Types can be broken into two major categories:

1. **Concrete Types** have an implementation directly visible to the programmer.
   For example:

```
typedef struct complex
{
  double re, im;
}
```

2. **Abstract Types** hide their implementation from the user, and are defined by operations.
   We can think of 'float' as abstract, since f, e, and s vary in location by endian-ness.

There is a sort of battle between abstraction and efficiency (low level access).

Given that we can define multiple types, there must be a well defined definition of equivalence.
There are two standard approaches to this:

1. **Structural Equivalence**
   $\equiv$ two types are the same if and only if their structure is the same.
   This is equivalent to saying that two types are assumed to be the same unless they behave differently.
   This approach tends to be most applicable to concrete types.

```
typedef int T;
typedef int U;
T v;
U *p = &v;
```

2. **Name Equivalence**
   $\equiv$ two types are the same if and only if they have the same name.
   This approach tends to be most applicable to abstract types.

```
struct R {double re, im;}
struct S {double re, im;}
struct R v;
struct S *p = &v; // ERROR
```

Suppose we wanted to implement structural equivalence in C. Then R and S would be equivalent, which leads to complications, for instance we can do:

```
struct T {int val; struct U* pair;};
struct U {int val; struct T* pair;};
// and these would be equivalent as well
```

These complications were avoided for C, which was optimized for efficiency.

Type equivalence is more complicated when it comes to subtypes.
The behavior we want is: $T == U \iff T \subseteq U \;\&\&\; U \subseteq T$.
Thus consider the following code in PASCAL

```
type alpha = 'a'..'z';
var a: char; var b: alpha;
a := b; // legal
b := a; // trouble -- might be out of range
```

What do we do in this situation? We have two options:

1. report an error at compile time
2. report a runtime error if the bounds are violated.

Pascal chooses the latter, so we say it uses *soft* type checking.
This is called **subtype polymorphism**. It appears in object-oriented languages within the idea of a subclass.
A subclass is more restrictive than its parent, which lets it have more operations

```
Class C : public p;
P *p = C; // ok
C *p = P; // bad
```

For an example, consider C's char* and char const*; which is the type and which is the subtype? Consider:

```
char buf [2000];
char *p = buf;
char const *q = buf; // allowed, we just can't modify buf via q
q = 'x'; // BAD
// whereas
char const str [] = "xyz";
char const *q = str; // OK
char *p = str; // allowed, but can cause runtime errors
*p = 'x'; // allowed, but we may get s SIGSEGV
```

Since char* allows for operations that const char* doesn't, char* is a subclass of char const*

A harder example: char const* const* vs char **q

```
char **q = char const * const *p; // BAD
char const * const *p = char **q; // BAD in C due to special type rules;
// was decided to be a special case in C++
```

Since we know const restricts our operations, char** is a subclass of char const * const *.

Rule Complexoty is a double-edge sword, since strong type checking increases complexity.
This introduces the idea of **polymorphism** ≡ the property of a symbol operation to accept multiple 'forms' For example, in C a + b works whether a,b are 'int'/'float'/'long'/'ulong'.

This is an example of compile-time polymorphism, which is in statically checked languages.
In Python, this happens at runtime.
How does this happen (in C)?

```
double f(double a, double b) {return a + b;}
float g(float a, float b) {return a + b;}
// these compile to different machine code
```

Polymorphism can fall into two major categories:

1. **Ad-Hoc Polymorphism**
   This has the following properties:

   - accept a finite number of types.
   - grew up organically.
   - have coherent individual rules, but little overall organization.

   This can be broken down into two subcategories:

   (a) **Overloading** ≡ the act of identifying operations and functions by types/content of parameters.
   Functions with the same name will have different ABI/API, and the compiler chooses by type.
   The above two functions are an example, which may beg the question, "Wouldn't it be better to just operate on the 64 bit float always?"
   NO

   - it would be slower
   - it would round incorrectly for 32 bit arguments

   Thus we choose to implement two functions of the same name.
   During assembly, the compiler does **name mangling** for identification
   ≡ adding illegal characters into names specified by the user.
   This makes compiling C with other languages tough, since names won't match.

   (b) **Coercion**
   ≡ implicit type conversion required for an expression to be valid.
   Consider the '+' operator in C. It:

   - accepts combinations of int, uint, long, ulong, long long, ulong long float, double, long double.
   - does not accept char, uchar, short, ushort.

   We would thus expect to have 196 variations of the operator, but with coercion we only need 4.
   This is because coercion follows the following rules:

   - if sizeof(t) $<$ sizeof(int), then cast T to int.
   - if sizeof(T_1) $>$ sizeof(T_2), cast T_1 to type(T_2).
   - else, cast both numbers to unsigned.

   This can result in some unexpected behavior:

```
int i, j;
long l;
i = j + l; // trap
// sometimes the values can even change
int i = -1;
unsigned j = i; // changed value; j == UINT_MAX
unsigned short k = i; // loses info
assert(i == j);
// this can even happen implicitly on comparison
int i = -1;
unsigned z = 0;
assert(i < z); // EXCEPTION -- i is converted to unsigned
// even literals are not immune to this behavior
assert(-1 < 2^34); // EXCEPTION -- 2^34 is converted to int, where it overflows
assert(-2^32 < 0); // EXCEPTION -- 2^32 is converted to unsigned, then negated,
                   // but since it is unsigned, it is still nonnegative
```

These two types of ad-hoc polymorphism can interact to cause undefined behavior:

```
inf f(double x, int y);
int f(int x, double y);
f(3, 5); // which one does it call?
```

2. **Parametric Polymorphism** ≡ the ability of a function or operation to take an unlimited number of types.
   Consider Java; prior to 2005, there was no polymorphism, so a class may be of the form:

```
public interface List {
  void add (Object);
  Iterator iterator();
} // these are required for any object
public interface Iterator {
  void remove(); // remove the last item accessed
  Object next(); // gets the next item in the list
  bool hasNext();
}
List l;  ... l.add("abc"); ...
for (Iterator i = l.iterator(); i.hasNext())
  if ((string)(i.next()).length() == 1) i.remove(); // removes all lengt
```

   Java users complained about this runtime check and clutter for years, so Java added **generic types**.

```
public interface List <E> {
  void add (E);
  Iterator<E> iterator();
}
public interface Iterator <E> {
  E next();
  bool hasNext();
  void remove();
}
for (Iterator<string> i = l; l.hasNext())
  if (l.next().length() == 1) i.remove();
```

   We have seen this idea before with C++ **templates**. These two ideas form the basic techniques of parametric polymorphism:

   (a) Templates (C++, Ada)
   ≡ a stand in for not yet compiled code
   - faster and lower level
   - full checking and optimization permitted

   For instance: List<E> stands for List<int>, List<double>, etc, BUT for each required type, a separate form of the function must be compiled.

   (b) Generics (OCaml, Java, newer languages in general)
   ≡ only the translated machine code for all types used is copied
   - all checking is done on compilation
   - more streamlined, less optimizeable
   - represents all objects by pointers