresponds to the integer value 0
- The symbolic constant **NULL** (defined in `<stdio.h>`) can be used for this purpose

The void * type

- A generic pointer can be defined in C by referencing a type **void ***
- A generic pointer (`void *`) can be converted (and assigned) to a pointer of any other type (ex. `int *`)

```
int *px;
char *s0;
void *generic;
...
generic = px;
...
s0 = generic;
```

Assignment

- Up to this point we only discussed DECLARATIONS of variables pointer
- What about assignment? WHAT DO WE ASSIGN?
  - A memory address to a variable pointer?
  - A value to a variable pointed by a pointer?
- Two types of assignments:
  - **Pointer as a datum**: you assign to a variable pointer the result of an expression that computes a pointer/address (of a correct data type)
  - **Pointer as a reference**: you assign to the datum (variable) pointed (by a pointer) a value that is compatible with the data type

Pointer as datum: examples

```
p = &x;
s = p;
pname = &(stud.name);
p_i = &data[i];
```

Pointer as datum: examples

```
p = &x;
s = p;
pname = &(stud.name);
p_i = &data[i];
```

Pointer to variable x

Pointer as datum: examples

```
p = &x;
s = p;
pname = &(stud.name);
p_i = &data[i];
```

Assignment between two pointers

Pointer as datum: examples

```
p = &x;
s = p;
pname = &(stud.name);
p_i = &data[i];
```

Pointer to a field of a struct

Pointer as datum: examples

```
p = &x;
s = p;
pname = &(stud.name);
p_i = &data[i];
```

Pointer to an element of an array

Pointer as reference: examples

```
*p = 3*(x+2);
*s = *p;
*p_i = *p_i+1;
```

Pointer as reference: examples

```
*p = 3*(x+2);
*s = *p;
*p_i = *p_i+1;
```

Assigns result of expression (int) to the datum pointed by p

Pointer as reference: examples

```
*p = 3*(x+2);
*s = *p;
*p_i = *p_i+1;
```

Copies the variable pointed by p in the variable pointed by s

Pointer as reference: examples

```
*p = 3*(x+2);
*s = *p;
*p_i = *p_i+1;
```

Increments the variable pointed by p_i

Relational operators == and !=

- A comparison between two pointers returns a true value if the pointers refer to the same datum (i.e., to the same memory address)

```
p1==p2
```

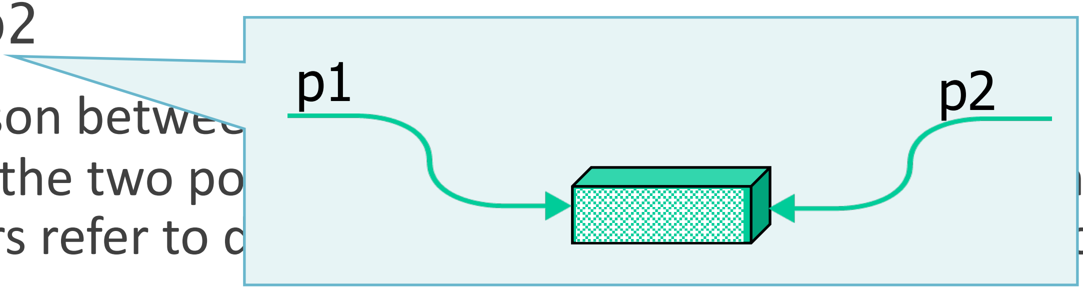
- A comparison between two pointed data returns a true value if the content of the two pointed variables are the same, even though the pointers refer to data that are stored in different memory locations

```
*p1==*p2
```

Relational operators == and !=

- A comparison between two pointers returns a true value if the pointers refer to the same datum (i.e., to the same memory address)

    p1==p2



- A comparison betwe ue if the
  content of the two po ough
  the pointers refer to o ory
  locations

    *p1==*p2

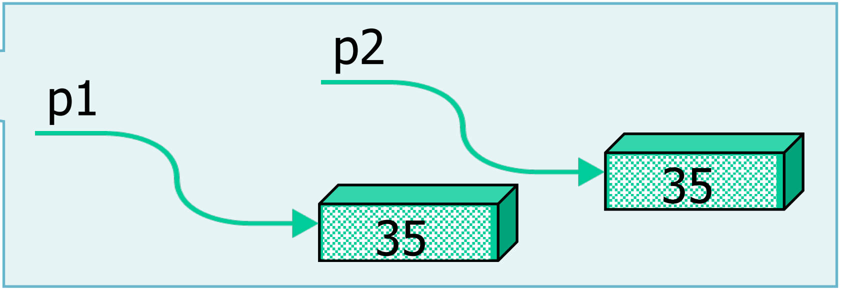
Relational operators == and !=

- A comparison between two pointers returns a true value if the pointers refer to the same datum (i.e., to the same memory address)

  p1==p2

- A comparison between two pointed data returns a true value if the content of the two pointed variables are the same, even though the pointers refer to data that are stored in different memory locations

  *p1==*p2



Pointers arithmetic

- A variable pointer contains a memory address, that is an integer
- The following arithmetic operations are possible:
  - Increment and decrement (by 1):

    `int *px;` ➡ `px++; px--;` are valid operations
  - Sum and subtraction by an integer value: +i -i

    `int *px` ➡ `px+=3; px-=5;` are valid operations
  - Subtraction (not sum!!!) of two pointers **of the same type**

    `int *px, *py;` ➡ `px-py` is a valid operation

    `px+py` is **NOT** a valid operation

Pointers arithmetic

- The operations on pointers do not follow the rules of integer arithmetic, but **depend on the pointed type**
- Given the instruction `p=p+i;` or `p++;` the actual increment is not `i` or `1`, but:
  - for `p=p+i` the increment is `i*(sizeof(*p))`
  - for `p++` the increment is `sizeof(*p)`

  `i` e `1` do not represent adjacent addresses, but data of the pointed type.
- Example

```
int *px, i=3; /* assume px = 1000 */
px += i; /* px will not be 1000 but 1000 + 3*sizeof(int)*/
```

Pointers arithmetic

How many bytes does a datum occupy? **sizeof()**

- The operations on pointers do not follo
  but depend on the pointed type
- Given the instruction `p=p+i;` or `p++;` the actual increment is not `i` or `1`, but:
  - ○ for `p=p+i` the increment is `i*(sizeof(*p))`
  - ○ for `p++` the increment is `sizeof(*p)`

  `i` e `1` do not represent adjacent addresses, but data of the pointed type.
- Example

```
int *px, i=3; /* assume px = 1000 */
px += i; /* px will not be 1000 but 1000 + 3*sizeof(int)*/
```

Pointers arithmetic

- Subtraction of pointers:
  - The result of a subtraction is the number of elements (of the specific pointed type) that are within the two given pointers (i.e., addresses)
  - Valid only with operands (pointers) of the same type!
  - Example:

```
int *px, *py, diff;
/* assume py=1000,  px = 1012*/
diff = px - py;
```

`diff` is not 12, but `(1012-1000)/sizeof(int)`

Example

```
int a[3]={1,9,2}, *p_a=&a[0];
char b[5]={'a','e','i','o','u'}, *p_b=&b[0];
```

Example

```
int a[3]={1,9,2}, *p_a=&a[0];
char b[5]={'a','e','i','o','u'}, *p_b=&b[0];
```

Contextual declaration and initialization of an array of int and an array of char

Example

```
int a[3]={1,9,2}, *p_a=&a[0];
char b[5]={'a','e','i','o','u'}, *p_b=&b[0];
```

Declaration and initialization of 2 pointers to the first cell of the array

```
int a[3]={1,9,2}, *p_a=&a[0];
char b[5]={'a','e','i','o','u'}, *p_b=&b[0];
```

The instructions:

```
printf("a[0]=*p_a=%d,p_a=%p\n",a[0],p_a);
printf("a[1]=*(p_a+1)=%d,p_a+1=%p\n",a[1],p_a+1);
printf("b[0]=*p_b =%c,p_b=%p\n",b[0],p_b);
printf("b[3]=*(p_b+3)=%c,p_b+3=%p\n",b[3],p_b+3);
```

Will print on the screen:

```
a[0]=*p_a=1,p_a=0028FEF8
a[1]=*(p_a+1)=9,p_a+1=0028FEFC
b[0]=*p_b=a,p_b=0028FEF3
b[3]=*(p_b+3)=o,p_b+3=0028FEF6
```

- Incrementing (or decrementing) by 1 a pointer is the same as computing the pointer to the next (or preceding) adjacent datum of the same type in memory

Example:

```
int x[100], *p = &x[50], *q, *r;
...
q = p+1; /* is the same as q=&x[51] */
r = p-1; /* is the same as r=&x[49] */
q++;     /* now q points to x[52] */
```

- Adding (or subtracting) an integer value i to a pointer is the same as incrementing or decrementing by 1 the pointer i times

Example:

```
int x[100], *p = &x[50], *q, *r;
...
q = p+10; /* is the same as q=&x[60] */
r = p-10; /* is the same as r=&x[40] */
r -= 5;   /* now r points at x[35] */
```

Parameters passing

- In C language parameters are passed to functions only “by value”
  - The value of the **actual** parameter is **copied** into the **formal** parameter when the function is called
- In theory there is no “by reference” parameters passing
- In the practice, by reference passing is obtained as follows:
  - A pointer to a datum is passed by value to the function (“by pointer” passing)
  - The function **uses the pointer to access to the same datum of the caller**

Example: swap of 2 integers (WRONG!!!)

Wrong attempt of implementing a function to swap the content of two variables

```
void swapInt (int x, int y) {
    int tmp =x;
    x=y; y=tmp;
}
...
void main (void) {
   int a, b;
   ...
   swapInt(a,b);
   ...
}
```

Example: swap of 2 integers (WRONG!!!)

Wrong attempt of implementing a function to swap the content of two variables

```
void swapInt (int x, int y) {
    int tmp =x;
    x=y; y=tmp;
}
...
void main (void) {
   int a, b;
   ...
   swapInt(a,b);
   ...
}
```

The swap happens between the local variables of the function, but not in the main!

Example: swap of 2 integers (CORRECT)

Function that swaps the content of two variables, using pointers

```
void swapInt (int *px, int *py) {
    int tmp = *px;
    *px=*py; *py=tmp;
}
...
void main (void) {
   int a, b;
   ...
   swapInt(&a,&b);
   ...
}
```

Example: swap of 2 integers (CORRECT)

Function that swaps the content of two variables, using pointers

```
void swapInt (int *px, int *py) {
     int tmp = *px;
     *px=*py; *py=tmp;
}
...
void main (void) {
    int a, b;
    ...
    swapInt(&a,&b);
    ...
}
```

The main passes pointers to a and b to the function

Example: swap of 2 integers (CORRECT)

Function that swaps the content of two variables, using pointers

```
void swapInt (int *px, int *py) {
    int tmp = *px;
    *px=*py; *py=tmp;
}
...
void main (void) {
   int a, b;
   ...
   swapInt(&a,&b);
   ...
}
```

The function swaps the content of the DATA POINTED by the given pointers.