CSE 220: Systems Programming

Pointers and Data Representation

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

More on `void` Pointers

Void pointers are powerful for raw memory manipulation.

You can use them to put arbitrary values into memory.

You will use this in PA3 and PA4!

We will look at using `void *` to:

- Pass a pointer of an arbitrary type
- Read and write arbitrary types in memory

Floating Point

Along the way, we will detour into floating point representation.

Floating point is the counterpoint to integer representation.

It is used to:

- represent rational numbers
- approximate real numbers

Binary floating point formats have some surprising properties.

Dumping Memory

```c
#include <stdio.h>

void dump_mem(const void *mem, size_t len) {
    const char *buffer = mem;    // Cast to char *
    size_t i;

    for (i = 0; i < len; i++) {
        if (i > 0 && i % 8 == 0) { printf("\n"); }

        printf("%02x ", buffer[i] & 0xff);
    }
    if (i > 1 && i % 8 != 1) { puts(""); }
}
```

dump_mem Details

What is this for?

```
const char *buffer = mem;
```

dump_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler "we're going to use `mem` as an array of bytes".

dump_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler "we're going to use `mem` as an array of bytes".

What about this:

```
if (i > 0 && i % 8 == 0){ printf("\n"); }
```

dump_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use `mem` as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ printf("\n"); }
```

It prints a newline after every 8th byte excepting the first.

dump_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use `mem` as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ printf("\n"); }
```

It prints a newline after every 8th byte excepting the first.

Finally:

```
buffer[i] & 0xff
```

dump_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use `mem` as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ printf("\n"); }
```

It prints a newline after every 8th byte excepting the first.

Finally:

```
buffer[i] & 0xff
```

This is necessary to avoid sign extension.

Inconvenient Representation

Pointers to `void *` can be used to store and interpret representations that are inconveniently represented in C.

Consider the following structure:

```
struct Inconvenient {
    int fourbytes;
    long eightbytes;
} inconvenient;
```

This structure contains 12 bytes of data, but occupies 16 bytes. (Because of padding…)

To communicate this structure we wish to send only 12 bytes.

Serialization

Communicating such data is often done via serialization.

Serialization is the storage of data into a byte sequence.

In C, we do this with pointers, and often `void` pointers.

Consider:

```
void *p = malloc(12);
*(int *)p = inconvenient.fourbytes;
*(long *)(p + sizeof(int)) = inconvenient.eightbytes;
```

This builds a 12-byte structure without padding.
(In the process, it violates alignment restrictions.)

Flexible Sizes

Another use for `void` pointer representation is flexible sizes.

Consider a structure (not legal C):

```
struct Variable {
    size_t nentries;
    int entries[nentries];
    char name[]; /* name is NUL-terminated */
} variable;
```

This structure does not have a well-defined size.

Its size depends on `nentries` and the length of `name`!

Packing the Data

We can serialize this data as follows:

```
size_t nentries = 3;
int entries[] = { 42, 31337, 0x1701D };
const char *name = "Caleb Widowgast";

void *buf = malloc(sizeof(size_t)
                   + nentries * sizeof(int)
                   + strlen(name) + 1);
void *cur = buf;
```

Packing the Data

We can serialize this data as follows:

```
*(size_t *)cur = nentries;
cur += sizeof(size_t);
for (int i = 0; i < nentries; i++) {
    *(int *)cur = entries[i];
    cur += sizeof(int);
}

for (int i = 0; i <= strlen(name); i++) {
    *(char *)cur++ = name[i];
}
```

Packing the Data

We can serialize this data as follows:

```
size_t nentries = 3;
int entries[] = { 42, 31337, 0x1701D };
const char *name = "Caleb Widowgast";
```

```
03 00 00 00 00 00 00 00
2a 00 00 00 69 7a 00 00
1d 70 01 00 43 61 6c 65
62 20 57 69 64 6f 77 67
61 73 74 00
```

Using dump_mem()

We have previously used dump_mem() to analyze integers.

We will now use it to look at floating point.

Dumping a float looks like this:

```
float f = 1.0;
dump_mem(&f, sizeof(float));
```

Note that &f is of type float *, but can be passed to void *.

What is "Floating Point"?

A floating point number, such as a `float` or `double`, is a number with a variable number of digits before or after the decimal point

(On computers, a variable number of bits before or after the binary point!)

Examples:
3.14159
6.022×10^{23}
6.626×10^{-34}

What is “Floating Point”?

A floating point number, such as a `float` or `double`, is a number with a variable number of digits before or after the decimal point

(On computers, a variable number of bits before or after the binary point!)

Examples:
3.14159
6.022×10^{23}
6.626×10^{-34}

It would take nearly 200 bits to represent all three of these numbers precisely.

What is “Floating Point”?

In order to represent numbers of very small or very large magnitude, floating point allows the point to move.

The number of digits of precision is fixed.

Some (loose) terms:

- **Significand**: The meaningful digits of a number
- **Exponent**: The “distance” of those digits from zero in powers of the arithmetic base

Floating Point Representation

In base 10, a floating point number is of the form $x \times 10^y$.

If we consider Avogadro's Number (6.022×10^{23}):

- The significand x is 6.022
- The exponent y is 23.

This requires six digits to store, versus 24 digits for 602200000000000000000000.

In base 2, a floating point number is $x \times 2^y$.

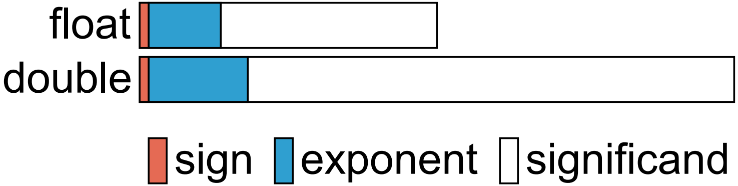
IEEE 754 Floating Point

IEEE Standard 754 defines a particular floating point format.

If a floating point number is $x \times 2^y$, in IEEE 754:

- A single precision number (`float`) has a 23-bit x and 8-bit y
- A double precision number (`double`) is 52-bit x and 11-bit y

Each has a one-bit sign.



Storing IEEE 754 Components

However, x and y are not stored directly!

x (the significand) is stored:

- Normalized to a value right of the binary point
- With an assumed leading 1 preceding the binary point

This means that a stored significand of 0 is $x = 1.0$

y (the exponent) is stored as $y + 127$.

This means that an exponent of 0 is stored as 127.

Examining Floats

```
float f1 = 2.0f;
float f2 = 0.2f;

dump_mem(&f1, sizeof(f1));
dump_mem(&f2, sizeof(f2));
```

Examining Floats

```
float f1 = 2.0f;
float f2 = 0.2f;

dump_mem(&f1, sizeof(f1));
dump_mem(&f2, sizeof(f2));
```

Output:

```
00 00 00 40
cd cc 4c 3e
```

Deconstructing 2.0

Why is `2.0f` `0x40000000`?

0 10000000 0000000 00000000 00000000

Remembering our significand and exponent storage rules, this means:

$x = 1.0$ (x is stored as significant digits after the point: $1 + 0$)

$y = 1$ (y is stored plus 127: $128 - 127$)

Thus: $1.0 \times 2^1 = 2.0$

(We didn't use 1.0 because it's kind of a special case.)

Deconstructing 0.2

This became 0x3e4ccccd, or:
0 01111100 1001100 11001100 11001101

Is this surprising?

Deconstructing 0.2

This became 0x3e4ccccd, or:
0 01111100 1001100 11001100 11001101

Is this surprising?

What just happened?

Deconstructing 0.2

This became 0x3e4ccccd, or:
0 01111100 1001100 11001100 11001101

Is this surprising?

What just happened?

The significand isn't decimal!

It's after the binary point.

Fractions cleanly represented in decimal, like $^1/_5$, may not be clean in binary — sort of like $^1/_3$ in decimal.

The Binary Point

Suppose we have a b-bit binary number with bits both before and after the binary point, such that:

- There are w whole-number bits before the binary point
- There are f fractional bits after the binary point
- The largest bit before the point is b_{w-1}
- The smallest bit before the point is b_0
- The largest bit after the point is b_{-1}
- The smallest bit after the point is b_{-f}

A *w*.*b*-bit Binary Number

The *w* whole-number bits are defined as in integers:

$$b_i, i \geq 0 \doteq b_i \cdot 2^i$$

The *f* fractional-number bits are defined as follows:

$$b_i, i < 0 \doteq b_i \cdot 2^{-b_i}$$

Thus, its total value is:

$$\sum_{i=0}^{w-1} b_i \cdot 2^i + \sum_{j=1}^{f} b_i \cdot 2^{-j}$$

An Example Binary-Point Computation

Consider 11.101b:

$$
\begin{aligned}
11.101\text{b} &= 1 \cdot 2^{1} + 1 \cdot 2^{0} + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\
&= 2 + 1 + {}^{1}\!/_{2} + 0 + {}^{1}\!/_{8} \\
&= 3\,{}^{5}\!/_{8} \\
&= 3.625
\end{aligned}
$$

More Floating Point

IEEE 754 is more complicated than we covered here.
(You'll read more about it in the text.)

We have covered the big ideas, however.

Some important implications to consider:

- Very large (either positive or negative) floating point numbers become imprecise because of that $\times 2^y$ factor.
- Very small (close to zero) floating point numbers become imprecise for the same reason.
- Double precision numbers can still be quite large and precise!
- The possible floating point values are unevenly spaced.[1]

[1] See "Denormalized Values" in your text for a caveat.

Summary

- The `void *` type can be used for raw memory manipulation
- Casting `void *` to another type is convenient
- Math on `void *` is by byte
- Floating point numbers hold rational values in base 2
- Not all non-repeating decimal numbers are non-repeating in binary
- IEEE 754 floating point

Next Time …

- Function calls and automatic variables

References I

Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 2: 2.4 Intro, 2.4.1-2.4.3, 2.4.6, 2.5. Pearson, 2016.

[2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Chapter 5: Intro, 5.1-5.6. Prentice Hall, 1988.

License

To retrieve a copy of this material, or related materials, see `https://www.cse.buffalo.edu/~eblanton/`.