15 Types - Structural Equivalence & Name Equivalence

Thursday, November 12, 2020 8:41 AM

C

```
typedef double celsius;
typedef double fahr;

celsius c =  100.0;
fahr f;

f = c;
```

Go

```
type celsius float64
type fahr float64

var c celsius = 100.0
var f fahr

f = c
```

f = CtoF(c)

Java

```
public class Fahr {
    private double temp;
   …
}

public class Celsius{
   private double temp;
   …
}

Celsius c = new Celsius(100.0);
Fahr f;

f = c;
```

Pascal

```
type A = array [1..10] of integer;
type B = array [0..9] of integer;

var a : A;
    b : B;
```

a := max(a);
b := max(b);

Are **a** and **b** equivalent?
Why does it matter?

In Pascal the index range + size of an array is part of its type.

function max (x : array [1..10] of integer) : integer;

Derived Types

Ada

```
type celsius = new real;
type fahr = new real;
```

```
type celsius = real;
type fahr = real;
```

Modula2

open arrays

```
FUNCTION max (x : array of integer): integer;
```

x[?] x[0..n-1]

len(x)

```
int max(int a, int b){
}

double max(double a, double b){
}

int max(int x[], int n) {
}

int max(int[] x) {
}
```

x.length

15 Type Compatibility, Type Conversion, Type Coercion

Thursday, November 12, 2020 8:41 AM

Type Compatibility:
Programming languages do not require type equivalence but strive for defining type compatibility.

Two types are **compatible** if:

1. The two types are equivalent
2. One is a subtype of the other
3. Both are arrays with same element types (and index).

Also, there is a situation where the two types may not be structurally equivalent but there is a way to **convert** one type into another.

```
int x;
double y;
x = 3;
y = y + x;
```

```
y = x;
x = y;   X
```

Type Conversion
- Explicitly (cast): `x = (int) y;`
- Implicit: `y = x;` ⟵ Type Coercion; `y = y + x/y;`

```
int n = (int)(Math.random() * 10);
```

[0..9] ← (int) ← [0.0 .. 1.0)

n = 0 !!

```
System.out.println("12" + 3);
```

"3" → "123"

123

→ "12" + 3 → "123"

→ "12" − 3 → ?

"12" → 12

? = 9

Javascript

Type Inference

x := 5.3

15 Control Abstraction - Subroutines

Thursday, November 12, 2020 8:58 AM

* Lab#4 is posted
* Assignment #4 is posted

Control Abstraction *vs* Data Abstraction

Subroutines:

- **Functions** - return (a) value
- **Procedures** - do not return any value

Functions

```
double area (double radius) {
    return Math.PI * radius * radius;
} // area()
```

Java

```
def area(radius):
    return math.pi * radius * radius
```

Procedure

```
void drawCircle(double x, double y, double r) {
    . . .
    return;
    . . .
    return;
} // drawCircle()
```

formal parameters

Vocabulary and Design Considerations

- **Caller, function call, invocation**

```
double a = area(r);
```

actual parameters

area (r1+r2)

- **Signature**

```
double area (double);
```

area)

- **Arguments/actual parameters, formal parameters**
- **Return**
  - primitive type only? ✓
  - Can be composite type? ✓
  - Can be a function? ✓
  - Can return 0 or 1 value? ✓
  - Can return multiple values? ✓

```
return <expression>
```

- **Parameter Association**
  - **Positional**
  - **Named/Keyword Parameters**
  - **Mixed Association**

x y r

```
drawCircle(x1, y1, r1);
drawCircle(x=x1, y=y1, r=r1)
drawCircle(r=r1, y=y1, x=x1)
drawCircle(x1, y1, r=r1)
drawCircle(x1, y=y1, r)
```

  - **Default Parameters**

```
def convert(n, base=2):
    . . .
```

def convert(n, base=2):

```
binary = convert(12)
octal = convert(12, 8)
hexadecimal = convert(12, base=16)
```

00001100

14

C

  - **Variable number of arguments**

printf(*string*, *v1*, *v2*, *v3*, ...);

15 Subroutines

Thursday, November 12, 2020 12:09 PM

Variable number of arguments

C

```
int max(int a, …) {
   va_list args;
   …
} // max()
```

va_list

max(a)
max(x,y)
max(x,y,z)

Java

```
int max(int a, int… ns) {
    int m = a;
    for (int x : ns)
        if (x > m)
            m = x;
    return m;
} // max()
```

Python

```
def max(a, *argv):
    m = a
    for x in argv:
        if x > m:
            m = x
    return m
```

GO

```
func max(a int, b int) int {
    if a > b {
        return a
    } else {
        return b
    }
} // max()
```

GO

```
func max(a int, ns …int) int {
    m := a
    for _, x := range xs {
        if x > m {
            m = x
        }
    }
    return m
} // max()
```

```
func max(a int, ns …int) (m int) {
    m = a
    for _, x := range xs {
        if x > m {
            m = x
        }
    }
    return m
} // max()
```

variadic functions

for i, x := range xs {