Data Representation and Storage

- Learning Objectives
  - Define the following terms (with respect to C):
    - Object
    - Declaration
    - Definition
    - Alias
    - Fundamental type
    - Derived type
  - Use pointer arithmetic correctly
  - Explain how C data types arranged in memory.
    - Sizes
    - Alignment
    - Endianness

Some definitions (in C)

- Object: A region of storage
  - Distinct objects never overlap
  - Objects may have multiple names …
- Aliases: Multiple names for the same object
  - Different pointers to the same object are called aliases of each other.
- Example:

```
int foo;
int *name1, *name2;

name1 = &foo;
name2 = &foo;
```

More Definitions

- Definition: Allocates an object and creates a name for it.
  - Examples:

```
int foo;
char bar;
float baz;
```

- Declaration: Alerting the compiler that there exists an object of some name/type, but does not necessarily allocate the space for it.
  - Example:

```
extern int errno;
int func(void);
```


Object Sizes

- Every object in C has a size.
- You can get the size of an object using `sizeof`
  - Examples:

```
printf("An integer has size %zu\n", sizeof(int));

int x;
printf("An integer has size %zu\n", sizeof(x));
```

- `sizeof` returns a value of type `size_t`
- The size of an object determines the values it can hold.
  - Example: A `char` is 8 bits – the maximum value it can contain is 255. Why?

Fundamental Types

- C has a set of built-in or fundamental types:
  - int, unsigned int – signed and unsigned integers (4 bytes)
  - long, unsigned long – signed and unsigned longs (8 bytes)
  - short, unsigned short – signed "short" integer (2 bytes)
  - char – character (1 byte)
  - float, double, long double – usually, 4 bytes, 8 bytes, and 16 bytes respectively

These are all sizes on x86-64; some of them differ for x86-32

Screen Capture

- Run the vars1 and vars2 programs – notice a couple of things:
  - We have some BIG hex numbers! (If you are not comfortable with hex, check out the short video that goes over binary and hex.)
  - When we run vars1 (which prints out the addresses of shorts), we see that the addresses are all even.
  - When we run vars2 (which prints out the addresses of ints), we see that the addresses are all multiples of 4.

Binary and Hex in One Slide

- Data in a computer are stored in bits (binary digits).
- A bit can be either 0 or 1.
- Larger data types (e.g., char, int, short, etc) can be expressed as a string of binary digits, where each bit is a different power of two:
  - 101011 = $2^5 + 2^3 + 2^1 + 2^0 = 32 + 8 + 2 + 1 = 43$
- This can be tedious with 64 bit numbers …
- Hexadecimal represents data in base 16
- 1 hex digit = 4 bits (4 bits can represent up to 16 values)
- An int (4 bytes) can be expressed with 8 hex digits:

0x00001A2B = $1 * 16^3 + A * 16^2 + 2 * 16^1 + B * 16^0$
= 4096 + 10 * 256 + 2 * 16 + 11
= 4096 + 2560 + 32 + 11 = 6699

Screen Capture

- We ran vars3 and vars4 and noticed that even though we tossed some char variables in the mix, the addresses of the shorts were still even and the addresses of the ints were still divisible by 4.
- Why is this???

Alignment

- C has strict rules about where variables can be placed.
- The alignment, A, of a type, T, is a size such that the address of every object of type T is a multiple of A.
  - Shorts will always have even addresses (A = 2)
  - Ints will always have addresses that are a multiple of 4.
  - Longs will have addresses that are a multiple of ?.
- Sizeof(T) is always a multiple of alignof(T)
- Alignment of a struct is the maximum alignment of all the components in the struct.

Structs

- Comparable to “records” in other languages. Similar to the data part of classes.
- Lets you group together a set of objects that you want associated with one another.
- Example declaration:

```
struct point {
    int x;
    int y;
    int z;
};
```

```
struct student{
        char name[22];
        unsigned int age;
        char house[15];
};
```

- Example definition:

```
struct point p;
```

```
struct student eddie;
```

Derived Types

- These are types that you (the programmer) build from the fundamental types (or from other derived types).
- There are three primary derived types:
  - Arrays: a collection of contiguously allocated objects of the same type.
  - Structs: a collection of fields
  - Unions: a way to store different data types in the same memory location.

More Structs

- You can combine declaration and definition:

```
struct point {
   int x;
   int y;
   int z;
}p;
```

```
struct student{
       char name[22];
       unsigned int age;
       char house[15];
} eddie;
```

- Frequently, we create typedefs for structures:

```
typedef struct {
    int x;
    int y;
    int z;
} point;

point p;
```

```
typedef struct {
       char name[22];
       unsigned int age;
       char house[15];
} student;

student eddie;
```

Arrays

- Defined using []
  - `char    carray[10];  // an array of 10 characters`
  - `int     iarrary[52]; // an array of 52 integers`
- Array elements are laid out contiguously in memory.
- All the elements of the array are the same type.
- Elements accessed by index:
  - `carray[3] = 'a';`
  - `iarray[51] = 1234;`

Alignment, Arrays & Pointer Arithmetic (Oh my)

- Pointers, types, and arrays in C are kind of magical!
- Key concept:
  - Given a pointer, `P`, to something of type `T`, `P + i` is identical to `&P[i]`.
  - Corollary: if `P` is a pointer, `&P[0] == P`
- Example:

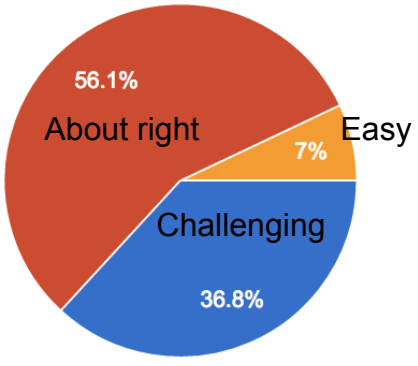
```
int *ip = malloc(10 * sizeof(int));
ip == &ip[0];
&ip[4] == ip + 4;
ip[6] == *(ip + 6);
```

More Pointer Magic

- A pointer is an address.
- Let's say that `P` is a pointer and its value (address) is `0x601060`.
- If `P` is of type `int *` (pointer to an integer), then:
  - &P[0] = 0x601060
  - &P[1] = 0x601064
  - P + 1 = 0x601064

From Thursday's Survey

How Challenging was the Treasure Hunt?



Comfort with C