CptS 355- Programming Language Design

Types and Type Checking

Instructor: Sakire Arslan Ay

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Types

- What is “type” in a programming language? What is it and what does it do?
  - Type defines a collection of values that share common properties - usually a common set of operations.
  - Type tells you what is legal to do with some value in the language.

Types

What is a **type error**?

- An attempt to use a value in an operation inconsistent with the value's type.
- Examples:
  - The following will produce a type error in all languages.

    ```
    x = 17
    x (  );
    ```

  - The following may produce a type error in some programming languages

    ```
    3 + 4.5
    ```

Compile-time (static) type checking versus Run-time (dynamic) type checking

- Compile-time (static) type checking:
  - Examples (Haskell):
    - Checking that all return values of a function have the same type
    - Checking that all patterns are exhaustive
  - Compile-time type checking is necessarily conservative:
    - it may flag as an error something that would not ever cause a run-time error.
  - Advantages:
    - Less runtime overhead
    - The whole program is checked

Compile-time (static) type checking versus Run-time (dynamic) type checking (cont.)

- Run-time (dynamic) type checking:
  - Run-time type checking is expensive - must be done each time the program is executed.
  - Advantages:
    - Allows certain programming styles not possible with compile-time type checking.
    - More flexible data structures
      - For example, lists in Python may contain values of any type where lists in Haskell must have elements of the same type.

Type Safety: strong typing

- Ensure that every use of a value is compatible with its type.
  - Static strong typing → compiler error
    - Example : Haskell
  - Dynamic strong typing → error at the point of misuse.
    - Example: Python

- In type-safe languages, values are managed "*from the cradle to the grave*":
  - Objects are created and initialized in a type-safe way.
  - An object cannot be corrupted during its life time: its representation is in accordance with its type.
  - Objects are destroyed, and their memory reclaimed, in a type-safe way.
  - Any change of type requires explicit conversion

Type Safety: strong typing

- C doesn't have type safety
  - C heap values are created in a type-unsafe way.
  - C casts and unchecked array accesses can corrupt memory during its lifetime.
  - C deallocation is unsafe, and can lead to dangling pointers.

```
void f(char* char_ptr) {
   double* d_ptr = (double*)char_ptr;
   (*d_ptr) = 3.5;
}

int *i = (int*)malloc(sizeof(int));
int j = 0;
*i = 4;
free(i);
/* ... some code that might allocate memory, then: */
*i = j; /* Use deallocated memory. */
```