CSCI 262
Data Structures

5 – Pointers and Memory

Prelude

NUMBERS

Positional Notation

Also called *place-value* notation

- Each place represents a power of the *base*
- Each numeral is multiplied by positional value

E.g., base 10 (decimal):

$(4273)_{10} = 3 \times 10^0 + 7 \times 10^1 + 2 \times 10^2 + 4 \times 10^3$

Other Bases

Computer scientists tend to think in powers of 2:

- Hexadecimal (base-16) – use digits 0-9, a-f (or A-F)
  $(4273)_{10} = (10b1)_{16} = 1 \times 16^0 + 11 \times 16^1 + 1 \times 16^3$
- Octal (base-8) – mostly out of use now
  $(4273)_{10} = (10261)_8 = 1 \times 8^0 + 6 \times 8^1 + 2 \times 8^2 + 1 \times 8^4$
- Binary! (0s and 1s)
  $(4273)_{10} = (0001\ 0000\ 1011\ 0001)_2$

Bits and Bytes

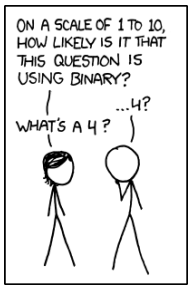
Computers work with *bits* – 0's and 1's

- (Positive) integers are represented in base 2:
  $0_{10} = 0_2$,
  $1_{10} = 1_2$,
  $2_{10} = 10_2$,
  $3_{10} = 11_2$,
  $4_{10} = 100_2$,
  $5_{10} = 101_2$, etc.

The maximum (unsigned) integer we can store in n bits is 2^n-1.

- Computers organize bits into *bytes* – 8-bit chunks
- C++ data types are organized into bytes
  - `char` uses 1 byte
  - `int` uses 4 bytes
  - `double` uses 8 bytes
- Get size of a variable/object type with sizeof:

```
int sz_of_dbl = sizeof(double);
```

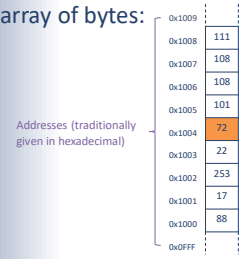



http://xkcd.com/953/

POINTERS AND MEMORY

Memory

Computer memory is organized as an *indexed* array of bytes:

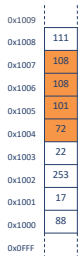


We say that the byte value 72 is stored at address 0x1004…

It is traditional to represent memory as a vertical array.

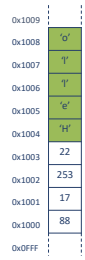
All right thinking people start at the bottom and count up ☺

Memory



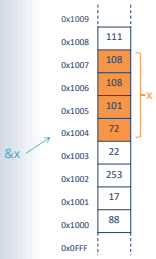
However, from the programmer's perspective, the value stored at 0x1004 depends on the type. It could be an *int* value (4 bytes)…

Memory



Or the start of a string…

Reference (address of) Operator &



Suppose this `int` value corresponds to the variable `x`:

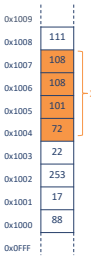
```
…
int x = 1819043144;
…
```

We can obtain the address of `x` using the operator `&`:

```
…
cout << &x << endl; // prints 0x1004
…
```

(Try it!)

Pointers



A *pointer* is a variable that stores an address:

```
…
int x = 1819043144;
int* p = &x;  // p now stores 0x1004
```

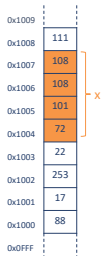
The *type* of the variable `p` is `int*`.
Note that `int*` is only for pointers to `int`; every type T has a corresponding pointer type T*.

You can write

```
int* p;
int *p;
int * p;
```

- the compiler interprets them all the same.

Dereference Operator *



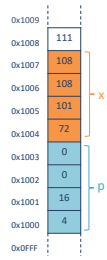
You usually don't want to see the address itself, but what is at the address – you can get the pointed-to value by using *:

```
…
int x = 1819043144;
int* p = &x;
cout << *p << endl;
```

The previous line outputs the same thing as:

```
cout << x << endl;
```


So Where Do Pointers Live…?



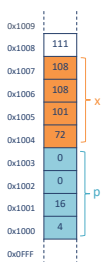
In memory, of course!

```
…
int x = 1819043144;
int* p = &x;
```

And yes, you can declare pointers to pointers, ad infinitum…

```
int** pp = &p;
int*** q = &pp;
```

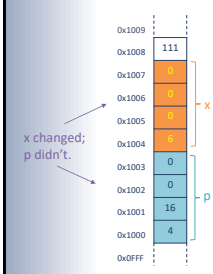

Pointer Independence



Suppose we change the value of x:

```
…
int x = 1819043144;
int* p = &x;
```

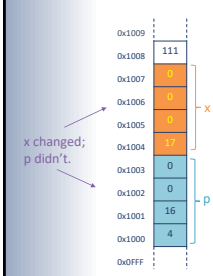

Pointer Independence



Suppose we change the value of x:

```
…
int x = 1819043144;
int* p = &x;
x = 6;
```


Assigning Through *



Suppose we change the value of x:

```
…
int x = 1819043144;
int* p = &x;
x = 6;
```

We can also assign through the * operator:

```
*p = 17;
```


Pointers As Variables

Pointers can be assigned like any other variable:

```
int x, y;
int* p = &x;  // p points to x
int* q = p;   // now q and p point to x
p = &y;       // now p points to y, q to x
*q = 15;      // x now stores 15
```

The `nullptr` Pointer

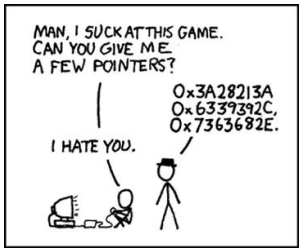
- C++ defines a special keyword for pointers which do not currently point to anything: `nullptr`

```
int* p = nullptr;
```

- A null pointer is *never* a valid memory address:

```
int* p = nullptr;
cout << *p << endl;   // crash
*p = 42;              // also crash
```

- Prior to C++ 11, the value NULL was used instead of nullptr. You will see a lot of code using NULL.



http://xkcd.com/138/

POINTER NOTES

Dereferencing Pointers

- Given a pointer p to some value:
  *p dereferences p, is equivalent to the value.
- Suppose p points to an object or structure:
  `(*p).foo` dereferences p and accesses the member foo
  `p->foo` does the same thing
- In the next lecture we'll see that array indexing is another kind of dereferencing:
  `p[i] == *(p + i)`
  (But we'll have to explain pointer arithmetic first.)

Multiple Pointer Declaration

An oddity of C/C++: we must do

```
int *p, *q;   // we have to use * for both
```

even though `int*` is the type.

Otherwise:

```
int *p, q;   // p is an int*, but q is an int
```


Up Next

Please read Chapter 7 in your textbook!

- Friday, Sept. 14
  - Lab 4 – Memory
  - Project 1 due
  - Next assignment: TBD
- Monday, Sept. 17
  - Pointers & Arrays
  - Dynamic Memory Allocation