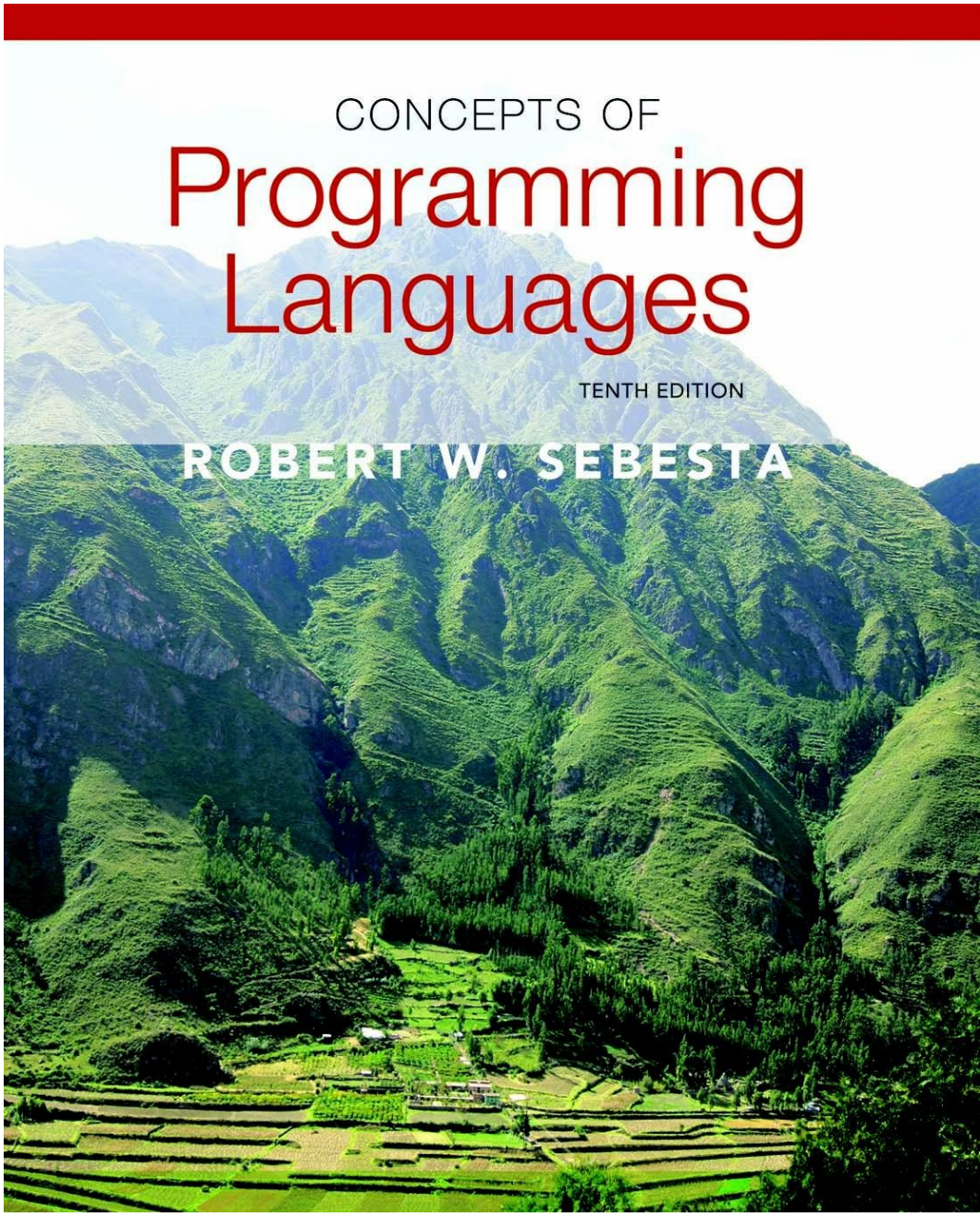
Ders9

Data Types
Type Checking



Chapter 6 Topics

- Type Checking
- Strong Typing
- Type Equivalence
- Theory and Data Types

Type Checking

One motivation of introducing types is to increase safety. There are several kinds of errors made in writing programs:
- Syntax: Wrong usage of the language
- Semantics: Wrong usage of the language
- Logic: the program does not meet its specification.

- We are focusing on type errors which are semantic errors.
- Example: for a given function or procedure invocation, we check whether an actual parameter correctly instantiates a formal parameter and we call it a type error if the function is applied to a parameter of the wrong type.
- A program without any type error is called type safe or type secure.

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
  - This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if type errors are always detected
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Static Type Checking vs Dynamic Type Checking

Error checking can be done at two different times:
- Static checking happens before runtime.
- Dynamic checking requires the program to be executed.

• Evaluation: Static checking is preferable for two reasons
  - efficiency
  - completeness

Note: static checking does not exclude run-time errors!

• Static checking is sometimes called compile-time checking. But sometimes static checks are also done during link time, when modules are imported.

Static Type Checking vs Dynamic Type Checking

Static typing means that programs are checked before being executed, and a program might be rejected before it starts.

Dynamic typing means that the types of values are checked during execution, and a poorly typed operation might cause the program to halt or otherwise signal an error at run time. A primary reason for static typing is to rule out programs that might have such "dynamic type errors".

Strong Typing vs Weak Typing

- A type system is called strong if all type errors are found.
- A type system is called weak if it is not strong.

Strong Typing vs Static Typing

- Strong typing and static typing are different.
- Static typing refers to the time at which checking is done.
- Strong typing refers to the completeness of the checking.

Strong Typing

Language examples:

- C and C++ are not: parameter type checking can be avoided; unions are not type checked
- Ada is, almost (`UNCHECKED CONVERSION` is loophole)

  (Java and C# are similar to Ada)

Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)

- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Is Java Strongly Typed?

Yes and No! Most typing errors will be caugt by Java but because of implicit type conversions, Java is not as strongly typed as some other languages. Java allows certain castings to happen implicitly. In all numeric types an up-cast or widening cast is implicit in java

Take a look at the following implicit cast:

```
int i = 10;
float f = i; // An implicit cast from 'int' to 'float' : up cast or widening cast
```

Weakly Typed: Javascript

```
<html>
<head>
<script language="JavaScript">
var apples=1.43;
var oranges=2.33;
var pears=4.32;
var total=apples + oranges + pears;
var message="Your total is $";
var deliver= message + total +".";
document.write(deliver);
</script>
</head>
<body bgcolor="indianred">
</body>
</html>
```

Weak Typing

Some programmers refer to a language as "weakly typed" if simple operations do not behave in a way that they would expect. For example, consider the following program:

x = "5" + 6

Different languages will assign a different value to 'x':

- One language might convert 6 to a string, and concatenate the two arguments to produce the string "56".
- Another language might convert "5" to a number, and add the two arguments to produce the number 11.
- Yet another language might convert the string "5" to a pointer representing where the string is stored within memory, and add 6 to that value to produce a semi-random address.

- And yet another language might simply **fail to compile this program**, saying that the two operands have incompatible type.

Languages that work like the first three examples have all been called "weakly typed" at various times, even though only one of them (the third) represents a safety violation.

Name Type Equivalence

- *Name type equivalence* means the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not equivalent with integer types
  - Formal parameters must be the same type as their corresponding actual parameters

Structure Type Equivalence

- *Structure type equivalence* means that two variables have equivalent types if their types have identical structures
- More flexible, but harder to implement

Examples:

But: what exactly does it mean to have the same structure?

- Examples:

– Should the following two types be equivalent?

```
type foo = record
    a,b : integer
end;

type foo = record
    a: integer;
    b: integer;
end;
```

Examples:

But: what exactly does it mean to have the same structure?
• Examples:
– Should the following two types be equivalent?

```
type student = record
      name, address: string;
      age: integer;
end;

type school = record
      name, address : string;
      age: integer;
end;

x: student;
y: school;
...
x := y;
```

Type Equivalence (continued)

- Consider the problem of two structured types:
  - Are two record types equivalent if they are structurally the same but use different field names?
  - Are two array types equivalent if they are the same except that the subscripts are different? (e.g. `[1..10]` and `[0..9]`)
  - Are two enumeration types equivalent if their components are spelled differently?
  - With structural type equivalence, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

What if two types are not equivalent?

- Overloading
- Casting
- Coercion

Operator Overloading

Overloading
- Definition: Overloading means an identifier refers to more than one object or operation in a given scope.
- What can be overloaded?
  Almost every language allows overloading for arithmetical operators. Ada and C++ allow overloading of functions and procedures.
  Constants can also be overloaded.

Operator Overloading

Alternative: coercion: Implicit type conversions
How exactly is overloading defined? Is changing the precedence of an operator the same as overloading?

- The process of disambiguating an overloaded identifier is called overload resolution. It happens during compile time.
- Evaluation:
follows mathematical convention, correctly used, overloading provides a great opportunity for information hiding,
- if misused, it can make programs unreadable.
- Note: Overloading and coercion do not always mix well.

Type Conversion

Type conversion is the operation that converts the argument of a function to the type the function expects.

```
int x;
float z;
...
x = x + z;
```

Without type conversion there is a type error in the last line (assume + is integer addition).

– The conversion can be done explicitly or implicitly. In the latter case we call it coercion.

Explicit Type Conversion

- Languages that support explicit conversion must provide special operators:
  - cast operator in C:

x = x + (int)z

  - cast operator in C++

x = x + (int)z

also: x = x + int(z)

- Issues to discuss:
  - What happens to the internal representation?
  - What happens to the value of a variable?
  - Is the semantics preserved?

Implicit Type Conversion: Coercion

Coercion (implicit conversion)

- Definition: Coercion is implicit type conversion.

– Coercion is like a symptom of weak-typing

- Coercion can be done both statically and dynamically.

– Well-known language for coercions: ALGOL.

– Often in arithmetical expressions, in particular in mixed mode expressions.

– When used in combination with overloading it makes programs hard to understand.

Type Conversion

- Issues to discuss:
  - What happens to the internal representation?
  - What happens to the value of a variable?
  - Is the semantics preserved?

Type Equivalence (continued)

Casting

```
#include <stdio.h>

int main(){
  int* i;
  *i = 65;
  char* ch;
  ch = (char*) i;
  printf (" *i: %d",*i);
  printf (" *ch: %c",*ch);

  return(0);
}
```

Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management