Types

A type is a set of values having predefined characteristics, e.g. integer, floating point, character, string, pointer.

- size in memory
- *value range (min – max)*

Types provide implicit context for operations so that the programmer does not have to specify that context explicitly, e.g.

- `a+b` Integer addition if a and b are of integer types. (concat if they are strings)
- `new MyType()` Heap is allocated without having to specify object size, and constructor is called automatically.

Types limit the set of operations that may be performed in a semantically valid program (it helps us program correctly)

e.g.

- Prevent adding char and struct
- Prevent passing a file as a parameter to a subroutine that expects an integer

High-level languages associate types with values to provide contextual information and error checking.

Common Types

Discrete types = มีหลายค่า (นับขึ้น นับลงได้)

- The domains to which they correspond are countable.
- There is the notion of predecessor and successor.
- E.g. integer, boolean, char, enumeration, subrange (เช่น 0 ถึง 10)

อยู่ใน

Scalar types = มีค่าที่แน่นอน

- They hold a single data item (single-valued types).
- E.g. discrete, real (ไม่เป็น discrete (นับขึ้นลงไม่ได้ เช่น 0.05 จะมาเป็น 0.051 ไม่ได้)), floating point number

Composite types = เอา data เล็กๆ มาเก็บรวมกัน

- Non-scalar types created by applying a type constructor to one or more simpler types.
- E.g. record (struct), array, string, set, pointer, list, file

(record (struct), array) คล้ายๆ Object

string ประกอบจาก char

Type System (1) = มีการนิยาม + เอาไปใช้งาน

Consists of

- A mechanism to define types and associate them with certain language constructs
  - *Mechanism: predefined types vs. composite types (having type constructors to build from simpler types)*
  - *Associated construct: named constant, variable, record field, parameter, subroutine, literal constant, complicated expression*

ให้คนนิยมใช้เอง

```
//C struct
typedef struct pnt {
  int x, y;
} Point;

Point point_new(int x, int y) { … }
```

```
(* Pascal array, subrange *)
type
   ch_array = array[char] of 1..26;
   test_score = 0..100;

var
   alphabet: ch_array;
   score: test_score;
```

test_score มีค่าตั้งแต่ 0 ถึง 100

นิยามตัวแปร

ชื่อ

type

```
//Java enumeration, class
public enum Day {
  SUNDAY, MONDAY, TUESDAY,
  WEDNESDAY,THURSDAY, FRIDAY,
  SATURDAY
}
public class EnumTest {
  Day day;
  … }
…
EnumTest firstDay;
```

สร้าง object ที่เป็น type EnumTest

Type System (2)

ต้องมี! type system ถึงจะสมบูรณ์

Consists of (cont.)

- **Type equivalence rules** to determine when the types of two values are the same. (เท่ากันเมื่อไหร่)
- **Type compatibility rules** to determine when a value of a given type can be used in a given context. (may need some auto conversion) (ใช้ใน context ที่ได้ไหม)
- **Type inference rules** to determine the type of an expression based on the types of its constituent parts or the surrounding context.

การรวม type เพื่อดูว่า type ของ result เป็นอย่างไร เช่น a + b*c → type ตอนจบจะเป็นอย่างไร

Type Checking = Type ตรงกันหรือไม่

Process of ensuring that a program obeys the language's type compatibility rules * (checking if an object of a certain type can be used in a certain context)

A violation of the rules is known as a type clash. = type ไม่ตรง

```
//c
int a;
a = "xyz";    //clash
```

Static vs. Dynamic Typing

Statically-typed language

- Type is bound to the variable, and type checking can be performed at compile time, e.g. Pascal, Java, C, C#

```
//Java
String s = "abcd";    //s will forever be a string
```

- In practice, most type checking can be performed at compile time and the rest at run time.

```
//C
int n;  int iarr[3];
…
iarr[n] = 5; //indexoutofbound at runtime
```

Dynamically-typed language

- Type is bound to the value, and types are checked at run time, e.g. Lisp, Perl, PHP, Python, Ruby

```
#Python
s = "abcd"      # s is a string
s = 123         # s is now an integer
p = s - 1
```

Type Equivalence Rules = ของ 2 สิ่งคือว่าเป็น type เดียวกันได้อย่างไร

In a language in which the user can define new types, there are two ways of defining type equivalence.

Structural equivalence → คือว่า a เป็น type เดียวกัน b ไหม โดยดู "โครงสร้างภายใน"

- Based on meaning behind the declarations
- Two types are the same if they consist of the same components.
- E.g. Algol-68, Modula-3, (to some extent) C, ML.

a = b
ถ้าโครงสร้าง "เหมือนกัน"
ต่อให้ชื่อ type ต่างกัน ก็เหมือนกันเป๊ะ

Name equivalence → ชื่อต้องเหมือนกันด้วย

ภาษาสมัยใหม่

- Based on declarations
- Each definition introduces a new type.
- More fashionable these days
- E.g. Java, C#, Pascal, Ada

Structural Equivalence (1)

Name equivalent จะสื่อว่าใช้แทนกันไม่ได้เลย

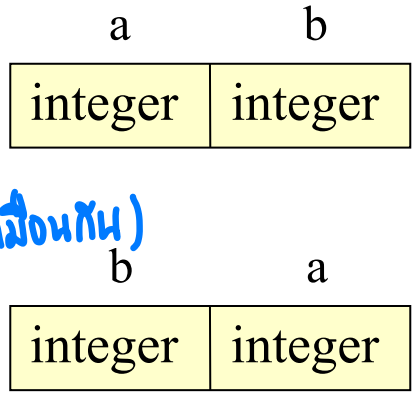
Structural equivalent → R1 กับ R2 แทนได้ (R3 ไม่ sure) เพราะ Structure เหมือนกัน

Expand the definitions of two types. If the same, they are equivalent.

But exact definition of structural equivalence varies from one language to another.

```
type R1 = record
  a, b : integer
end;
type R2 = record
  a : integer;
  b : integer
end;
type R3 =record
  b : integer;
  a : integer; //order differs
end;
```

ลำดับนิยามต่างกัน (แต่ Structure เหมือนกัน)



R1 and R2 are equivalent. What about R2 and R3?

Most languages say R2 and R3 are equivalent, ML says no.

Structural Equivalence (2)

Inability to distinguish between types that programmer may think of as distinct but which happen to have the same internal structure. Compiler with structural equivalence will accept this.

```
type student = record
  name, address : string
  age: integer
type school = record
  name, address : string
  age : integer
x : student; y : school;
…
x := y;
```

จาก structural equivalent ดีว่าใช้แทนกัน

แต่! ไม่ตรงตามวัตถุประสงค์ (อาจผิด logic sense)

name	address	age
string	string	integer

name	address	age
string	string	integer

```
//C structural equivalence for scalar types
typedef float celsius;
typedef float fahrenheit;
…
celsius c; fahrenheit f;
…
f = c; // does not check the logic
```

โครงสร้างข้างในเหมือนกัน

แทนที่ได้ ✓

Name Equivalence

If the programmer takes the effort to write two type definitions, then those are meant to represent different types.

```
//C
typedef struct b1 {
    char  title[20];
    char  author[20];
    int   book_id;
} Book1;
typedef struct b2 {
    char  title[20];
    char  author[20];
    int   book_id;
} Book2;
…
Book1 book1; Book2 book2;
…
book2 = book1;  //no match for operator =, operand types are Book2 and Book1
```

ข้างในเหมือน

And they cannot be casted!!!

! แต่ก็มีการ extends ต้องดูอีกที

error

(book 2) การ casting จะผ่านไม่ผ่าน ขึ้นอยู่กับภาษา

!!!

Exercise: Structural vs. Name Equivalence

```
//Java
class MyCard {
    public MyCard() { … }
    public int suit() { … }
    public int rank() { … }
    private int suitValue;
    private int rankValue;
}
class YourCard {
    public YourCard() { … }
    public int suit() { … }
    public int rank() { … }
    private int suitValue;
    private int rankValue;
}
class MyCardChild extends MyCard { … }
```

Given

MyCard mc;

What kind of type equivalence (structural or name) is used in each statement?
What happens to each statement when type checking is performed?

1. mc = new YourCard(); ……
2. mc = (MyCard) new MyCardChild(); ……
3. mc = new MyCardChild(); ……

Exercise: Structural vs. Name Equivalence

```
//Java
class MyCard {
    public MyCard() { … }
    public int suit() { … }
    public int rank() { … }
    private int suitValue;
    private int rankValue;
}
class YourCard {
    public YourCard() { … }
    public int suit() { … }
    public int rank() { … }
    private int suitValue;
    private int rankValue;
}
class MyCardChild extends MyCard { … }
```

Given

MyCard mc;

What kind of type equivalence (structural or name) is used in each statement?
What happens to each statement when type checking is performed?

1. mc = new YourCard(); ….name equivalent (Error เพราะชื่อไม่ตรงกัน)
2. mc = (MyCard) new MyCardChild(); name equivalent (ไม่มีปัญหาเพราะ MyCardChild extends มาจาก MyCard)
3. mc = new MyCardChild(); name equivalent (ไม่มีปัญหา)

ใช้ name equivalent

แม่ a = new ลูก ☑

ลูกมีทุกอย่างของแม่ (แปลงกลับมาเป็นแม่ก็ไม่มีปัญหา)

เป็น type MyCard ตรงกัน

ดึงที่มีทุกอย่างของ MyCard ที่เป็น Parent

Java
Superclass = (cast) subclass

ดังนั้น ตาม compatibility rule ของภาษา Java ที่ให้ super class สามารถรับการ assign จาก subclass ได้ ทำให้ไม่ว่าจะ นำตัวแปรชนิดคลาส Object ไปรับข้อมูลคลาสชนิดใดก็สามารถทำได้โดยไม่ต้องทำการ cast ใด ๆ เลย

1.compile error by name equivalence, type MyCard is expected for the assignment operator, even though YourCard has similar structure

2. mc = (MyCard) new MyCardChild(); //compile okay and run okay. By name equivalence, as the assignment operator expects the type MyCard, so the reference to MyCardChild object is cast to the reference to MyCard. And at runtime the type system knows how to convert the reference to MyCardChild object to a reference to MyCard object because, by inheritance or extends, the structure of object MyCardchild is compatible with the structure of object MyCard as MyCardchild has the attributes and methods that MyCard object has. So the cast is successful and mc sees a MyCardChild object as a MyCard object and references to it.

3. mc = new MyCardChild(); //compile okay and run okay. You can omit the cast (MyCard) in statement 2. Java uses loose name equivalence between superclass and subclass.

Type Conversion and Cast (1)

In a statically-typed language, there are many contexts in which values of a specific type are expected, e.g.

- `a = expression` (expression is expected to be of the same type as a)
- `a+b` (a and b are expected to be either integer or float)
- `foo(arg1, arg2, …, argN)` (arguments are expected to be of the types declared in foo's header)

If the programmer wishes to use a value of one type in a context that expects another, he or she will need to specify an **explicit type conversion** (or **type cast**) to enforce type equivalence.

Variables can be cast into other types, but they do not get converted. You just read them assuming they are another type.

Type Conversion and Cast (2)

ถ้าเป็น structural equivalent
ให้คนเขียนภาษาตัดสินใจได้เลยว่าจะให้แปลงได้/ไม่ได้
โดยไม่ต้องมีอะไรเพิ่ม

Depending on the types involved, conversion may or may not require code to be executed at run time. There are three principal cases:

1. If two types are **structurally equivalent** (same low-level representations and set of values) but the language uses name equivalence, no code will need to be executed at run time.

```
type student = record
  name, address : string
  age: integer
type school = record
  name, address : string
  age : integer
x : student; y : school;
…
x := y;  //error

X := (student)y;   //compile ok now but may not run
                   //depending on the language.
                   //no code needed, just the language decision.
```

✓

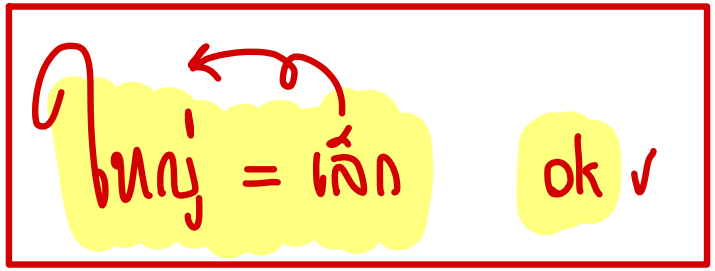
ขึ้นกับคนเขียนระบบ

2. The types have different sets of values, but **intersecting values** are represented in the same way (e.g. one type is a subrange of the other, one type is two's complement signed integers and the other is unsigned).
   - If the provided type has some values that the expected type does not, code must be executed at run time to check if the current value is valid in the expected type.

```
Typedef test_score 0..100;
…
int i;
test_score j;
…
…
i = j; //ok.

j = i; //compile error

j = (test_score)i; //compile ok
                   // if value of i is in range ->run ok
                   // if value of i is not in range -> runtime error
                   // Need code to run for this range check!
```



3. The types have different low-level representations but a **correspondence among their values** can be defined (e.g. integer to floating-point, floating-point rounded to integer). Most processors provide a machine instruction for this.

```
…
double i;
int j;
…
…
i = j; //ok. Machine instruction provides auto conversion.

j = i; //compile error

j = (int)i; //compile ok
            // run ok -> machine instruction does the conversion.
             //          precision is lost though.
```

Exercise: Type Conversion and Cast

กรณีที่ 1 cast โดยไม่ต้องดำเนินการใด ๆ เพิ่มเติม เพราะโครงสร้างข้อมูลเหมือนกัน และภาษาโปรแกรมนั้นที่ยินยอมให้ทำได้

กรณีที่ 2 cast โดยทำการตรวจสอบความถูกต้องของค่าก่อนในตอน run-time ว่าสามารถทำได้หรือไม่

กรณีที่ 3 convert โดยใช้ machine instruction (ไม่ต้องใส่ cast โปรแกรมทำให้)

Which of the three cases of run-time code for type checking applies to the following?

1 = Structural Equivalent → โครงสร้างข้างในเหมือนกัน

2 = Intersecting value → มีค่าบางค่าทับกัน

3 = Correspondence among their value (เช่น Integer ↔ floating point) → มีการสูญเสียความแม่นยำได้

```
--Ada
n : integer;                --assume 32 bits
r : long_float;             --assume IEEE double-precision
t : test_score;             --type test_score is new integer range 0..100;
c : celsius_temp;           --type celsius_temp is new integer;
...
1. t := test_score(n);      ..... 2 .....
2. n := integer(t);         ..... 1 ..... (Ok เพราะ t เล็กกว่า n) .....
3. r := long_float(n);      ..... 3 .....
4. n := integer(r);         ..... 3 + 2 .....
5. n := integer(c);         ..... 1 ..... (Ok เพราะ structure เหมือนกัน) .....
6. c := celsius_temp(n);    ..... 1 ..... ไม่ได้มีข้อกำหนดเรื่อง range .....
```

type ที่ต้องการจะ cast — ตัวแปร

ต้อง check ตาม case ไหน

(รายละเอียด pdf หน้า 11)

<3 เป็น 3 ด้วยกรณีใหญ่กว่า / ตัวมันเปลี่ยนได้>

ค่าเกิน range

Type Compatibility Rules

Most languages do not require type equivalence in every context. They say that a **value's type must be compatible with that of the context** in which it appears.

Whenever a language allows a value of one type to be used in a context that expects another, the language implementation must perform an **automatic implicit conversion** to the expected type. This is a **type coercion**.

Like explicit type conversion, coercion may require run-time code to perform a dynamic semantic check or to convert between low-level representations.

Type Coercion

Coercion allows types to be mixed without explicit indication of intent from programmer.

```
//C
short int s;          //16 bits
unsigned long int l; //32 bits
char c;               //8 bits
float f;              //32 bits, IEEE single-precision
double d;             //64 bits, IEEE double-precision
…            //something may be interpreted differently,
             //or some precision may be lost
s = l;
l = s;
s = c;
f = l;
d = f;
f = d;
```

ทางภาษาจะไม่ยอม

```
//C
//array and pointer can be mixed

int n;
int *a;
int b[10];
a = b;
n = a[3];
```

pointer to int

ระบบconvertให้อัตโนมัติ

Universal Reference Type

superset ของ type ทั้งหมด / ทั้งหมดของภาษานั้น

A **void*** pointer cannot be dereferenced unless it is cast to another type.

Several languages provide a universal reference type (**compatible with any data value**), e.g.

C, C++	void *
Clu	any
Modula-2	address
Java	Object
C#	object

```
//C
int x;  char y;  void *z;
z = &x;   z = &y;
```

```
//Java
Object a;
Cat c = new Cat(); Dog d = new Dog();
a = c;
a = d;
```

superclass ของทุกคลาส

ได้หมด

แต่เวลาใช้จริงต้อง casting ให้เป็น cat/dog ก่อน ไม่งั้นจะใช้ไม่ได้

```
Cat c = (Cat) a;
```

แต่อาจ error เพราะ a ไม่ใช่ cat จริงๆ

ใช้ casting

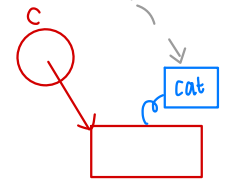
Arbitrary l-values (locations) can be assigned into an object of a universal reference type.

Assignment of a universal reference **back into** the object of a **particular reference type** requires the object to be self-descriptive and include a type tag in the representation of each object. (Normally, this will need programmer to do casting!)

Such type tags are common in OO languages.

เป็นการเช็คค่า typetag ที่ผูกไว้กับ object

```
if( a instanceOf Dog )
```

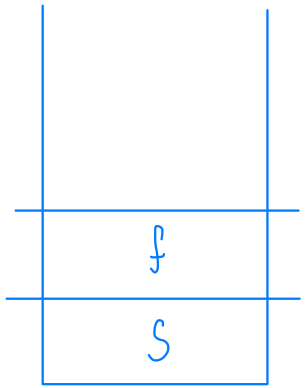


Exercise: Universal Reference Type

What happens to the last statement below? How can you ensure the safety of universal-to-specific assignment?

```
import java.util.* //library containing Stack container class
…
Stack my_stack = new Stack();
String s = “Hi, Mom”;
foo f = new foo();
…
Object aString = my_stack.push(s);
Object aFoo = my_stack.push(f);
…
s = my_stack.pop();
```

```
class Stack {
  Object push(Object item) {…}
  Object pop() {…}
  …
}
```

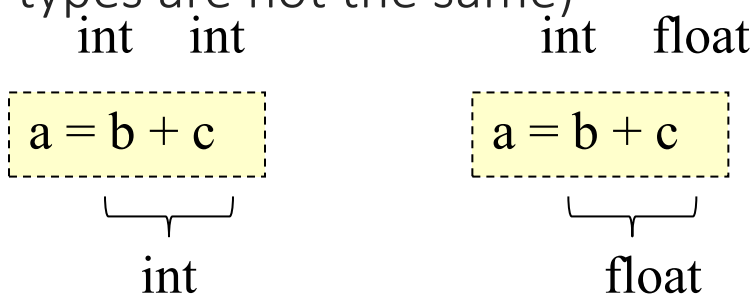


Type Inference Rules

Sometimes type of a whole expression needs to be inferred from the types of subexpressions (and possibly the type expected by the surrounding context) for type checking.

In many cases, the answer is easy, e.g.

- Result of an arithmetic operator usually has the same type as the operands (possible after coercing one of them if their types are not the same)



- Result of an assignment operator has the same type as the left-hand side.
- Result of a function call is of the type declared in the function's header.

In other cases, the answer is not obvious, e.g.

```
(* Pascal *)
type Atype = 0..20;
     Btype = 10..20;
var a: Atype; b: Btype; c: integer;
c = a + b;     (* subrange base type (integer), not the type sometype = 10..40 *)
```