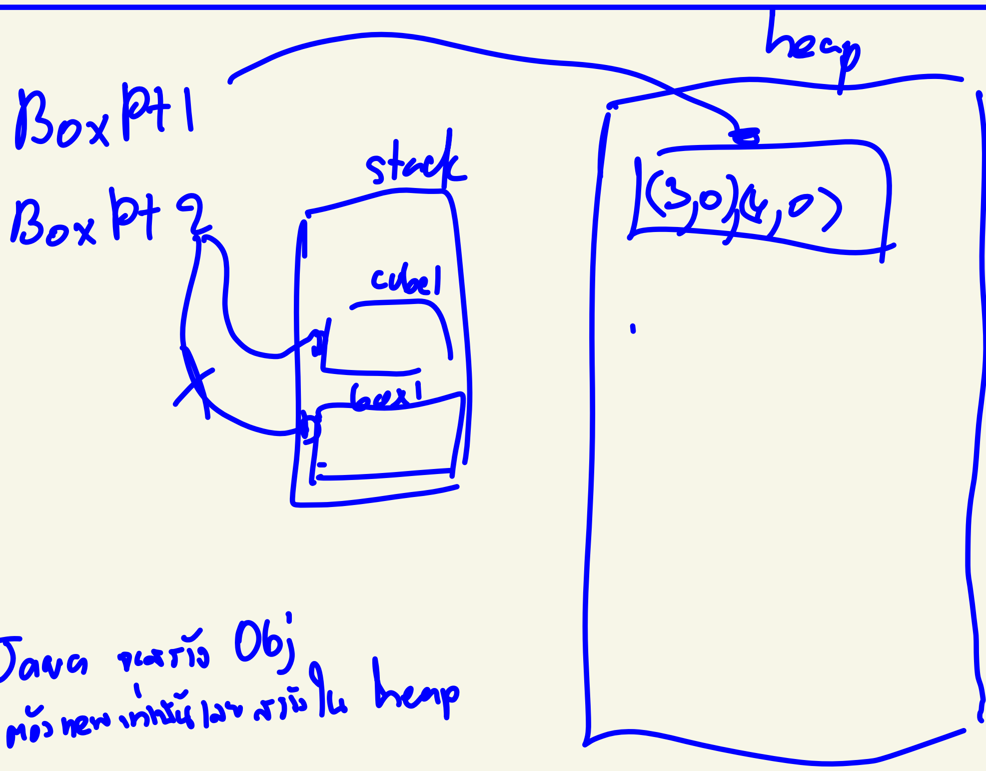
Static scope

ดูตัวแปรที่ parameter > นอก scope function

สามารถแก้ไขค่าได้แต่จะแก้ตัวที่ถูกเรียกใช้

Dynamic scope

ดูตัวแปรที่ประกาศล่าสุดตรงนอก scope ที่ function ถูกเรียก



Java จะสร้าง Obj
ต้อง new เท่านั้นเลย สร้างใน heap

Cpp สร้าง Obj
new → ใน heap (ต้องใช้ new เท่านั้น) (ถ้า new แล้วต้อง delete เท่านั้น ไม่งั้น memory leak)
ไม่ new → ใน stack
box (1,2,3) (ไม่ต้องเป็นการสร้าง ใน heap)

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

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

Scalar types

real number

- They hold a single data item (single-valued types).
- E.g. discrete, real, int, boolean, enumeration
- some people disagree because:
- *They say it must have magnitude that can be compared to other values of the same type.*
- *Date-time can be considered scalar!*

2016-10-06 15:55:55

arr, string

Composite types

- 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

Type System (1)

การนิยาม type

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*

Start index

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

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

นิยาม type

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

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

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

Type System (2) 3องค์ประกอบ type-system

Consists of (cont.)

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

Type Checking

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**. It causes compile error!

```
//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

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

type

Structural equivalence ดูโครงสร้างภายใน

a = b

- 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.

Name equivalence ดูชื่อ type

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

Structural Equivalence (1)

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;
```

a	b
integer	integer

b	a
integer	integer

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.

Just an example. This is no specific language.

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

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
```

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
```

And they cannot be casted!!!
(Pascal allows this though)!

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 → X......
2. mc = (MyCard) new MyCardChild(); ......name ✓......
3. mc = new MyCardChild(); ......✓......

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)

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 in some language (not Java)
                  // but may not run depending on the language.
                  //no code needed, just the language decision.
```

In pascal, the program runs

```pascal
begin
    v1.name   := 'Jojo';
    v1.address := '190/1';
    v1.age := 20;

    v2.name := 'Satit';
    v2.address := 'Patumwan';
    v2.age := 100;

    // v1 := v2;  //This does not compile
    v1 := student(v2);
    writeln ('v1 name : ', v1.name);
    writeln ('v1 address : ', v1.address);
    writeln ('v1 age : ', v1.age);
```

```
Free Pascal Compiler version
Copyright (c) 1993-2021 by F
Target OS: Linux for x86-64
Compiling main.pas
Linking a.out
42 lines compiled, 0.1 sec
v1 name : Satit
v1 address : Patumwan
v1 age : 100
```

In Java, it does not even compile!

```java
public class Student {
    int age;
    String name;
}
```

```java
public class School {
    int age;
    String name;
}
```

```java
public class TestStudentSChool {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student s = new Student();
        School sc = (School)s;
    }

}
```

Cannot cast

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!
```

Pascal allows incorrect value!

```
Program HelloWorld(output);
type
   test_score = 0 .. 100;
var
  i: integer;
  j: test_score;
begin
    i := 130;
    j := 0;

    //i := j; //ok.

    //j = i; //compile error

    j := test_score(i); //compile ok
    writeln ('j value : ', j);
end.
```

```
Linking a.out
/usr/bin/ld.bfd: warning: link.res contain
26 lines compiled, 0.1 sec
j value : 130
```

?

C++, Java, C# do not have
Such subrange

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.

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

j = i; //compile error

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

Pascal

```pascal
Program HelloWorld(output);

var
  i: integer;
  j: double;
begin
    i := 130;
    j := 150;

    //i := j; //compile error.

    //j = i; // compile error

    j := double(i); //compile ok

    //i := integer(j);  //compile error

    writeln ('j value : ', j);
    writeln ('i value : ', i);
end.
```

```
Compiling main.pas
Linking a.out
29 lines compiled, 0.1 sec
j value :  1.30000000000000000E+002
i value : 130


...Program finished with exit code 0
```

Exercise: Type Conversion and Cast

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

```
--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); ......if n in range 0...100 it ok!......
2. n := integer(t); ......ok......
3. r := long_float(n); ......int → float ok......
4. n := integer(r); ......float → int ok if in range......
5. n := integer(c); ......always ok!......
6. c := celsius_temp(n); ......always ok!......

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, 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; Java ไม่ได้
l = s; ✓
s = c; ?
f = l; ✓
d = f; ✓
f = d; x
```

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

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

Classที่เป็น Universal

Universal Reference Type

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

z (Cat) a;

```
//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;
```

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.

if(a instance of Dog)

chivava is instanceof Dog

but can use a.getClass().getName()

Exercise: Universal Reference Type

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

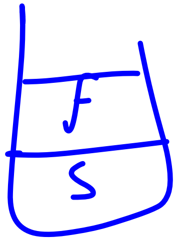
import java.util.* //library containing Stack container class

…

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);

… Object

s = my_stack.pop();

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

ใช้ → instance of แล้ว cast ก่อน

Object มาใส่ string ไม่ได้

error type ไง

Type Inference Rules กฎการรวม type

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)

$$a = \underbrace{\overset{\text{int}}{b} + \overset{\text{int}}{c}}_{\text{int}} \qquad\qquad a = \underbrace{\overset{\text{int}}{b} + \overset{\text{float}}{c}}_{\text{float}}$$

- 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 *)
```

integer