# Data Types

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

November 13, 2020

# Outline

1. **Scalar Types**
   - Built-in Types
   - User-Defined Ordinal Types
2. **Composite Types**
   - Array Types
   - String Types
   - Record Types
   - Union Types
   - Set Types
   - Pointer and Reference Types
   - Recursive Type
3. **Type Checking**
4. **Case Study: Python**

# Introduction

- A data type is
  - a homogeneous collection of values and
  - a set of operations which manipulate these values
- Uses of type system:
  - Conceptual organization
  - Error detection
  - Implementation

# Type System

A type system consists of:

- The set of predefined types
- The mechanisms to define a new type
- The mechanisms for the control of types:
    - Type equivalence
    - Type compatibility
    - Type inference
- The specification which type constraints are statically or dynamically checked

# Scalar Types

Scalar Types are

- **atomic**
- used to compose another types
- sometimes supported directly by hardware
- booleans, characters, integers, floating-point, fixed-point, complex, void, enumerations, intervals,...

Skip Scalar Types

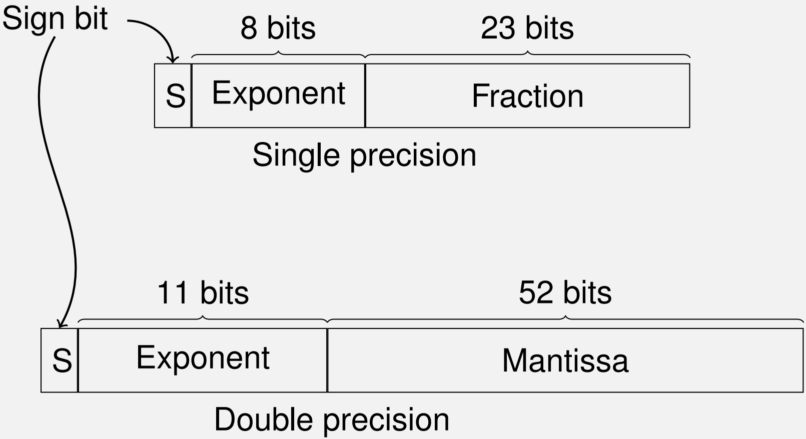
# Integer

- Languages may support several sizes of integer
  - Java's signed integer sizes: byte, short, int, long
- Some languages include unsigned integers
- Supported directly by hardware: a string of bits
- To represent negative numbers: two's complement

# Floating-Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types (e.g., float and double)
- Precision and range
- IEEE Floating-Point Standard 754

# IEEE-754


Sign bit
8 bits
23 bits
S
Exponent
Fraction
Single precision
11 bits
52 bits
S
Exponent
Mantissa
Double precision

# Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits
- *Advantage*: accuracy
- *Disadvantage*: limited range, wastes memory

# Boolean

- Simplest of all
- Range of values: two elements, one for “true” and one for “false”
- Could be implemented as bits, but often as bytes

# Character

- Stored as numeric codings
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode
  - Includes characters from most natural languages
  - Originally used in Java
  - C# and JavaScript also support Unicode

# User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Examples of primitive ordinal types in Java
  - integer
  - char
  - boolean

# Enumeration Types

- All possible values, which are named constants, are provided in the definition
- C# example
  *enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};*
  *days myDay = Mon, yourDay = Tue;*
- Design issues:
  - Is an enumeration constant allowed to appear in more than one type definition?
  - Are enumeration values coerced to integer?
  - Are any other types coerced to an enumeration type?

# Enumeration Type (2)

- **Readability**
  - no need to code a color as a number
- **Reliability**
  - operations (don't allow colors to be added)
  - No enumeration variable can be assigned a value outside its defined range
  - Better support for enumeration than C++: enumeration type variables are not coerced into integer types
- Implemented as integers

# Subrange Type

- an ordered contiguous subsequence of an ordinal type
  *type pos = 0 .. MAXINT;*
- Subrange types behave as their parent types; can be used as *for* variables and array indices
  *type sv = array[1 .. 50] of string;*
- Subrange types are the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

# Composite Types

- An object in composite type contains many components which can be accessed individually
- component's type may be the same (homogeneous) or different (heterogeneous)
- the number of components may be fixed or changed
- there may be operations on structured-type object or its components
- there may be component insertion/removal operations
- there may be creation/destruction operations

# Array Types

- Collection of homogeneous data elements
- Each element is identified by its position relative to the first element and referenced using subscript expression
  *array_name (index expression list)* $\rightarrow$ *an element*
  - What type are legal for subscripts?
    - Pascal, Ada: any ordinal type (integer, boolean, char, enumeration)
    - Others: subrange of integers
  - Are subscripting expressions range checked?
    - Most contemporary languages do not specify range checking but Java, ML, C#
    - Unusual case: Perl

Skip Array Type

# Subscript Binding and Array Categories

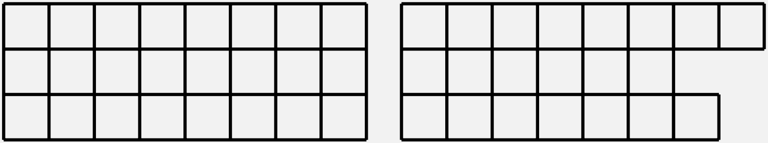
- Static
  *static int x[10];*
- Fixed Stack-dynamic
  *int x[10]; //inside a function*
- Stack-dynamic
  *cin »n;*
  *int x[n];*
- Fixed Heap-dynamic
  *int[] x = new int[10];*
- Heap-dynamic
  *cin »n;*
  *int[] x = new int[n];*

# Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example
    *int list [] = {4, 5, 7, 83}*
  - Character strings in C and C++
    *char name [] = "freddie";*
  - Arrays of strings in C and C++
    *char *names [] = {"Bob", "Jake", "Joe"};*
  - Java initialization of String objects
    *String[] names = {"Bob", "Jake", "Joe"};*

- C, C++, Java, C#: jagged arrays
  *myArray[3][7]*
- Fortran, Ada, C#: rectangular array
  *myArray[3,7]*

rectangular jagged

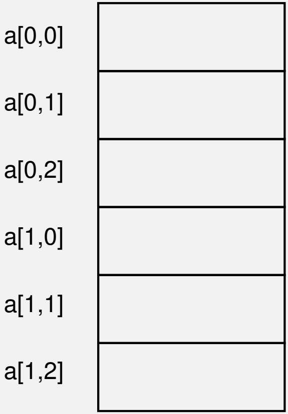


# Slices

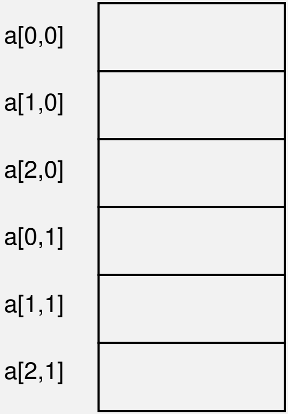
- A slice is some substructure of an array; nothing more than a referencing mechanism
- Slices are only useful in languages that have array operations
- E.g. Python
  *vector = [2, 4, 6, 8, 10, 12, 14, 16]*
  *mat = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]*
  *vector[3:6], mat[1], mat[0][0:2]*

- Access function maps subscript expressions to an address in the array
- Single-dimensioned: list of adjacent memory cells
- Access function for single-dimensioned arrays: **address(list[k]) = address(list[lower_bound]) + ((k-lower_bound) * element_size)**

# Accessing Two-dimensional Arrays




Row-major order
used in most languages

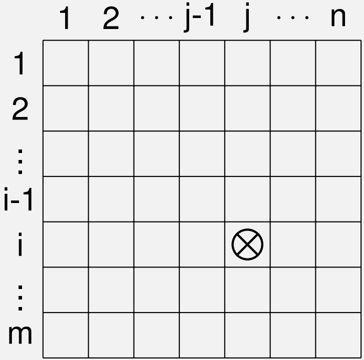



Column-major order
used in Fortran

Row-major order:

**Location (a[i,j]) = $\alpha$ + (((i - row_lb) * n) + (j - col_lb)) * E**

where $\alpha$ is address of a[row_lb,col_lb] and E is element size



# Compile-time Descriptors

| |
|---|
| Array |
| Element type |
| Index type |
| Index lower bound |
| Index upper bound |
| Address |

Single dimensional array

| |
|---|
| Multidimensional array |
| Element type |
| Index type |
| Number of dimensions |
| Index range 1 |
| ⋮ |
| Index range n |
| Address |

Multi-dimensional array

# Associative Arrays

- An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*
  For example,
  dt = [("name","John");("age","28");("address","1 John st.")]
  dt["name"] $\Rightarrow$ "John"
  dt["address"] $\Rightarrow$ "1 John st."
  - User defined keys must be stored
- Similar to Map in Scala
- Design issues: What is the form of references to elements

## String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Should the length of strings be static or dynamic?
- Typical operations
  - Assignment
  - Comparison (=, >, etc.)
  - Concatenation
  - Substring reference
  - Pattern matching (regular expression)

Skip String Type

- **Static**: String length is fixed at compiling time
  - Python, Java String class
  - compile-time descriptor
- **Limited Dynamic**: String length may be changed but less than a limit
  - C, C++
  - run-time descriptor
- **Dynamic**: String length may be changed without any limit
  - Perl, JavaScript
  - run-time descriptor; linked list

Ada supports all three string length options

# Descriptor

| |
|---|
| Static string |
| String length |
| Address |

Compile-time descriptor
for static length strings

| |
|---|
| Limited dynamic string |
| Maximum length |
| Current length |
| Address |

Run-time descriptor
for limited dynamic length
strings

# Record Types

- A record:
  - heterogeneous aggregate of data elements
  - individual elements are identified by names
- Popular in most languages, OO languages use objects as records
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed

▸ Skip Record Type

# Definition of Records in Ada

Record structures are indicated in an orthogonal way

***type*** *Emp_Name_Type* ***is record***

  *First: String (1..20);*

  *Mid: String (1..10);*

  *Last: String (1..20);*

***end record****;*

***type*** *Emp_Rec_Type* ***is record***

  *Emp_Name: Emp_Name_Type;*

  *Hourly_Rate: Float;*

***end record****;*

*Emp_Rec: Emp_Rec_Type;*

# References to Records

- Notation:
  - Dot-notation: *Emp_Rec.Emp_Name.Mid*
  - Keyword-based:
    *Mid OF Emp_Name OF Emp_Rec*
- Format:
  - **Fully qualified references**: include all record names
  - **Elliptical references**: may leave out some record names as long as reference is unambiguous
    *Mid, Mid OF Emp_Name, Mid OF Emp_Rec*

# Operations in Records

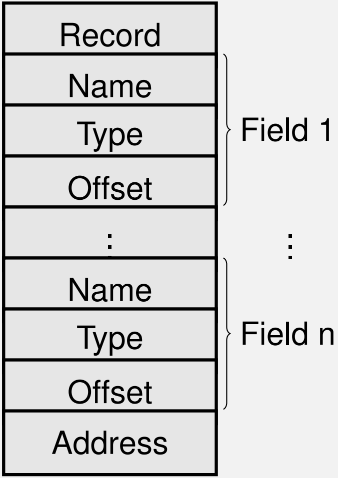
- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides MOVE CORRESPONDING
  Copies fields which have the same name

# Evaluation

- Straight forward and safe design
- Comparison of arrays and records

| Arrays | Records |
| --- | --- |
| homogenous | heterogeneous |
| elements are processed in the same way | elements are processed in different way |
| dynamic subscripting | static subscripting |

# Implementation of Record Type



## b-byte aligned

A b-byte aligned object has an address that is a multiple of b bytes.

## Example

1. A **char** (one byte) will be 1-byte aligned.
2. A **short** (two bytes) will be 2-byte aligned.
3. A **int** (four bytes) will be 4-byte aligned.
4. A **long** (four bytes) will be 4-byte aligned.
5. A **float** (four bytes) will be 4-byte aligned.

## Padding

when a structure member is

- followed by a member with a larger alignment requirement, or
- at the end of the structure to make the structure size be multiple of the biggest member size.

```
struct MyStruct {
    char data1;
    int data2;
    char data3;
    short data4;
    char data5;
};
```

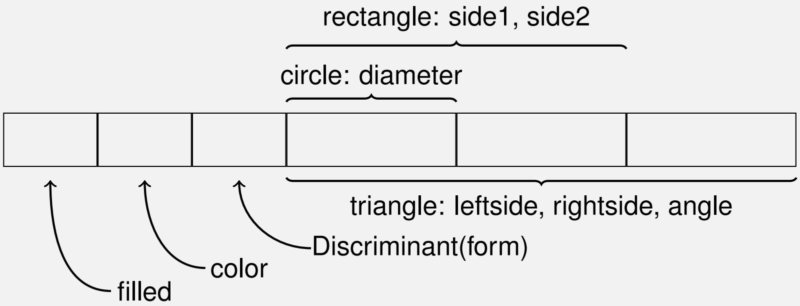
What is the size of the above struct?

- A union is a type whose variables are allowed to store different type values at different times during execution

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Color: Colors;
    case Form is
        when Circle => Diameter: Float;
        when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
    end case;
end record;
```

Skip Union Type

# Ada Union Type Illustrated

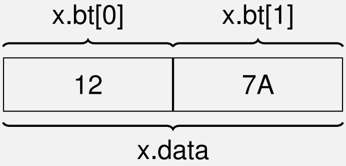


# Design issues

- Should type checking be required?
- Discriminated vs. Free Union
  - Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called **free union**
  - Type checking of unions require that each union include a type indicator called a **discriminant**
    - Supported by Ada
- Should unions be embedded in records?

# Example

```
union {
    int data;
    char bt[2];
} x;
x.data = 0x7A12;
cout << x.bt[0] << endl; //18
cout << x.bt[1] << endl; //122
```



# Evaluation of Unions

- Potentially unsafe construct in some languages
  - Do not allow type checking
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language

```
x: set of 1..10;
y: set of char;
```

- represent the concept of set
- has operators: membership, union, intersection, different,...
- implemented by bit chain or hash table.

# Pointer Types

int *ptr;

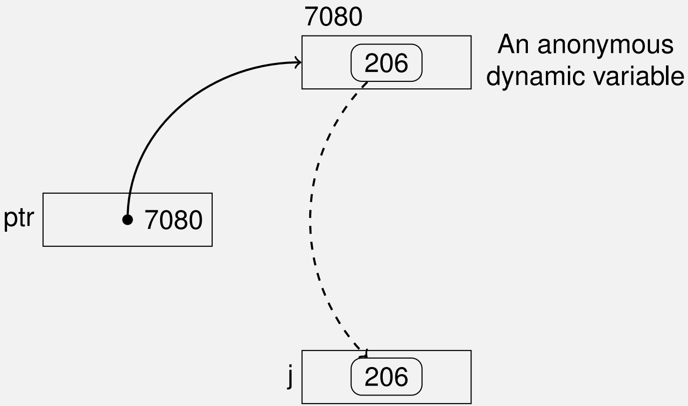
- A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil*
- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
  - A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)

Skip Pointer Type

# Pointer Operations

- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable's value to some useful address
  int *p,*q;
  p = q
- Dereferencing yields the value stored at the location represented by the pointer's value
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit operation via *
    j = *ptr
    sets j to the value located at ptr

# Pointer Dereferencing Illustrated




The dereferencing operation j = *ptr

# Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been de-allocated
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)

```
int *ptr;
int count, init;
...
ptr = &init;
count = *ptr;
```

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when it was allocated
- Used for dynamic storage management and addressing

- Pointer arithmetic is possible

```
int list [10]; int *ptr; ptr = list;
*(ptr + 1)
*(ptr + index)
ptr[index]
```

- Explicit dereferencing and address-of operators
- Domain type need not be fixed (void *)
- void * can point to any type and can be type checked (cannot be de-referenced)

# Pointer Operations

- Pointer points to a record in C/C++
  - Explicit: (*p).name
  - Implicit: p -> name
- Management of heap use explicit allocation
  - C: function **malloc**
  - C++: **new** and **delete** operators

# Design Issues of Pointers

- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

```
int A;
int &rA = A;
A = 1;
cout << rA << endl;
rA++;
cout << A << endl
```

- Pointers refer to an address, references refer to object or value
- C++ includes a special kind of pointer type called a reference type that is used primarily for formal parameters
- Java extends C++'s reference variables and allows them to replace pointers entirely
- C# includes both the references of Java and the pointers of C++

# References vs. Pointers in C++

| Reference Type | Pointer |
|---|---|
| `int A;`<br>`int& rA = A;` | `int A;`<br>`int* pA = &A;` |
| rA $\Rightarrow$ A | *pA $\Rightarrow$ A |
| N/A | pA++ |
| cannot reseated | pA = &B |
| cannot be null | pA = null |
| cannot be uninitialized | int* pA |

# Evaluation of Pointers

- Dangling pointers and garbage are big problems
- Pointers are like goto's–they widen the range of cells that can be accessed by a variable
- Essential in some kinds of programming applications, e.g. device drivers
- Using references provide some of the flexibility and capabilities of pointers, without the hazards

# Representations of Pointers

- Most computers use single values
- Intel microprocessors use segment and offset

# Dangling Pointer Problem

- Tombstone: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but set to nil
  - Costly in time and space
- Locks-and-keys: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

# Blank

# Recursive Type

A value of a *recursive type* can contain a (reference to) value of the same type.

```
type int_list = {int val;
                 int_list next;}
{3,{43,{-1,{6,null}}}}

type char_tree = {char val;
                  char_tree left;
                  char_tree right;}
{A,{B,{C,null,null},{D,{E,null,null},null}},
   {F,null,null}}
```

```
type char_btree =
        Tree of char * char_btree * char_btree
      | Null

Tree('A',Tree('B',Tree('C',Null,Null),
                  Tree('D',
                           Tree('E',Null,Null),
                           Null)),
          Tree('F',Null,Null))

type 'a btree = Tree of 'a * 'a btree * 'a btree
              | Null

Tree(4,Tree(3,Null,Null),Tree(6,Null,Null))
```

# Type Expression: Motivation Example

```
x: array [1..10] of record
    a: array [5..10] of integer;
    b: record
        c: real;
        d: array[1..3] of real;
    end;
    d: string[3];
end;
```

- A **basic type** is a type expression.
  boolean, char, integer, float, void, subrange.
- A **type name** is a type expression.
- A **type constructor** applied to type expressions is a type expression. Including:
  - Arrays: array(I,T) where I: index type, T:element type
  - Products: T1 $\times$ T2
  - Records: record((name1 $\times$ T1) $\times$ (name2 $\times$ T2) $\times \ldots$)
  - Pointers: pointer(T)
  - Functions: T1 $\rightarrow$ T2
- A **type variable** is a type expression.

# Example

- int $\Rightarrow$ int
- typedef int siso; $\Rightarrow$ siso
- int t[10]; $\Rightarrow$ array(0..9,int)
- int foo(int a,float b) $\Rightarrow$ (int $\times$ float) $\rightarrow$ int
- struct int a;int b $\Rightarrow$ record((a $\times$ int) $\times$ (b $\times$ int))
- int *p $\Rightarrow$ pointer(int)
- template <class T> struct vd T a; T b[3];
  $\Rightarrow$ record((a $\times$ T) $\times$ (b $\times$ array(0..2,T)))

## Definition

**Type checking** is the activity of ensuring that a program respects the rules imposed by the type system

- **Static type checking** is performed in **compiling** time. It is often applied for static type binding languages.
- **Dynamic type checking** is performed in **running** time. It is often applied for
  - dynamic type binding languages
  - Some features in static type binding language that cannot be type checked during compiling time.

# Type Inference

## Definition

Type inference is the ability of a compiler to deduce type information of program unit.

## Example on Scala

def add(x:Int) = x + 1

Return type of function add is inferred to be Int

## Mechanism

- Assign type (built-in or variable type)to leaf nodes in AST.
- Generate type constraints in each internal node in AST.
- Resolve these type constraints

# Type Equivalence

- an operand of one type can be substituted for one of the other type without coercion.
- Two approaches:
  - Equivalence by name: same type name

```
type Celsius = Float;
type Fahrenheit = Float;
```

  - Structural equivalence: same structure

```
type A = record                type B = record
    field1: integer;               field1: integer;
    field2: real;                  field2: real;
end                            end
```

# Static Type Checking for Structural Equivalence

```
function sequiv(Type s,Type t):boolean
begin
    if (s and t are the same basic type) then
          return true;
    else if (s = array(s1,s2) and t = array(t1,t2) then
          return sequiv(s1,t1) and sequiv(s2,t2);
    else if (s = s1 × s2 and t = t1 × t2) then
          return sequiv(s1,t1) and sequiv(s2,t2);
    else if (s = pointer(s1) and t = pointer(t1)) then
          return sequiv(s1,t1);
    else if (s = s1 → s2 and t = t1 → t2) then
          return sequiv(s1,t1);
    else
           return false;
```

**Definition**

Type T is compatible with type S if a value of type T is permitted in any context where a value of type S is admissible

Example, *int* and *float*

A type T is compatible with type S when:

- T is equivalence to S
- Values of T form a subset of values of S
- All operations on S are permitted on T
- Values of T correspond *in a canonical fashion* to values of S. (*int* and *float*)
- Values of T can transform to some values of S.

**Definition**

Type conversion is conversing a value of this type to a value of another type

- Implicit conversion - coercion
- Explicit conversion - cast

## Definition

- *Monomorphic*: any language object has a unique type
- *Polymorphic*: the same object can have more than one type

Example, +: $int \times int \rightarrow int$ or $float \times float \rightarrow float$

## Kind of Polymorphism

- *Ad hoc polymorphism* - Overloading
- *Universal Polymorphism*
  - Parametric polymorphism (swap(T& x,T& y))
  - Subtyping polymorphism (in OOP)

# Example of Parametric Polymorphism

```
template<typename T>
void swap (T& x, T& y){
    T tmp = x;
    x = y;
    y = tmp;
}
int a = 5, b = 3;
swap(a,b);
cout << a << "␣" << b << endl;
```

# Example of Subtyping Polymorphism

```
class Polygon
    public:
        virtual float getArea() = 0;
class Rectangle : public Polygon
    public:
        float getArea()
            return height * width;
    private:
        float height,width;
class Triangle : public Polygon
    public:
        float getArea()
            float p = (a + b + c) / 2;
            return sqrt(p*(p-a)*(p-b)*(p-c));
    private:
        float a,b,c;
Shape *s;
s = (...) ? new Rectangle(3,4) : new Triangle(3,4,5);
s->getArea();
```

- Scalar Data Types
  - **Number**: int (normal, octal-0o, hexa-0x, bin-0b), float
  - **Boolean**: bool
- Composite Data Types
  - **Number**: complex (yj) => real(), imag()
  - **String**: str
  - **Sequence**: list, tuple, range
  - **Mapping**: dict
  - **Set**: set, frozenset

# Immutable vs. Mutable Data Type

- Immutable Data Types: cannot modify their contents
  - Number (int,float,complex)
  - Boolean (bool)
  - String (str)
  - Sequence (tuple, range)
  - Set (frozenset)
- Mutable Data Types: able to modify their contents
  - Sequence (list)
  - Mapping (dict)
  - Set (set)

# Python Lists

- Like array but more flexible
  - Lists are ordered

    ```
    [1,2,3] == [1,3,2]
    ```
    => False
  - Lists can contain any arbitrary objects.

    ```
    x = [1,'a',2.3,[4,[6],5]]
    ```
  - List elements can be accessed by index.

    ```
    x[0]            => 1
    x[1:3]          => ['a', 2.3]
    x[3][1][0]      => 6
    ```
  - Lists are mutable and dynamic.

    ```
    x[0] = 2            => x -> [2,'a',2.3,[4,[6],5],'c']
    x[1:4] = [4,5,6]    => x -> [2,4,5,6,'c']
    x.append(12)        => x -> [2,4,5,6,'c',12]
    del x[0]            => x -> [4,5,6,'c',12]
    ```

# List Comprehension

- Motivation: create a list from another list

```
lst = []
for ele in <another list>:
   lst.append(<expression>)
```

```
lst1=[[1,2,3],[4,5],[6,7,8,9]]
lst2 = []
for ele in lst1:
   lst2.append(sum(ele))
```

- Syntax

  lst = [<expression> **for** ele **in** <another **list**> (**if** <condition>)?]

- Mapping

  lst2 = [**sum**(ele) **for** ele **in** lst1] => [6,9,30]

- Mapping with filtering

  lst2 = [**sum**(ele) **for** ele **in** lst1 **if** **len**(ele) > 2] => [6,30]

- Nested

  lst2 = [ele **for** row **in** lst1 **for** ele **in** row] => [1,2,3,4,5,6,7,8,9]

x = [1,2,'a','c',4.3]

- Indexing: return an element
  - Start from 0
    x[0] => 1
  - Accept negative, where -1 is the last element
    x[-1] => 4.3
- Slicing: return a list
  [start? : last? (: step)?]
  - x[1:3] => [2,'a']
  - x[1:] => [2,'a','c',4.3]
  - x[:-1] => [1,2,'a','c']
  - x[:] => [1,2,'a','c',4.3]
  - x[1:4:2] =>[2,'c']
  - x[::2] => [1,'a',4.3]
  - x[::-1] => [4.3,'c','a',2,1]

- Tuples are like Lists except for two following properties:
  - Enclosed by ( ) instead of [ ]
  - **immutable**
- Ranges are **immutable** sequences of integers
  - generated by **range(start?,stop,step?)**

    ```
    list(range(5))  => [0,1,2,3,4]
    list(range(1,5))  => [1,2,3,4]
    list(range(1,5,2))  => [1,3]
    ```

  - accessed by index

    ```
    range(1,5)[3]   => 4
    ```

  - used in for loop

    ```
    for x in range(1,5,2):
      print(x)
    ```

- is mapping from from **keys** to **values** like **struct** but:
  - **keys** => any hashable type, **values** => any type

```
x = {'fname': 'Teo', 'age': 50,
       3: {2.78: 'float',True:'bool'}}
```

  - access by **key** enclosed in [ ]
  - mutable and dynamic

```
x['fname'] = 'Ty'   => replace 'Teo' by 'Ty'
x[3][True]          => 'bool'
x[4] = 'four'       => add new component
del x[3]            => remove component 3
x                   => {'fname':'Ty','age':50,4:'four'}
```

  - Operators and Built-in functions: read `https://realpython.com/python-dicts/`

# Sets and Frozenset

- Sets are
  - unordered with unique elements

    {1,2,3} == {1,3,2} => True
    {1,2,3} == {1,2,2,3,1} => True

  - heterogeneous type

    ```
    x = {1, 'abc', True}
    ```

  - dynamic but elements of sets are immutable

    x.add(4) => {1,'abc',True,4}
    x.remove('abc') => {1,True,4}

  - Operators and Methods: read
    `https://realpython.com/python-sets/`
- Frozen sets are like sets except they are immutable

  ```
  x = frozenset({1, 'abc', True})
  ```

# Summary [1]

- Type system is mainly used to error detection
- Primitive type
- Structure type
- Type checking

# References

Maurizio Gabbrielli and Simone Martini, Programming Languages: Principles and Paradigms, Chapter 8, Springer, 2010.