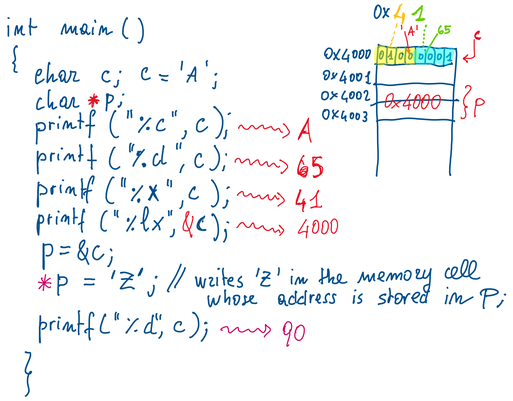
2.1 C-Language Interface to the machine

martedì 27 febbraio 2024 14:39



Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

In C, pointers are variables that store the memory addresses of other variables. Operations involving pointers are crucial for tasks like dynamic memory allocation, array manipulation, and implementing complex data structures like linked lists and trees. Below is a detailed description of the operators that act on pointers in C, their meanings, and examples for each.

1. Address-of Operator (&)

- Meaning: Returns the memory address of its operand.
- Example:

```
intvar = 5;
int*ptr = &var; // ptr now holds the address of var
```

2. **Dereference Operator (*)**

- Meaning: Accesses the value at the address stored in a pointer variable.
- Example:

```
intvar = 5;
int*ptr = &var; intvalue = *ptr; // value is now 5, the value pointed to by ptr
```

3. Pointer Arithmetic

Pointer arithmetic allows pointers to be manipulated in ways that make it easy to access array elements or traverse memory. The size of the type to which the pointer points affects the calculation.

- Increment (++): Moves the pointer to the next memory location of the type it points to.

```
intarr[] = {10, 20, 30}; int*ptr = arr; ptr++; // Now points to arr[1]
```

- Decrement (--): Moves the pointer to the previous memory location of the type it points to.

```
int*ptr = &arr[2]; ptr--; // Now points to arr[1]
```

- Addition (+): Adds an integer value to a pointer.

```
int*ptr = arr; ptr = ptr + 2; // Now points to arr[2]
```

- Subtraction (-): Subtracts an integer value from a pointer or calculates the difference between two pointers.

```
int*ptr = &arr[2]; ptr = ptr - 1; // Now points to arr[1]// Difference between two pointersintdiff = ptr - arr; // Equals 1
```

4. Relational Operators

Pointers can be compared using relational operators such as ==, !=, <, >, <=, and >=.

- Equality (==): Checks if two pointers point to the same memory location.

```
if(ptr1 == ptr2) { // Pointers point to the same address)
```

- Inequality (!=): Checks if two pointers point to different memory locations.

```
if(ptr1 != ptr2) { // Pointers point to different addresses)
```

- Greater than (>), Less than (<), Greater than or equal to (>=), Less than or equal to (<=): Compare the numerical values of two pointer addresses.

```
if(ptr1 < ptr2) { // ptr1 points to a memory address lower than ptr2)
```

5. Assignment Operators

- Simple Assignment (=): Assigns a value to a pointer.

```
intvar = 10; int*ptr = &var; // Assigns the address of var to ptr
```

- Compound Assignment (e.g., +=, -=, *=): These operators are not commonly used with pointers, but technically, += and -= can be used to adjust the address a pointer holds, in a manner similar to pointer arithmetic.

```
ptr += 1; // Equivalent to ptr = ptr + 1
```

When using pointer operators, it's crucial to understand the type of data the pointer is addressing, as this affects how arithmetic and dereferencing operations work. Misusing pointer operations can lead to undefined behavior, including crashes and security vulnerabilities.

The [] operator in C is used for array subscripting. It provides a more intuitive way to access elements of an array, but under the hood, it's essentially syntactic sugar for pointer arithmetic. Here's how it relates to pointers and how it compares with using pointer offsets directly.
The [] Operator

- Meaning: Accesses an element of an array.
- Example:

```
intarr[3] = {10, 20, 30}; intvalue = arr[1]; // Accesses the second element of arr, value is now 20
```

Array Subscripting with Pointers

The expression arr[i] is equivalent to *(arr + i). This shows the close relationship between arrays and pointers in C. Here's how you can achieve the same with pointers:

- Pointer Equivalent:

```
int*ptr = arr;
intvalue = *(ptr + 1); // Also accesses the second element of arr, value is now 20
```

This demonstrates that the [] operator is essentially performing pointer arithmetic to access the desired array element.
Comparison Between Arrays and Pointer Offset
When you use an array name in an expression, it is (in most cases) converted to ("decays into") a pointer to its first element. This means that the array name acts very much like a pointer constant, pointing to the first element of the array.

- Direct Array Access vs. Pointer Offset:
  - Using arr[i] directly accesses the ith element of the array arr.
  - Using *(arr + i) or *(ptr + i) (where ptr is a pointer to the first element of arr) accesses the ith element by moving i positions from the start, leveraging pointer arithmetic.

Advantages and Disadvantages

- Readability: The [] operator makes code easier to read and understand, especially for those familiar with arrays but not the intricacies of pointers

- Flexibility: Using pointer arithmetic (*(ptr + i)) might offer more flexibility in some scenarios, especially when dealing with dynamic memory allocation or manipulating parts of an array based on conditions that change at runtime.

  Performance: There is no performance difference between using [] and explicit pointer arithmetic, as the compiler translates [] into the equivalent pointer arithmetic operation.