Numbering Systems

We are all familiar with decimal, or base 10:

\[ 79_{10} = 9 \times 10^0 + 7 \times 10^1 \]

Computer scientists tend to think in powers of 2...

So we get:

- binary (base 2):
  \[ 1001111_2 = 1 \times 2^6 + 1 \times 2^1 + 1 \times 2^2 + ... = 79_{10} \]

- octal (base 8):
  \[ 117_8 = 7 \times 8^0 + 1 \times 8^1 + 1 \times 8^2 = 79_{10} \]

- hexadecimal (base 16):
  \[ 4F_{16} = 15 \times 16^0 + 4 \times 16^1 = 79_{10} \]

Bits and Bytes

Computers work with bits – 0’s and 1’s

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

- Computers organize bits into bytes – 8-bit chunks

- All C++ data types are organized into bytes
  - chars use 1 byte
  - ints use 4 bytes
  - doubles use 8 bytes

Memory

Computer memory is organized as an indexed array of bytes:

We say that the byte value 72 is stored at address 0x04...

However, from the programmer’s perspective, the value stored at 0x04 depends on the type. It could be an int value...
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 “0x04” (Try it!)
```

Pointers

A pointer is a variable that stores an address:
```
int x = 1819043144;
int* p = &x; // p now stores 0x04
```
Note that int* is only for pointers to int. every type T
has a corresponding pointer type T*.

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 would output 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;
```

Playing with Pointers

Suppose we change the value of x:
```
int x = 1819043144;
int* p = &x;
x = 6;
```

Playing with Pointers

Suppose we change the value of x:
```
int x = 1819043144;
int* p = &x;
```

p
Playing with Pointers

Suppose we change the value of x:
```c
int x = 0x6c6c6548;
int * p = &x;
x = 6;
```
We can also assign through the * operator:
```c
*p = 17;
```

Working with Pointers

When declaring multiple pointers:
```c
int *p, *q; // we have to use * for both
```
Pointers can be assigned like any other variable:
```c
p = &x; // p points to x
q = p; // now q points to x
p = &y; // now p points to y
*q = 15; // x now stores 15
```

The NULL Pointer

- C++ defines a special value for pointers which do not currently point to anything: NULL
  ```c
  int* p = NULL;
  ```
- A NULL pointer is never a valid memory address:
  ```c
  int* p = NULL;
  cout << *p << endl; // crash
  *p = 42; // also crash
  ```
- Typically NULL == 0 (memory address 0x0)

Arrays

Arrays are just sequential chunks of memory:

```
char s[5] = {'H','e','l','l','o'};
char *p = s;
```

```
cout << *p << endl; // prints 'H'
cout << *s << endl; // also prints 'H'
```

Incrementing Pointers

Here's a neat trick:
```
for (p = s; p < s + 5; p++)
  cout << *p << endl;
```
```
What does this print out?
  cout << *p << endl;
```
```
How about this?
for (p = s; p < s + 5; p++)
  cout << *p << endl;
```
```
```
**Pointer Arithmetic**

```c
int arr[] = {0, 1, 2, 3, 4};
int* p = arr;
p++;  // p increases by 4, not 1
p = arr + 3;  // p = address of start of
              // arr plus 12 bytes!
p -= 2;
cout << *p << endl;
```

Take home: pointer arithmetic depends on type of pointer.

---

**C-style Strings**

In C, strings are simply arrays of char:
```c
char *s = "Hello!";
```

This array has size 7, not 6: the last entry stores value 0, or \\

```
\textbackslash 0
```

Without this value, no way to detect the end of a string!
With it, though, we can do:
```c
for (char* p = s; *p != '\0'; p++) { ... }
```

String literals in C++ are still represented this way, but convert easily to
the string type:
```c
string foo = "Hello";
```

or
```c
string("Hello")
```

---

**Pointers and Reference Parameters**

Reference parameters are not pointers!
Reference parameters are not pointers!
Reference parameters are not pointers!

If you have a function
```c
void foo(int &x) { ... }
```
Inside `foo`, you cannot do
```c
*x = 10;  // incorrect!
```
You just do
```c
x = 10;  // correct
```

Sources of confusion:
- `&` denotes a reference parameter
- `&` also used as address-of operator
- References use pointers “under the covers”