Programmazione e Calcolo Scientifico

Matteo Cicuttin

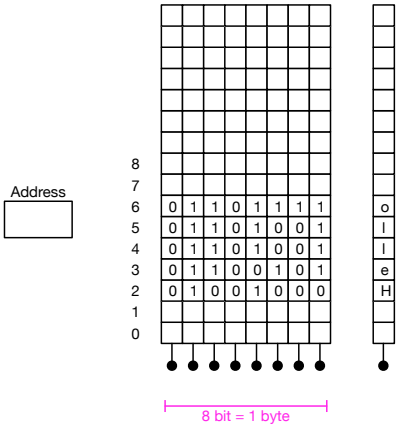
Politecnico di Torino

March 19, 2024

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.

(Bjarne Stroustrup)

Memory from the software point of view

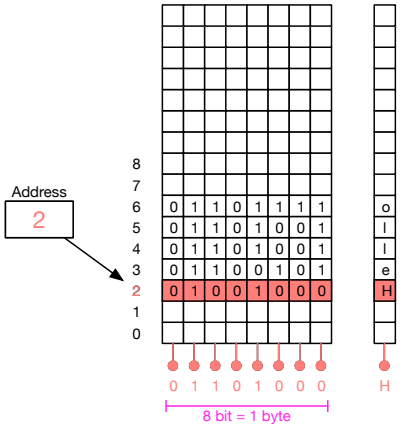


Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Memory from the software point of view

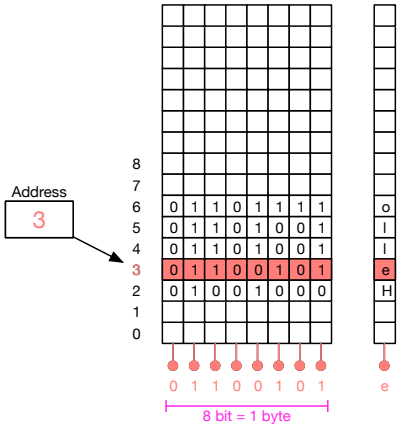


Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Memory from the software point of view

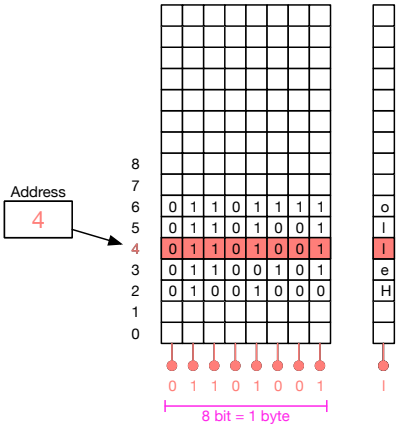


Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Memory from the software point of view

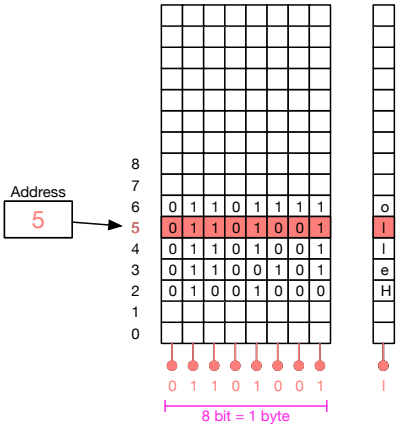


Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Memory from the software point of view

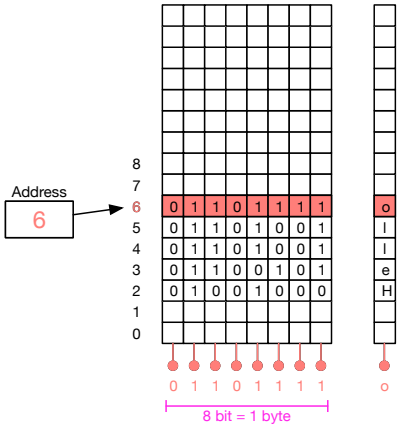


Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Memory from the software point of view



Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Arrays

- We already discussed `std::string`: forget about it for today.
- We discussed `char` variables to hold single characters, for example

```
char c = 'H';
```

How to store our complete string `'Hello'`? More generally, how to store conceptually contiguous stuff?

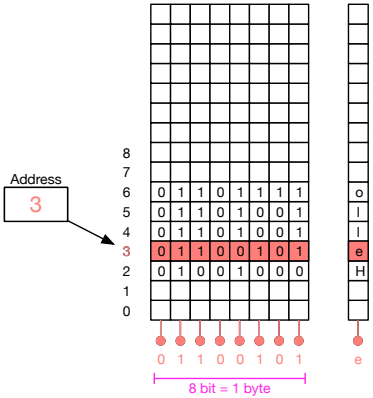
```
char c0 = 'H';
char c1 = 'e';
char c2 = 'l';
// and so on...
```

Not a really smart strategy, right?

In C and C++, when we need to store a **fixed** amount of data, we can use **arrays**.

- `char cs[5] = { 'H', 'e', 'l', 'l', 'o' };`
- `int is[4] = { 42, 7, 9, -1 };`
- `double ds[3] = { 2.754, 1e-7, -2.2e9 };`

Array memory layout - char



We said that the address points to the byte we want to read.

```
char cs[5] = { 'H', 'e', 'l', 'l', 'o' };
```

The above array maps perfectly to our memory because `sizeof(char)` $= 1$ byte.

- `cs[0] == 'H'` and lives at address 2
- `cs[1] == 'e'` and lives at address 3
- ...and so on...

Array memory layout - int

The size of an integer is 4 bytes. Said otherwise, `sizeof(int)` $= 4$.

How do we lay out the following array?

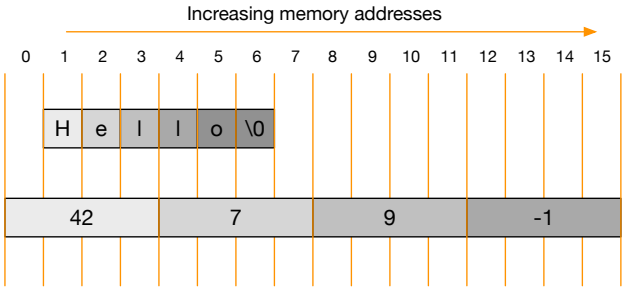
```
int is[4] = { 42, 7, 9, -1 };
```

Array memory layout - int

The size of an integer is 4 bytes. Said otherwise, `sizeof(int)` $= 4$.

How do we lay out the following array?

```
int is[4] = { 42, 7, 9, -1 };
```



- `is` starts at 0, so `is[0] == 42` and lives at address 0
- `is[1] == 7` and lives at address 4
- ...and so on...

Array memory layout - summary

 Warning! 

Let's `T` be some type (`int`, `double`, whatever):

- `T arr[N];` declares an array. `N` must be a compile-time constant. It means fixed-size.
- Array indices start at 0.
- `arr[n];` accesses the `n`-th element of the array under the condition `n >= 0 && n < N`.
  If `!(n >= 0 && n < N)` the hell will break loose and the world will end.
  We will discuss how to protect array accesses in the next lessons.
- The `i`-th element lives at address `base + i*sizeof(T)`, where `base` is the array start address.

If you understand this, you will have no problems at all with pointers.

DON'T WAIT TO ASK QUESTIONS! DO IT NOW!

Null-terminated strings

Warning!

Strings must always be null-terminated. Therefore,

```
char cs[5] = { 'H', 'e', 'l', 'l', 'o' };
```

is an array of 5 chars but not a well-formed C string. A well formed C string is

```
char cs[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

or, with better notation

```
char cs[6] = "Hello";
```

where the terminating zero is implicit. The zero termination means "end of string".

→ Use `std::string` to handle strings ←

Null-terminated strings

An observation on null-terminated strings

The character "0" is different from "\0"!

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

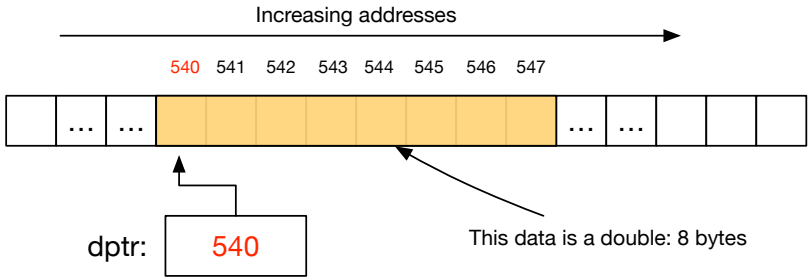
Source: www.LookupTables.com

Pointers

→ A pointer is a variable that contains an address ←

Let's `T` be some type (`int`, `double`, whatever). A **pointer** is declared by adding a star to the type:

```
T*      ptr;        // a pointer to an object of type T
int*    iptr;       // a pointer to an integer
double* dptr = 540; // a pointer to a double
```



Pointers - exercise

A little exercise on pointers. Assumptions:

- A new operator: the unary &. It gives you the address where something is stored.
- `sizeof(int)` = 4 and `sizeof(double)` = 8.
- Let's say `iarr` starts at 10000 and `darr` starts at 9000.

Tell me the values of `ip` and `dp`:

```
int iarr[10];
int* ip = &iarr[4];
double darr[7];
double* dp = &darr[3];
```

Pointers - exercise

A little exercise on pointers. Assumptions:

- A new operator: the unary &. It gives you the address where something is stored.
- `sizeof(int)` = 4 and `sizeof(double)` = 8.
- Let's say `iarr` starts at 10000 and `darr` starts at 9000.

Tell me the values of `ip` and `dp`:

```
int iarr[10];
int* ip = &iarr[4];
double darr[7];
double* dp = &darr[3];
```

- `ip = 10000 + 4*sizeof(int) = 10000 + 4*4 = 10016`
- `dp = 9000 + 3*sizeof(double) = 9000 + 3*8 = 9024`

Dereferencing pointers

If a pointer is an address of some value, how do we use the pointed value?

A new operator: the dereference operator *.

```
double darr[5] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

double* pd3;          // pointer to double
pd3 = &darr[3];       // we assign to pd3 the address of the 4th element
double d3 = *pd3;     // we get that element and we copy it to d3
```

The * in front of a pointer variable *dereferences* it: it returns the value pointed by the pointer.

What does `cout << d3 << "\n";` print?

Pointer syntax

C++ is derived from C, and C was devised in the '70s. **Syntax has many pitfalls**.
To declare two integer variables `var1` and `var2` you have two ways:

```
int var1; // one line, one variable
int var2;
```

```
int var1, var2; // two variables in one go
```

- I told you that to declare a pointer to `int` you add a `*` in the type, as in `int*`. However, the dereference operator has what is called *right-to-left associativity*. So, if you write

```
int* var1, var2;
```

`var1` is a **pointer to int**, whereas `var2` is an `int`. To get two pointers to int:

```
int *var1, *var2;
```

Pointer syntax

 Warning! 

C++ is derived from C, and C was devised in the '70s. **Syntax has many pitfalls**.
To declare two integer variables `var1` and `var2` you have two ways:

```
int var1; // one line, one variable
int var2;
```

```
int var1, var2; // two variables in one go
```

- I told you that to declare a pointer to `int` you add a `*` in the type, as in `int*`. However, the dereference operator has what is called *right-to-left associativity*. So, if you write

```
int* var1, var2;
```

`var1` is a **pointer to int**, whereas `var2` is an `int`. To get two pointers to int:

```
int *var1, *var2;
```

Pointers and dynamic memory allocation

Please, please be sure to perfectly understand pointers.

Today we discussed arrays of **fixed size**: they stay of the same size during the whole program execution.

Real programs need variable size data structures, and for this dynamic memory allocation comes into play.

Pointers and dynamic memory are closely related and soon we will use plenty of pointers.

Control structures

Until now I talked about control structures only by analogy with your previous programming knowledge.

Now we introduce "formally" the main control structures of C++

- Conditional selection: `if`/`else`
- Selection by cases: `switch`
- Unbounded iteration: `while`
- ~~Bounded~~ C++ does not have bounded iteration: `for`
- `break`/`continue`: stop iteration prematurely or skip iteration

Control structures: if

General syntax:

```
if ( <condition> ) {
    // true branch
}
else {
    // false branch
}
```

Example:

```
int max(int a, int b) {
    if ( a < b ) {
        return b;
    }
    else {
        return a;
    }
}
```

Better:

```
int max(int a, int b) {
    if ( a < b ) {
        return b;
    }

    return a;
}
```

Control structures: switch

The `switch` statement allows to avoid chaining multiple if/else when you have to select a value

```
switch ( <value> ) {
    case <value1>:
        // code for value1
        break;

    case <value2>:
    case <value3>:
        // code for value3 and value4
        break;

    default:
        // otherwise...
        break;
}
```

```
int fib(int n) {
    switch (n) {
        case 0:
            return 1;
            break;

        case 1:
            return 1;
            break;

        default:
            return fib(n-1) + fib(n-2);
    }
}
```

Control structures: while and do/while

The `while` loop iterates *while* a condition is true. It has two variants.

```
while ( <condition> ) {
    // loop statements
}
```

```
do {
    // loop statements
} while ( <condition> );
```

- The variant on the left checks the condition before entering the first time
- The variant on the right does at least a cycle before checking the condition
- Variant with `do/while` is not so common...

```
int fact(int n) {
    int ret = 1;
    while (n) {
        ret *= n--; //same as ret = ret*n--;
    }
    return ret;
}
```

Control structures: for

```
for ( <initialization>; <condition>; <increment> ) {
    // loop statements
}
```

The `for` statement consists of three parts:

- `<initialization>`: initialize the iteration variable
- `<condition>`: check if we still need to iterate
- `<increment>`: increment the iteration variable

```
for (int i = 1; i <= 1024; i *= 2) {
    std::cout << i << "\n"; // prints some powers of 2
}
```

All three parts are optional: `for(;;)` gives you an infinite loop.

Control structures: break/continue

The keywords `break` and `continue` alter the execution of a loop.

- `break` stops and exits the loop when encountered
- `continue` goes to the next iteration

```
for(;;) {
    // statements
    if (something)
        break; // exit the loop
    //statements
}
```

```
int i = 100;
while (i--) {
    if (i%2 == 0)
        continue;
    std::cout << i << "\n";
}
```

Structures

Sometimes it is needed to group together related data. For this we use `structs`. For example:

```
struct meteo_data {
    std::string location_name;
    int         minute, hour, day, month, year;
    double      temperature;
    double      pressure;
};
```

Subsequently, to create an object of type `meteo_data`:

```
meteo_data md;
md.location_name = "Torino";
md.minute = 42;
md.hour = 21;
md.temperature = 16.3;
// and so on...
```