In addition to hardware/software interface evolution, languages can evolve too.
Languages like C/C++ have evolved so much that programs from the 70s couldn't run today.
There are even "software archaeologists" who specialize in translating this code.
The point is, we have a compatibility issue that we need to plan to address.

Let's consider some specific examples:

- BASIC:
  - was developed for the GE model 225 CPU. Specs:
    * about 40 microseconds per addition, and 500 microseconds per division.
    * about 40 KiB RAM, which was huge for the time. This is big enough that the device could fit all of BASIC in its RAM!
  - BASIC is a small base language which supports many extensions.
  - While this was critical at the time due to limited memory, it is now inconsequential.
- C
  - was developed for the PDP II Minicomputer CPU. Specs:
    * about 4 microseconds per addition and 1.2 microsecond cycle time
    * about 16 KiB RAM
  - C allows for fast memory access, but comparatively slow computation;
    This means that pointers were a very efficient way to manipulate data.
  - Modern computers can do computation easily, but access RAM comparatively slowly.

These languages can still run on modern devices, but their performance is worse.
We can, however, anticipate these changes
For instance, modern languages are not conducive to GPU (SIMD)/cloud computing (MIMD).
All of these issues are issues of **scale**; languages may not scale well to:

- larger machines with increased array size and multiple CPUs
- complex problems that require multiple programmers

We address these issues in a few object-oriented ways.

Names/Identifiers

These are arbitrary within a program, so why is this the first thing we discuss?
Choosing naming conventions is a major issue within software engineering.
There are two aspects to identifiers within a program:

- Binding Time
- Scope

We face a couple common issues:

1. Permitted Characters
   - In FORTAN, spaces were permitted within names. This led to serious errors.
     In the code controlling the Mariner 1 rocket, the following code was intended

     ```
     DO 10 I = 1,10
       FOO(I)
     10 continue
     ```

     but a period was typed instead of a comma, causing the code to be interpreted as

     ```
     DO 10 I = 1.10
     FOO(I)
     ```

     This resulted in the crashing of the rocket!
   - In C++, an identifier is of the form [a-zA-z_][a-zA-Z_0-9]*
     Characters from other languages were initially prohibited, but this changed in C++11
     This added complexity, since the latin and Cyrillic o's look similar but are encoded uniquely.
     Additionally, only some parts of names are case sensitive:

     ```
     double e = 1e16;
     double E = 1E16;
     assert(e == E > memcmp(e, E, sizeof(e)));
     ```

2. Reserved Words

   ```
   if (class < 12) return; // legal in C, not in C++
   ```

   Clearly, adding keywords to a language can add complexity and invalidate legacy code.
   We thus reserve names early in a language's development.
   For example, C/C++ reserve names starting with '_', so _Noreturn could safely be added in C11.

Binding

≡ an association between a name and a value.
We often access bound data via its mapping in the symbol table. These bindings can be deceptively simple:

```
short a = 10; // a is bound to 10
short *p = &a; // pointer is bound to the address of a
assert(int *p = &10); // ERROR: literals do not have memory addresses
assert(sizeof{10} = sizeof(a)); // NO -- 10 is 4 bytes but a is only one
```

Binding is done in one of two ways:

1. **Explicit Binding** ≡ binding written directly into a program
2. **Implicit Binding** ≡ binding done as a consequence of a program

Implicit binding is often considered bad practice, so explicit binding is often encouraged.
In many languages, types must be declared explicitly so as to avoid errors.
These languages bind variables to values between compilation and linkage, in a time called **binding time**.
To understand the different periods of binding, consider the following code:

```
void main()
{
  for (int i = 0; i < 10; i++) foo(i);
}
```

Portions of this function were bound at different times (listed chronologically):

1. 'void' and 'for' were bound at language **definition/authorship time**
2. 'int' was bound to 4/8/etc bytes at **language implementation time**
3. 'foo' was bounded to its definition and reference at **link time**
4. 'foo' and 'main' were bounded to their memory locations at **load time**
5. 'i' is bounded to multiple values at **runtime**

Even with the support for explicit binding, even C cannot avoid it completely.
Macros are expanded prior to runtime, but allow for implicit binding like so:

```
#define Foo 27
#IFDEF FOO ...
// if we typed FOO instead,
// we would get no error,
// but the if branch would be skipped
```

OCaml is considered a safer language than C, but OCaml uses nearly all implicit binding.
How is this possible? OCaml is safe because of redundancy and type checking.
We can do this in gcc with an option which bans implicit binding
   BUT even that wouldn't fix the above issue.

Types

Finding an exact definition of types is difficult, so we will use a few. This is the simplest:
   ≡ a set of values/objects
Defining a type explicitly via listing is called **enumeration**, and is used in functional languages like Lisp.
In computer languages, these enumerations are often defined in terms of **native types**.
Any type that can be constructed of native types is called a **constructed type**.
Object-Oriented languages often use another definition:
   ≡ a set of values and a set of operations on those values
Adding the idea of operations allows for observation of values and performance of actions; consider:

```
class complex
{
  double re;
  double im;
  double dist() {return sqrt(re*re + im*im);}
}
```

The above definitions can be used to analyze the C primitive 'float':

- is float a set of values? YES – 0, 0.2, 0.22, ...
- is float a representation? YES – defined as follows:

For many years, this implementation was hidden;
since not specified by the spec, implementations tended to be machine-dependent.
In the current day, one layout has won out, called IEEE-754:
Numbers are 32 bits, and represented by the following sequential sections:

1. the sign bit
2. the eight bit exponent value
3. the 23 bit fractional value

We interpret the float value (sef) according to the following rule:

$$\text{value}(sef) = \left\{ \begin{array}{lr} (-1)^s * 2^{e-127} * 1.f & \text{for } o < e < 255 \text{(normalized)} \\ (-1)^s * 2^{e-127} * 0.f & \text{for } e = 0 \text{ (denormalized)} \\ (-1)^s * \infty & \text{for } e = 255 \text{ \&\& } f = 0 \\ \text{Not a Number (NaN)} & \text{for } e = 255 \text{ \&\& } f! = 0 \end{array} \right\}$$

This implementation results in a couple special cases:

- there are two distinct zeroes
  → two variables can have different data with the same value.
- NaN evaluates false no matter what it is compared with
  → two variables can have the same data and different values.

Floats can trigger a few exceptions:

1. overflow – exponent too large (1e308 x 1e308)
2. underflow – exponent too small (1e-307 x 1e-307)
3. bad arguments – (0.0/0.0, sqrt(-1))
4. inexact results – (1.0/10 != 0.1)

Traditionally, hardware would trap on exceptions 1-3 and ignore 4, but now they all return special values.

Types have a few ways to avoid exceptions:

- **Annotations**
  Types document programs, and the compiler uses this documentation to report errors. (ex. int val;)

- **Inference**
  The compiler infers the type from the code (ex. 1.0, 1.0f, 1)
  There are two standard approaches:
  - bottom-up – C: 3 + 4.5 + 'a' → float
  - top-down – OCaml: [3, 4, 5] = (f y z) → int -> int -> int

- **Checking**
  In a strongly typed language, all operations are checked for type matches prior to being run.
  The benefits: it is safer, and there is no mistaking the type.
  The downsides: it is finicky, and can disallow legal operations.
  C and C++ are strongly typed (except void pointers can subvert this typing).
  OCaml and Java are strongly typed with no exceptions.
  Python is weakly typed with no exceptions.