Type Safety

Type safety

- Each object is ascribed a **type** (`int`, pointer to `int`, pointer to function), and
- Operations on the object are always *compatible* with the object's type
  - Type safe programs do not "go wrong" at run-time
- **Type safety** is **stronger** than memory safety

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p;
cmp("hello","bye"); // crash!
```

Memory safe, but not type safe

Dynamically Typed Languages

- **Dynamically typed languages**, like Ruby and Python, which do not require declarations that identify types, can be viewed as **type safe** as well
- Each **object** has **one type: Dynamic**
  - Each operation on a Dynamic object is permitted, but *may be unimplemented*
  - In this case, it *throws an exception*

Well-defined (but unfortunate)

Enforce invariants

- Types really show their strength by **enforcing invariants** in the program
- Notable here is the enforcement of **abstract types**, which characterize modules that keep their **representation hidden** from clients
  - As such, we can reason more confidently about their **isolation** from the rest of the program

For **more on type safety**, see
http://www.pl-enthusiast.net/2014/08/05/type-safety/

Types for Security

- **Type-enforced invariants can relate directly to security properties**
  - By expressing stronger invariants about data's privacy and integrity, which the type checker then enforces
- **Example**: **Java with Information Flow (JIF)**

```
int{Alice→Bob} x;
int{Alice→Bob, Chuck} y;
x = y; //OK: policy on x is stronger
y = x; //BAD: policy on y is not
       //as strong as x
```

Types have *security labels*

Labels define what information flows allowed

http://www.cs.cornell.edu/jif

Why not type safety?

- **C/C++** often chosen **for performance** reasons
  - Manual memory management
  - Tight control over object layouts
  - Interaction with low-level hardware
- **Typical enforcement** of type safety is **expensive**
  - **Garbage collection** avoids temporal violations
    - Can be as fast as malloc/free, but often uses much more memory
  - **Bounds** and **null-pointer checks** avoid spatial violations
  - **Hiding representation** may **inhibit optimization**
    - Many C-style casts, pointer arithmetic, & operator, not allowed

Not the end of the story

- **New languages** aiming to **provide similar features** to C/C++ while **remaining type safe**
  - Google's **Go**
  - Mozilla's **Rust**
  - Apple's **Swift**
- **Most applications do not need C/C++**
  - Or the risks that come with it

These languages may be the future of low-level programming