Types, checking, compatibility

- Range of types supported, associated syntax, and handling of type compatibility are some of the most recognized aspects of any language
- As mentioned earlier:
  - Static typing: types fixed prior to execution
  - Fixed dynamic typing: types determined during execution, but cannot be changed once set
  - Dynamic typing: types can change over time, are generally associated with the currently stored value, not explicitly with the variable/expression

Type conversions

- Type conversion: translating a value of one type into a “matching” value of another type)e.g. “37” -> 37)
  - Implicit conversions: performed automatically, require no explicit instruction from programmer, e.g. float x = 3; // 3 -> 3.0
  - Explicit conversions: programmer includes an instruction or directive stating they want a converion to take place, e.g. float x = (int)(3);
- For type conversion to take place, compiler/interpretter must know how to perform that conversion (some such conversions typically built-in, language may allow programmer to specify how others are to be handled)

Widening, narrowing conversions

- Different types, while abstractly compatible, might support different ranges of values, e.g. `int a;` vs `long b;` and require different amounts of memory to store
- Widening conversions: converting from a data type with a smaller range to a type with a larger range: must decide how to “fill in” the extra bits of storage, e.g. `b = a;`
- Narrowing conversions: converting from a data type with a wider range to a type with a larger range: must decide how to truncate the excess data, e.g. `a = b;`

Conversion warnings and errors

- Compiler/interpretters usually warn about narrowing conversions (obvious possibility for loss of data), but not about widening conversions
- Some narrowing conversions not obvious, e.g. suppose we had 32 bit floats and 32 bit ints

```
int i; // can hold ints in range +/-2billion(ish)
float f; // can hold floats in range +/-3x10^38(ish)
f = i;  // actually a narrowing conversion
// f uses some bits for precision, some for exponent
// whereas i can actually have 32 bits of precision
```

Type compatibility

- Checking if the operands for an operation have acceptable types for the operation, possibly allowing implicit conversions to take place first, e.g.

```
float x; int y;
.........
x = 5 * y;
```

  - Need to check if 5 and y are compatible with *
  - Then need to check if result and x are compatible with =

Static vs dynamic type checking

- Static checking takes place at compile time

  Really only possible where types are statically bound
- Dynamic checking takes place at run time
- Dynamic more flexible, but often requires more specific programmer involvement, results in slower execution
- Languages are strongly typed if all type checking can be done statically, weakly typed otherwise (most languages are weakly typed, to greater/lesser degrees)
- Features like type casting, void pointers, and unions often allow programmers to bypass regular type-checking processes

Name vs structural type checking

- Name checking: two items have the same data type if the names of their types are identical
- Structural checking: simply need the underlying content to be of the same types

```
struct Foo { int i; float f; } x; // var x of type Foo
struct Blah { int a; float b; } y; // var y of type Blah
```

- x and y are type compatible under structural checking, but not under name checking

Pros/cons of named type checking

- Name-based type checking gives developers less flexibility in terms of implicit type conversions/compatibility
- Name-based type checking allows developers to use the name as a means of error detection, e.g.

```
// implement types for temperatures and speeds as floats
typedef float Temperature;
typedef float Speed;
Speed s;
Temperature t;
s = t; // name-based checking warns us that we're mixing types
```