

Spring 2019

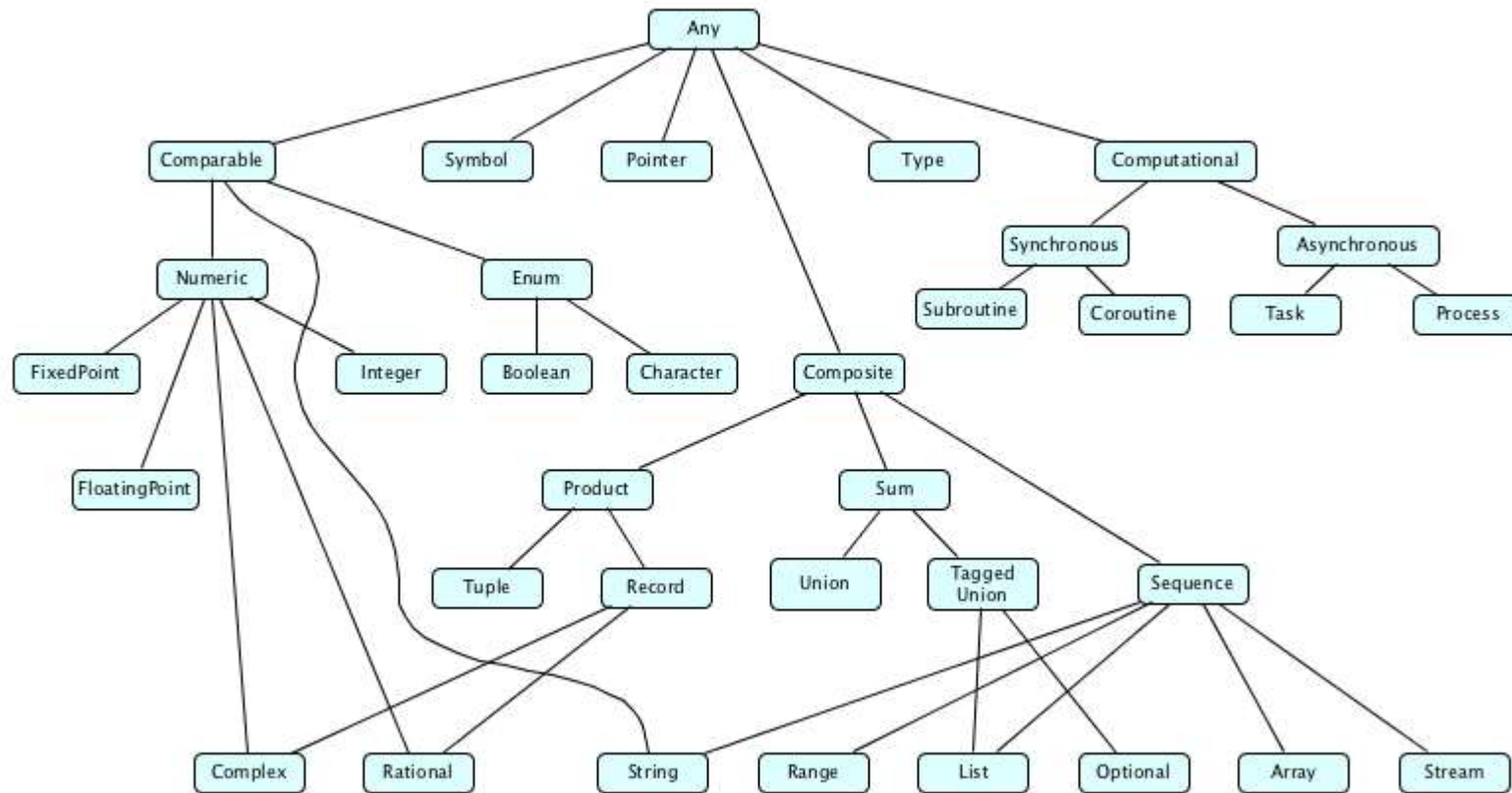
# CSE 216 : Programming Abstractions

---

TOPIC 3 – DATA TYPES AND TYPE SYSTEMS

# What is a type?

- A type consists of set of values and a set of allowable operations.



# Classification of Types

---

- What has a type? – Things that have values
  - Constants, variables, fields, parameters, subroutines, objects
- Classification of types:
  - Constructive: A type is built-in (e.g. Integer, Boolean) or composite (records, arrays).
  - Denotational: A type is a set or collection of values. (e.g. enum)
  - Abstraction-based: A type is defined by an interface, the set of operations it supports. (e.g. List)

# Definition of Types

---

- Defining a type has two parts:
  - A type's declaration introduces its name into the current scope.
  - A type's definition describes the type (the simpler types it is composed of).

Here are some examples of declarations that are not definitions, in C:

```
extern char example1;  
extern int example2;  
void example3(void);
```

Here are some examples of declarations that are definitions, again in C:

```
char example1; /* Outside of a function definition it will be initialized to zero. */  
int example2 = 5;  
void example3(void) { /* definition between braces */ }
```

# Type System

---

- Mechanism for defining types and associating them with operations that can be performed on objects of each type:
  - Built-in types with built-in operations
  - Custom operations for built-in and custom types

# Type System

---

- A type system includes rules that specify
  - Type equivalence: Do two values have the same type?  
(Structural equivalence vs name equivalence)
  - Type compatibility: Can a value of a certain type be used in a certain context?
  - Type inference: How is the type of an expression computed from the types of its parts?

# Type Errors

---

- A type error is a program error that results from the incompatible use of differing data types in a program's construct

```
int n = "n";
```

- To prevent (or at least discourage) type errors, a programming language puts in rules for type safety.
  - Type safety contributes to a program's correctness.
  - But keep in mind that it does not guarantee complete correctness.
  - Even if all operations in a program are type safe, there may still be bugs.
  - E.g., division of one number by another is type safe, but division by zero is unsafe unless the programmer explicitly handles that situation in some other manner.

# Type Checking

---

- Type checking is the process of verifying and enforcing the rules of type safety in a program.
- This may be done at compile-time, called static typing (and the language is called a statically typed language).
- Or, it may be done at runtime, which is known as dynamic typing (and the language is called a dynamically typed language).
- Another way to distinguish between the type checking in a language is based on how strongly it enforces the conversion of one data type to another.
- If a language generally only allows automatic type conversions that do not lose information, it is called a strongly typed language.
- Otherwise, it is called weakly typed.



# Common Kinds of Type Systems

---

- Strongly typed
  - Prohibits the application of an operation to any object not supporting this operation.
- Weakly typed
  - The type of a value depends on how it is used
- Statically typed or **typed**
  - Strongly typed and type checking is performed at compile time (Pascal, C, Haskell, SML, ...)
- Dynamically typed or **untyped**
  - Strongly typed and type checking is performed at runtime (LISP, Smalltalk, Python, ...)

# Common Kinds of Type Systems

---

- Programming languages are often classified according to some of the major programming paradigms – procedural, functional, and object oriented.
- Within each paradigm, some languages are typed while others are untyped.
- Within computer science, “**untyped**” really means **dynamically typed**.
- That is, a variable or an expression is assigned the type of the corresponding data (i.e., the “value” denoted by the variable or the expression).
- Similarly, “**typed**” typically means **statically typed**.

# Type Systems: Examples

---

- Java is **strongly typed**, with a non-trivial mix of things that can be checked **statically** and things that have to be checked **dynamically** (for instance, for dynamic binding):

```
String a = 1;           //compile-time error
int i = 10.0;           //compile-time error
Student s = (Student) (new Object()); // runtime
```

- Python is **strong dynamic** typed:

```
a = 1;
b = "2";
a + b      run-time error
```

- Perl is **weak dynamic** typed:

```
$a = 1
$b = "2"
$a + $b      no error.
```

# Static and Dynamic Binding in Java

---

- Static binding happens at compile-time while dynamic binding happens at runtime.
- Binding of private, static and final methods always happen at compile time since these methods cannot be overridden.
- When the method overriding is actually happening and the reference of parent type is assigned to the object of child class type then such binding is resolved during runtime.
- Illustrative example

## Trade-offs

---

- Strong static type checking (e.g. C)
  - + type errors are caught early at compile time
  - verbose code
- Strong dynamic type checking (e.g. Python)
  - + quick prototyping with lesser ‘amount’ of code
  - type errors are caught only at runtime
- Weak dynamic type checking (e.g. Perl)
  - + least verbose code writing
  - type errors are often not caught even at runtime
  - unintended program behavior may occur due to implicit type conversion at runtime

# Advantages of Type Systems

---

- **Documentation/legibility** – Typed languages are easier to read and understand since the code itself provides partial documentation of what a variable actually means.
- **Safety** – Typed languages provide early (compile-time) detection of some programming errors, since a type system provides checks for type-incompatible operations.
- **Efficiency** – Typed languages can precisely describe the memory layout of all variables, since every ‘instance’ of an ‘object’ of a certain type will occupy the same amount of space.
  - Except for dynamically resizing objects like a list.
  - But even then, we at least know how much memory each ‘cell’ of the list will occupy.
- **Abstraction** – Typed languages force us to be more disciplined programmers. This is especially helpful in the context of large-scale software development.

## Type System Rules

---

- A *type system* has rules for:
  - *type equivalence*: when are the types of two values the same?
  - *type compatibility*: when can a value of type A be used in a context that expects type B?
  - *type inference*: what is the type of an expression, given the types of the operands?

**a : int            b : int**

-----

**a + b : int**

# Type Equivalence

---

- The meaning of basic operations such as assignment (denoted by = in C) is specified in a language definition.

- Consider the statement:

$x = y;$

- Here the value of object y is copied into the memory locations for variable x.
- However, before an operation such as an assignment can be accepted by the translator, **usually the types of the two operands must be the same (or perhaps compatible in some other specified way).**



# Type Equivalence

---

- When do two given expressions have equivalent types?
- There are two possible approaches:
  1. Name equivalence: two types are equal if and only if they have the same constructor expression (i.e., they are bound to the same name)
  2. Structural equivalence: two types are equivalent if and only if they have the same “structure”.

# Name Equivalence

---

- Two types are equal if, and only if, they have the same name.

```
typedef struct {  
    int data[100];  
    int count;  
} Stack;  
  
typedef struct {  
    int data[100];  
    int count;  
} Set;  
  
Stack x, y;  
Set r, s;
```

- In case of name equivalence, following are valid:

x = y;

r = s;

- However, following is invalid:

x = r;

# Structural Equivalence

---

- Two types are equal if, and only if, they have the same "structure".
- Check equivalence by expanding structures all the way down to basic types

```
type student = record
  name, address : string
  age : integer

type school = record
  name, address : string
  age : integer

x : student;
y : school;
```

$x = y$ ; is valid in structural equivalence

$x = y$ , is not valid in name equivalence

# Structural equivalence

---

- Most languages agree that the format of a declaration should not matter:

```
struct { int b, a; }
```

is the same as the type:

```
struct {  
    int a;  
    int b;  
}
```

# Type equivalence

---

- Most modern languages use name equivalence because they assume that
  - If a programmer has gone through the trouble of repeatedly defining the same structure under different names,
  - Then s/he probably wants these names to represent different types.

# Alias Type

---

With name equivalence, it is sometimes a good idea to introduce *synonymous* names (e.g., for better readability of programs):

- `TYPE new_type = old_type; (* Modula-2 *)`

The `new_type` is called an **alias** of the `old_type`. This makes sense if we want to create synonymous types such as

- `TYPE human = person;`
- `TYPE item_count = integer;`

# Alias Type

---

```
TYPE stack_element = INTEGER;          (* alias *)
MODULE stack;
IMPORT stack_element;
EXPORT push, pop;
...
PROCEDURE push(elem : stack_element);
...
PROCEDURE pop() : stack_element;
...
```

- Stack is meant to serve as an abstraction that allows the programmer, to create a stack of any desired type (in this case INTEGER).
- If alias types were not considered equivalent, a programmer would have to replace every occurrence of `stack_element` with `INTEGER`.

# Extern in C

---

- Extern refers to external variables – also known as global variables

Here are some examples of declarations that are not definitions, in C:

```
extern char example1;  
extern int example2;  
void example3(void);
```

Here are some examples of declarations that are definitions, again in C:

```
char example1; /* Outside of a function definition it will be initialized to zero. */  
int example2 = 5;  
void example3(void) { /* definition between braces */ }
```

- Demo: <http://tpcg.io/hyELzJ>
- Further reading: <https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/>



# Type Cast/Coercion

---

- In statically typed languages, usually the values expected in a context must be of a certain type.
- E.g., in the assignment statement **x := expression**, the right-hand side expression is expected to have a type that is equivalent to that of x.
- ```
var a : integer; b, c : real;  
    ...  
    c := a + b;
```
- Type coercion refers to automatic, implicit conversion of one type to the expected type.
- Type cast refers to explicit type conversion done by a programmer.

# Type Coercion Example

---

- Used when you are mixing data types. If you assign a value to a variable that is not the same type as the variable, C++ may do the conversion for you; this is coercion.

```
int i;
```

```
float f;
```

```
i = 5.6 // i will be 5
```

```
f = 10 // f will be 10.0
```

```
f = 5.3 / 2 + 1 // f will be 5.3/2.0 + 1.0
```

```
f = 5 / 2 + 4.5 // f will be 6.5
```

- Coercion in JavaScript (Weak dynamically typed language):  
<https://dorey.github.io/JavaScript-Equality-Table/>

# Type Casting Example

---

- Used when the programmer wants to explicitly convert one data type to another. It shows the programmer's intention as being very clear.

```
int i;  
float f;  
i = 5.2 / f; // warning "loss of precision"  
  
i = int(5.2 / f) // no warning but still loss  
f = float(3 * i)
```

# Various Data Types

## Scalar and Composite Types

---

- At the very top, types can be divided into scalar and composite types.
- A scalar type is a type whose values
  - occupy a fixed amount of memory, and
  - are atomic, that is, a value is not subdivided further in any way.
  - e.g., int in C, C++, and Java.
- A composite type is a type whose values are composed of simpler component values. That is, it is an aggregation of other simpler types.
  - Each component can be a scalar type, or itself be a composite type.
  - e.g., int[ ] in C, C++, and Java.

# Various Data Types

## Primitive Types

---

- A primitive type is a type that is not defined in terms of any other type.
- All languages have some fixed set of primitive types.
  - E.g., In Java: int, short, long, byte, char, float, double, boolean
- Primitive types are the basic building blocks for other data types.
- Some people consider “reference” or “pointer” to be a separate primitive type.
- All languages have the primitive types ‘built-in’.
  - Some languages also offer a few built-in composite data types (e.g., Tuple in ML and Python, List in Java and Python).
  - We do not consider these built-in types to be primitive.

# Various Data Types

## Discrete Types

---

- Types can be discrete (countable/finite in implementation):
  - **boolean:**
    - in C, 0 or not 0
  - **integer types:**
    - different precisions (or even multiple precision)
    - different signedness
  - **floating point numbers:**
    - only numbers with denominators that are a power of 10 can be represented precisely
  - **decimal types:**
    - allow precise representation of decimals
    - useful for money: Visual Studio .NET:  
**decimal myMoney = 300.5m;**

# Various Data Types

## Character and String Types

---


- In most languages, characters are internally represented as an integer type.
- Each integer character code represented a particular char value.
  - Depending on the encoding (ASCII, Unicode, Latin-1, etc.), a char is represented by a 8, 16, or 32-bit integer.
- Since text information is important in computation (e.g., a compiler), many languages provide String as a built-in data type, internally represented by a sequence of characters (i.e., it is not a scalar or a primitive).
- Strings in C
  - A string is represented as a `char[ ]`
  - They can be of arbitrary length. The end of a string is denoted by NULL (which is `'\0'`).

# Various Data Types

## Ordinal Types

---

An **ordinal type** is a type where the values belong to a finite range of integer values.



Clearly, all integer types are ordinal types.



Characters are ordinal types.



Boolean is an ordinal type as well.



# Various Data Types

## Enumeration Types

---

- An enumeration type or enumerable type consists of a (typically small) finite set of values explicitly enumerated by the programmer.
  - Thus, an enumeration type is also an ordinal type.

```
/* C, C++, or Java definition of an
   enumeration type for the months in a year
*/
enum Month {
    JANUARY,
    FEBRUARY,
    MARCH,
    APRIL,
    MAY,
    JUNE,
    JULY,
    AUGUST,
    SEPTEMBER,
    OCTOBER,
    NOVEMBER,
    DECEMBER
}
```

# Various Data Types

## Subrange Type

---

- The subrange type is a contiguous sequence of an ordinal type. E.g., we could define a subrange types as follows:
  - `type teen = 13 . . 19;`
  - `type test_score = 0 . . 100;`
  - `type workday = MONDAY . . FRIDAY;`
- Subrange types are defined in PASCAL and all languages derived from it.
- They are also defined in Python, and frequently used as a constraint to define a subset of a data type:
  - `for x in range(1:10):`

# Various Data Types

## Composite Type

---

- A composite data type or a compound data type is a type that can be constructed from the language's primitive data types and other composite types.
  - Record (or union)
  - Array
  - Set
  - List
  - Map
  - Files

# Various Data Types

## Records

---

- A record consists of a number of fields:
  - In C it is implemented using struct keyword:

```
struct Vector {  
    float x;  
    float y;  
    float z;  
};
```

```
struct Color {  
    unsigned int red;  
    unsigned int green;  
    unsigned int blue;  
};
```

```
struct Vertex {  
    Vector v;  
    Color c;  
};
```

- There is a way to access the field:
  - `foo.bar;` <- C, C++, Java style.
  - `bar of foo` <- Cobol/Algol style

# Various Data Types

## Records

---

- Nested record definition in Pascal:

```
type ore = record
  name : short_string;
  element_yielded : record
    name : two_chars;
    atomic_n : integers;
    atomic_weight : real;
    metallic : Boolean
  end
end;
```

- Accessing fields:
  - ore.element\_yielded.name
  - name of element\_yielded of ore

# Various Data Types

## Sets

---

- Set: contains **distinct** elements without order.
  - Pascal supports sets of any discrete type, and provides union, intersection, and difference operations:

```
var A, B, C : set of char;  
D, E : set of weekday;  
...  
A := B + C;  
(* union; A := {x | x is in B or x is in C} *)  
A := B * C;  
(* intersection; A := {x | x is in B and x is in C} *)  
A := B - C;  
(* difference; A := {x | x is in B and x is not in C}*)
```
- A set can be implemented as a hash table, where the keys are the set elements, and there are no values associated with the keys.

# Various Data Types

## Lists

---

- A list is a discrete sequence of elements that allows duplicates.
  - Python lists: Array-lists are efficient for element extraction, doubling-resize