What are Types?
data types

• A **data type** constrains the **set of valid values** a piece of data can take on
  • An `int` in C can take on values from $[-2^{31}, 2^{31} - 1]$ 
  • A `char` in C can take on values from $[0, 255]$ 
  • Not always easy to define this set (what are the sets of valid values for `float`s?) 
  • Some times we express this information explicitly:
    
    ```
    int c = 0
    ```
  • Other times, it’s implicit:
    
    ```
    x = “Hello from Python”
    ```
data types

• Constraining the set of values helps determine many other things
  • How much space it takes up (ints take up 4 bytes, chars take up 1 byte)
  • How to interpret a sequence of bits: 01000001
    • If the data is an int, this is 65
    • If the data is a char, this is ‘A’
• What kinds of operations you can do on it
  • Can add together two ints
  • Cannot add together two bools
more types

• Pieces of data are not the only things that can have types
• **Functions** can have types too!
  ```
  int foo(int i, char c)
  has type \((int \times char) \rightarrow int\)
  ```

• Constrains behavior just like data types do:
  • When I call foo, I need to pass it an **int** and a **char**
  • When I use the return value of foo, I should treat it as an **int**
even more types

• Arrays:
  
  `int a[10]` : means that an array has exactly 10 items of type `int`

• Pointers:
  
  `float ** p` : means a pointer that points to another pointer that points to a `float`

• Structs:
  
  `struct {int x; float f;} s` : means a piece of data that contains an `int and a float`
what can go wrong?

• What can go wrong if we do not pay attention to types?
  • What happens if we generate code to add an int to a float?
  • What happens if we pass the wrong kind of data to a function?
  • What happens if we access past the end of an array?
  • What happens if we use the wrong kind of load to access the first field of a struct?

• In our simulator, many of these operations will trigger a runtime failure (try it!)
  • The simulator does *dynamic type checking* under the hood, but in reality, in many cases you will just get very strange behavior in your program
types as constraints

• Think of types as imposing constraints on the behavior of your program
  • Operations only between matching types
  • Functions called with appropriate arguments [is the previous point just a special case of this point?]

• Different programming languages, compilers, and runtime systems do different things to enforce these constraints
  • Not all constraints are always enforced!
Dynamic Type Checking
what happens?

.section .text
LA t1, 0x20000000
LI t2, 17 ; t2 = 17
SW t2, 0(t1) ; *t1 = t2
FLW f1, 0(t1) ; f1 = *t1

In a “real” machine:

t2 = 0000 0000 0000 0000 0000 0000 0001 0001 (17 in binary)

so

f1 = 0000 0000 0000 0000 0000 0000 0001 0001 (2.38e-44 in floating point)
what happens?

```assembly
.section .text
LA t1, 0x20000000
LI t2, 17 ; t2 = 17
SW t2, 0(t1) ; *t1 = t2
FLW f1, 0(t1) ; f1 = *t1
```

On our simulator:

```
AssertionError: Value in memory not of type <class 'float'>
```
what happens?

Our simulator does some basic **dynamic type checking**

- Keeps track of the type of data stored in memory
- Makes sure that loads and stores respect that type
  - Cannot load an integer value into a floating point register, and vice versa
what is dynamic type checking?

• Types constrain behavior of a program

• If those constraints are not respected, a program can produce weird behavior
  • Or worse, have a security vulnerability!

• Dynamic type checking checks those constraints at runtime to turn constraint violations into runtime errors

• Which constraints are checked, and where, is up to the language/runtime
dynamic checks in python

• Makes sure that operations only work on valid types

\[10 + \text{"x"} \rightarrow \text{TypeError: unsupported operand type(s) for +: 'int' and 'str'}\]

• Makes sure that list accesses are valid

\[x = 5 * [0] ; x[6] \rightarrow \text{IndexError: list index out of range}\]

• *Doesn’t* check that functions are called with the right types (why?)
how does dynamic typechecking work?

• Data carries along *meta-data* that specifies type information

  • Data type, lengths of strings, sizes of arrays, whether a reference is null, etc.

• At **run-time** this meta-data is used to check constraints before performing operations that might give bad behavior if constraints are violated

  • Not all constraints are checked all the time!
what to check?

- Different languages make different choices about what to check
  - Java will check that array access are in bounds, C will not
  - C++ will (sort of) check that downcasts succeed, Java will give a better runtime error
- What happens if a constraint is not checked?
  - Can cause an error lower in the system stack, e.g., a segmentation fault
  - Can cause silent problems (lots of security vulnerabilities!)
when to check?

- Dynamic type checking requires *run-time* processing
  - Adds overhead!
  - Array accesses in Java are much slower than array accesses in C
  - In some circumstances, can offload some of the work of type checking to the compiler, check *before the program even runs*
- This is called **static type checking**!