Data Representation and Storage

- Learning Objectives
  - Define the following terms (with respect to C):
    - Object
    - Declaration
    - Definition
    - Alias
    - Fundamental type
    - Derived type
  - Use size_t, ssize_t appropriately
  - Use pointer arithmetic correctly
  - (After practice in class) Explain how C data types arranged in memory.

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
  - long, unsigned long – signed and unsigned longs
  - short, unsigned short – signed "short" integer (16 bits)
  - char, unsigned char – signed and unsigned characters
  - float, double – although the standard does not define their sizes, on most platforms a float is 4 bytes and a double is 8 bytes we won't talk much about floating point numbers in this course).

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.

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;`

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[20];
        unsigned int age;
        char house[15];
};
```

- Example definition:

```
struct point p;
```

```
struct student margo;
```

More Structs

- You can combine declaration and definition:

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

```
struct student{
        char name[20];
        unsigned int age;
        char house[15];
} margo;
```

- Frequently, we create typedefs for structures:

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

point p;
```

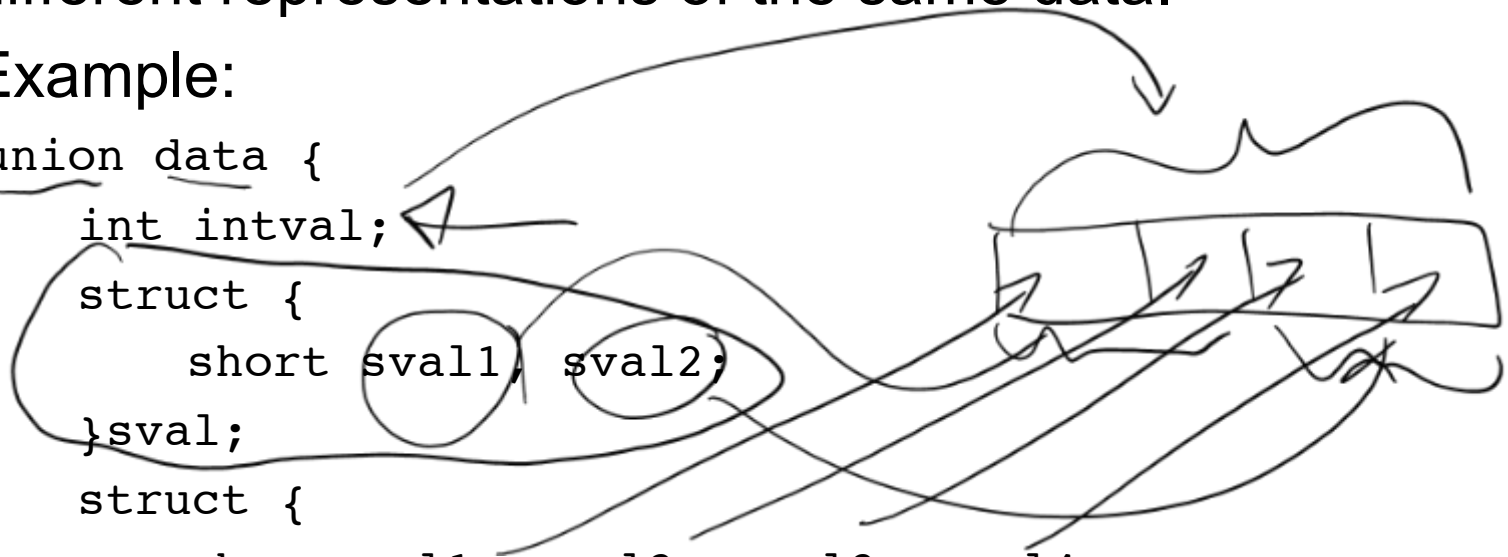
```
typedef struct {
        char name[20];
        unsigned int age;
        char house[15];
} student;

student margo;
```

Unions

- Unions are most frequently used when might want different representations of the same data.
- Example:

```
union data {
    int intval;
    struct {
        short sval1, sval2;
    }sval;
    struct {
        char cval1, cval2, cval3, cval4;
    }cval;
}unionvar;
```



Accessing parts of unions/structs

- We use the “dot” operator to access the different parts of a struct or union.
- Let’s define some variables using the types declared on the previous slides:

```
student margo, *margop;
union data u, *up;
```

`margo.name, margop->name` refer to the name field in the margo structure.

`u.intval, up->intval` refers to the data in the u union, referenced as an integer.

`u.cval, up->cval` refers to the structure in the union containing the four character variables.

`u.cval.cval1, up->cval.cval1` refers to a specific one of those character variables.

Pointer Arithmetic

- 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:

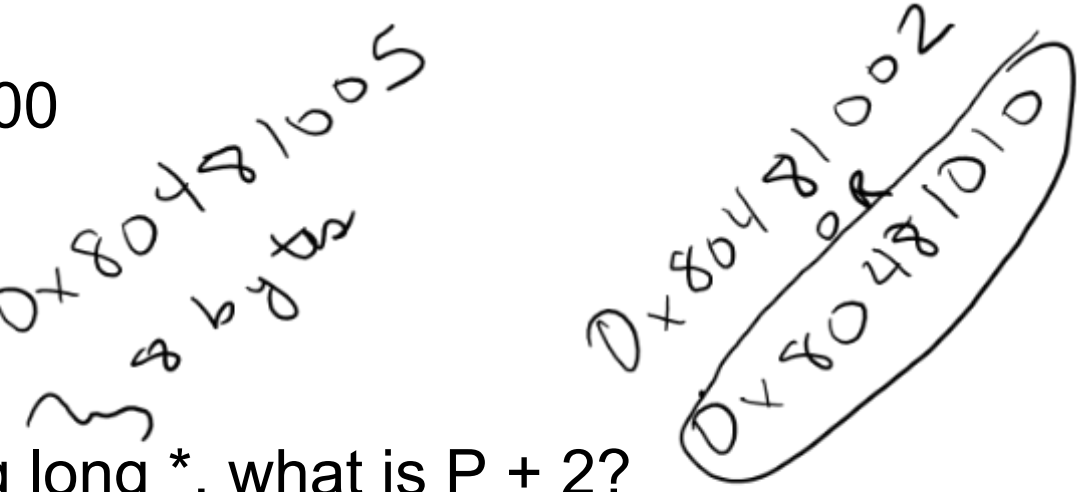
```
int *ip = malloc(10 * sizeof(int));
ip == &ip[0];
&ip[4] = ip + 4;
ip[6] = *(ip + 6);
```

More Pointer Magic

- A pointer is an address.
- Let's say that `P` is a pointer and its value (address) is `0x80481000`.
- If `P` is of type `int *` (pointer to an integer), then:
  - &P[0] = 0x80481000
  - &P[1] = 0x80481004
  - P + 1 = 0x80481004

Practice problem

- Let P = 0x80481000
- Given: Char *P;



- What is P + 5?
- If P is defined long long *, what is P + 2?
- If P is defined type * and sizeof(type) = 16, what is the value of P +1?

P+1 = P(as a plain #) + sizeof(T)

0x80481010