

Data Types

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

09, 2014

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

- 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

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

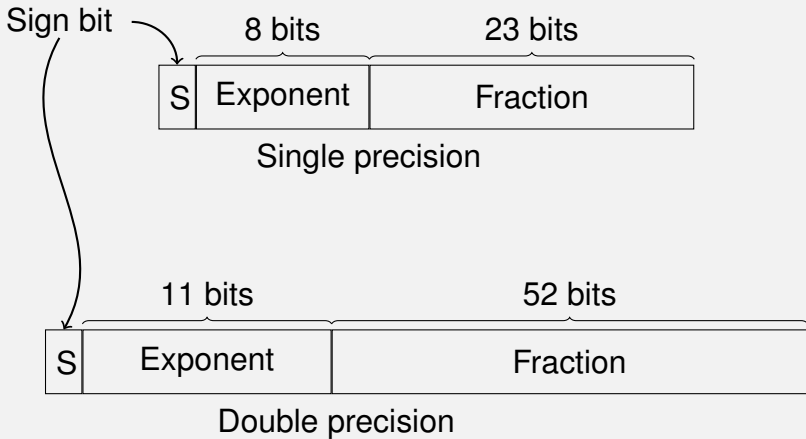
Scalar Types are

- **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

- 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**

- 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



- 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

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

- 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

- 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

- 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?

- **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

- 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

- 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

- 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) → 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

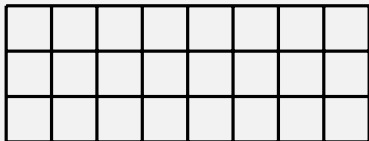
- Static
static int x[10];
- Fixed Stack-dynamic
int x[10]; //inside a function
- Stack-dynamic
cin »n;
int x[n];
- Fixed Heap-dynamic
int[] x = new int[10];
- Heap-dynamic
cin »n;
int[] x = new int[n];

- 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"};

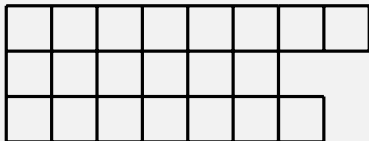
Rectangular and Jagged Arrays

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

rectangular



jagged



- 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

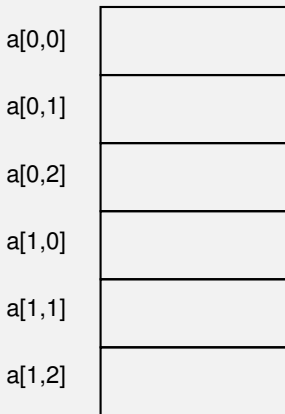
vector = [2, 4, 6, 8, 10, 12, 14, 16]

mat = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]

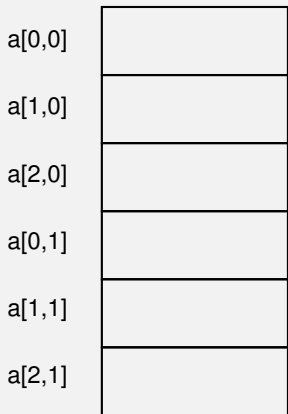
vector[3:6], mat[1], mat[0][0:2]

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

Accessing Two-dimensional Arrays



Row-major order
used in most languages



Column-major order
used in Fortran

Row-major order:

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

where α is address of $a[\text{row_lb}, \text{col_lb}]$ and E is element size

	1	2	...	j-1	j	...	n
1							
2							
\vdots							
i-1							
i					\otimes		
\vdots							
m							

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

- 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"] ⇒ "John"
```

```
dt["address"] ⇒ "1 John st."
```

- User defined keys must be stored
- Similar to Map in Scala
- Design issues: What is the form of references to elements

- 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

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

- 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

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;
```

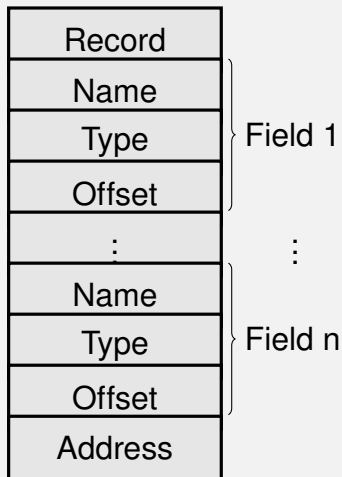
```
Emp_Rec: Emp_Rec_Type;
```

- 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

- 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

- 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



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.

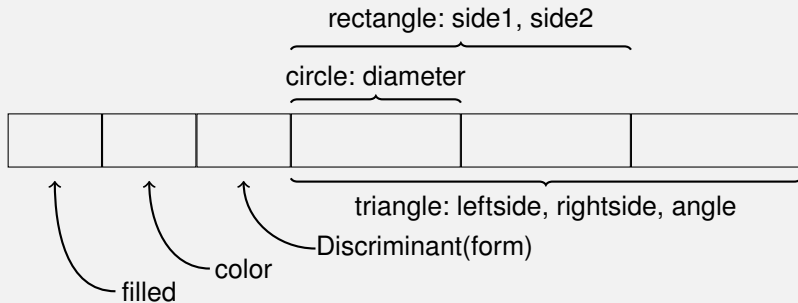
```
struct MyStruct {  
    char data1;  
    int data2;  
    char data3;  
    short data4;  
    char data5;  
};
```

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

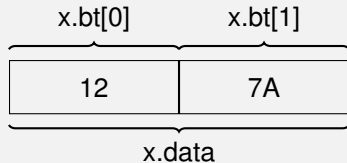
```
type Shape is (Circle, Triangle, Rectangle);  
type Colors is (Red, Green, Blue);  
type Figure (Form: Shape) is record  
    Filled: Boolean;  
    Color: Colors;  
    case Form is  
        when Circle => Diameter: Float;  
        when Triangle =>  
            Leftside, Rightside: Integer; 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

Ada Union Type Illustrated



- 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?


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



- 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.

```
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

- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable's value to some useful address

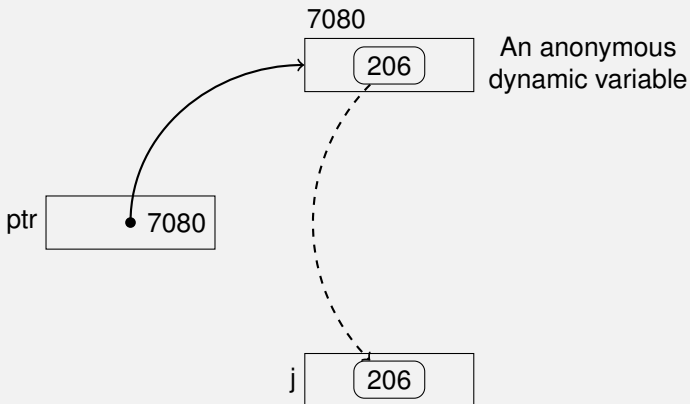
```
int *p,*q;
```

```
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



The dereferencing operation $j = *ptr$

- 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 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

- 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
<code>int A;</code> <code>int& rA = A;</code>	<code>int A;</code> <code>int* pA = &A;</code>
$rA \Rightarrow A$	$*pA \Rightarrow A$
N/A	<code>pA++</code>
cannot be reseated	<code>pA = &B</code>
cannot be null	<code>pA = null</code>
cannot be uninitialized	<code>int* pA</code>

- 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

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

- 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

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))
```

```
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: $\text{array}(I, T)$ where I : index type, T : element type
 - Products: $T_1 \times T_2$
 - Records: $\text{record}((\text{name1} \times T_1) \times (\text{name2} \times T_2) \times \dots)$
 - Pointers: $\text{pointer}(T)$
 - Functions: $T_1 \rightarrow T_2$
- A **type variable** is a type expression.

- `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.

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

- 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
```

```
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 ($\text{swap}(T\& x, T\& y)$)
 - Subtyping polymorphism (in OOP)

```
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();
```

```
abstract class Stack[A]  
  def push(x: A): Stack[A] =  
    new NonEmptyStack[A](x, this)  
  def isEmpty: Boolean  
  def top: A  
  def pop: Stack[A]  
class EmptyStack[A] extends Stack[A]  
  def isEmpty = true  
  def top = error("EmptyStack.top")  
  def pop = error("EmptyStack.pop")  
class NonEmptyStack[A](elem: A, rest: Stack[A])  
  extends Stack[A]  
  def isEmpty = false  
  def top = elem  
  def pop = rest  
val x = new EmptyStack[Int]  
val y = x.push(1).push(2)  
println(y.pop.top)
```



```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {  
    p.isEmpty ||  
    p.top == s.top && isPrefix[A](p.pop, s.pop)  
}
```

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



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