# 프로그래밍 언어론
# Lecture Note #09

조용주
ycho@smu.ac.kr

## 7.2 Type Checking

- In most statically typed languages, every definition of an object (constant, variable, subroutine, etc.) must specify the object's type
- We'll discuss type equivalence, type compatibility, and type inference
- Type compatibility is the one of most concern to programmers
  - determines when an object of a certain type can be used in a certain context
  - at a minimum, the object can be used if its type and the type expected by the context are equivalent (i.e., the same)
  - in many languages, however, compatibility is a looser relationship than equivalence: objects and contexts are often compatible even when their types are different

## 7.2 Type Checking

- our discussion of type compatibility will touch on the subject of
  - type conversion (also called casting)
    - changes a value of one type into a value of another
  - type coercion
    - performs a conversion automatically in certain contexts
  - nonconverting type casts
    - used sometimes in systems programming to interpret the bits of a value of one type as if they represented a value of some other type

## 7.2 Type Checking

- Whenever an expression is constructed from simpler subexpressions, the question arises: given the types of subexpressions (and possibly the type expected by the surrounding context), what is the type of the expression as a whole?
- This question  is answered by type inference
- Type inference plays an important role in ML, Miranda, and Haskell, in which almost all type annotations are optional, and will be inferred by the compiler when omitted

## 7.2.1 Type Equivalence

- In a language that allows the user to define new types, there are two principal ways of defining type equivalence
  - Structural equivalence
    - based on the content of type definitions
    - roughly speaking, two types are the same if they consist of the same components, put together in the same way
    - Algol 68, Modula-3, and (with various wrinkles) C and ML
  - Name equivalence
    - based on the lexical occurrence of type definitions
    - roughly speaking, each definition introduces a new type
    - Java, C#, standard Pascal and most Pascal descendants including Ada

## 7.2.1 Type Equivalence

- The exact definition of structural equivalence varies from one language to another
  - need to decide which potential difference between types are important and which may be considered unimportant
  - most people would probably agree that the format of a declaration should not matter
  - in a Pascal-like language with structural equivalence, the following would be the same

```
type R1 = record
    a, b : integer
end;
```

```
type R1 = record
    a : integer;
    b : integer
end;
```

  - but what about
    - ML says no
    - most languages say yes

```
type R1 = record
    b : integer;
    a : integer
end;
```

## 7.2.1 Type Equivalence

- consider the following arrays, again in a Pascal-like notation

```
type str = array [1..0] of char;
type str = array [0..9] of char;
```

  - the length of the array is the same in both cases, but the index values are different
  - are they equivalent?
  - most languages say no, but some (including Fortran and Ada) consider them compatible

## 7.2.1 Type Equivalence

- Structural equivalence is a straightforward but somewhat low-level, implementation-oriented way to think about types
  - its principal problem is an inability to distinguish between types that the programmer may think of as distinct, but which happen by coincidence to have the same internal structure
  - **structurally equivalent**

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

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

x : student
y : school
…
x := y -- is this an error?
```

## 7.2.1 Type Equivalence

- Name equivalence is based on the assumption that if the programmer goes to the effort of writing two type definitions, then those definitions are probably meant to represent different types
- In the example above, x and y will be considered to have different types under name equivalence: **name of the types used to declare x and y are different**

## 7.2.1 Type Equivalence

- Variants of Name Equivalence
  - one subtlety in the use of name equivalence arises in the simplest of type declarations:

```
type new_type = old_type;  /* Algol family */
typedef old_type new_type; /* C */
```

  - new_type is said to be an alias for old_type
    - should we treat them as two names for the same type or as names for two different types that happen to have the same internal structure?
  - when aliased types should probably not be the same

```
type celsius_temp = real;
     fahrenheit_temp = real;
 var c : celsius_temp;
     f : fahrenheit_temp;
…
f := c; (* this should probably be an error *)
```

## 7.2.1 Type Equivalence

- a language in which aliased types are considered distinct is said to have *strict name equivalence*
- a language in which aliased types are considered equivalent is said to have *loose name equivalence*
- Most Pascal-family language use loose name equivalence
- Ada achieves the best of both worlds by allowing the programmer to indicate whether an alias represents a *derived* type or a *subtype*
- a subtype is compatible with its base (parent) type
  - subtypes of the same base type are also compatible each other
- a derived type is incompatible

## 7.2.1 Type Equivalence

```
-- unsigned 16-bit integer
subtype mode_t is integer range 0..2**16-1;
…
type celsius_temp is new integer;
type fahrenheit_temp is new integer;
```

- under strict name equivalence, a declaration type A = B is considered a definition
- under loose name equivalence, it is merely a declaration
  - A shares the definition of B

```
type cell = … -- whatever
type alink = pointer to cell
type blink = alink
p, q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```

- strict name equivalence
  - p, q are the same
  - r, u are the same
- loose name equivalence
  - r, s, u are the same
- structural equivalence
  - all six variables are the same

## 7.2.1 Type Equivalence

- Type Conversion and Casts
  - in a language with static typing, there are many contexts in which values of a specific type are expected
    - in the statement

      ```
      a := expression
      ```

    - we expect the right-hand side to have the same type as a
    - in the expression

      ```
      a + b
      ```

    - the overloaded + symbol designates either integer or floating-point addition; we therefore expect either that a and b will both be the same

## 7.2.1 Type Equivalence

  - in a call to a subroutine,

    ```
    foo(arg1, arg2, …, argN)
    ```

  - we expect the types of the arguments to match those of the formal parameters, as declared in the subroutine's header

- suppose that we require in each of these cases that the types (expected and provided) be exactly the same
  - then the programmer need to specify an explicit *type conversion* (also sometimes called a type *cast*)
- depending on the types involved, the conversion may or may not require code to be executed at run time
- there are three principal cases

## 7.2.1 Type Equivalence

1. the types would be considered structurally equivalent, but the language uses name equivalence
   - the types employ the same low-level representation, and have the same set of values
   - the conversion is a purely conceptual operation
   - no code will need to be executed at run time

## 7.2.1 Type Equivalence

2. the types have different sets of values, but the intersecting values are represented in the same way
    - one type may be a subrange of the other
  - if the provided type has some values that the expected type does not, then code must be executed at run time to ensure that the current value is among those that are valid in the expected type
  - if check fails → a dynamic semantic error results
  - if check succeeds → the underlying representation of the value can be used, unchanged
  - some language implementations may allow the check to be disabled, resulting in faster but potentially unsafe code

## 7.2.1 Type Equivalence

3. the types have different low-level representations, but we can nonetheless define some sort of correspondence among their values

- a 32-bit integer → a floating point number
  - no loss of precision
  - most processors provide a machine instruction to effect this conversions
- a floating-point number → an integer
  - by rounding or truncating but fractional digits will be lost
  - the conversion will overflow for many exponent values
  - most processors provide a machine instruction
- conversion between different lengths of integers
  - by discarding or sign-extending high-order bytes

## 7.2.1 Type Equivalence

```
n : integer; -- assume 32 bits
r : long_float; -- assume IEEE double-precision
t : test_score; -- subrange 1..100
c : celsius_temp; -- new integer
…
t := test_score(n); -- run-time semantic check required
n : integer(t); -- no check req.; every test-score is an int
r := long_float(n); -- requires run-time conversion
n := integer(r); -- requires run-time conversion and check
n := integer(c); -- no run-time code required
c := celsius_temp(n); -- no r
```

- a type conversion in C is specified as following

```
r = (float) n; /*generates code for run-time conversion*/
n = (int) r; /*run-time conversion, with no overflow check*/
```

- C and its descendants do not by default perform run-time checks for arithmetic overflow on any operation, though such checks can be enabled if desired in C#

## 7.2.1 Type Equivalence

- Nonconverting Type Casts
  - occasionally, particularly in systems programs, one needs to change the type of a value *without* changing the underlying implementations
  - a change of type that does not alter the underlying bits is called a nonconverting type cast, or sometimes a type pun
  - in Ada, nonconverting casts can be effected using instances of a built-in generic subroutine called unchecked_conversion:

```
--assume 'float' has been declared to match IEEE single-precision
function cast_float_to_int is
    new unchecked_conversion(float, integer);
function cast_int_to_float is
    new unchecked_conversion(integer, float);
…
f := cast_int_to_float(n);
n := cast_float_to_int(f);
```

## 7.2.1 Type Equivalence

- C++ inherits the casting mechanism of C, but also provides a family of semantically cleaner alternatives
  - static_cast performs a type conversion
  - reinterpret_cast performs a nonconverting type cast
  - dynamic_cast allows programs that manipulate pointers of polymorphic types to perform assignments whose validity cannot be guaranteed statically, but can be checked at run time
  - syntax of each of these is that of a generic function:

```
double d = …
int n = static_cast<int>(d);
```

  - C-style type casts in C++ are defined in terms of const_cast, static_cast, and reinterpret_cast; the precise behavior depends on the source and target types

## 7.2.1 Type Equivalence

- Any nonconverting type cast constitutes a dangerous subversion can be difficult to find
- In a language with a weak type system such subversion can be difficult to find
- In a language with a strong type system, the use of explicit nonconverting type casts at least labels the dangerous points in the code, facilitating debugging if problems arise

## 7.2.2 Type Compatibility

- Most languages do not require equivalence of types in every context
- Instead, they merely say that a value's type must be *compatible* with that of the context in which it appears
  - in an assignment statement, the type of the right hand side must be compatible with that of the left-hand side
  - the types of the operands of + must both be compatible with some common type that supports addition (integers, real numbers, or perhaps strings or sets)
  - in a subroutine call, the types of any arguments passed into the subroutine must be compatible with the types of the corresponding formal parameters (opposite direction must be compatible, too)

## 7.2.2 Type Compatibility

- The definition of type compatibility varies greatly from language to language
  - Ada takes a relatively restrictive approach: type S is compatible with an expected type T if and only if
    - S and T are equivalent
    - one is a subtype of the other (or both are subtypes of the same base type)
      - T is a subtype of S
      - S is a subtype of T
    - both are arrays, with the same numbers and types of elements in each dimension

## 7.2.2 Type Compatibility

- Coercion
  - 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 → *type coercion*
    - may require run-time code to perform a dynamic semantic check
    - may require run-time code to convert between low-level representations
  - C, which has a relatively weak type system, performs quite a bit of coercion
  - Consider following declarations: assume 16, 32, 8, 32, 64 bits

```
short in s;
unsigned long int l;
char c;   /* signed or unsigned */
float f;  /* usually IEEE single-precision */
double d; /* usually IEEE double-precision */
```

## 7.2.2 Type Compatibility

- coercion may have a variety of effects when a variable of one type is assigned into another

```
/* l's low-order bits are interpreted as a signed number */
s = l;
/* s is sign-extended to the longer length, then its bits are
   interpreted as an unsigned number */
l = s;
/* c is either sign-extended or zero-extended to s's length;
   the result is then interpreted as a signed number */
s = c;
/* l is converted to floating-point. Since f has fewer
   significant bits, some precision may be lost */
f = l;
/* f is converted to the longer format; no precision is lost */
d = f;
/* d is converted to the shorter format; precision may be lost.
   If d's value cannot be represented in single-precision, the
   result is undefined, but NONT a dynamic semantic error */
f = d;
```

# 7.2.2 Type Compatibility

- coercion is a somewhat controversial subject in language design
  - it allows types to be mixed without an explicit indication of intent on the part of the programmer
  - it represents a significant weakening of type security

- Universal Reference Types
  - for systems programming, or to facilitate the writing of general-purpose *container* (*collection*) objects (lists, stacks, queues, sets, etc.) that hold references to other objects, several languages provide a *universal* reference type

|  |  |
|---|---|
| In C/C++ | → void * |
| Clu | → any |
| Modula-2 | → address |
| Modula-3 | → refany |
| Java | → Object |
| C# | → object |

## 7.2.2 Type Compatibility

- arbitrary l-values can be assigned into an object of universal reference type, with no concern about type safety
  - because the type of the object referred to by a universal reference is unknown, the compiler will not allow any operations to be performed on that object
- assignments back into objects of a particular reference type (e.g., a pointer to a programmer-specified record type) are a bit trickier, if type safety is to be maintained
  - we would not want a universal reference to a floating-point number to be assigned into a variable that is supposed to hold a reference to an integer
    - the subsequent operations on the "integer" would interpret the bits of the object incorrectly

## 7.2.2 Type Compatibility

- in object-oriented languages, the question of how to ensure the validity of a universal-to-specific assignment generalizes to the question of how to ensure the validity of any assignment in which the type of the object on left-hand side supports operations that the object on the right-hand size may not
- one way to ensure the safety of universal to specific assignments (or, in general less specific to more specific assignments) is to make objects self-descriptive – that is, to include in the representation of each object a *tag* that indicates its type
  - common in object-oriented languages
  - type tags in objects can consume a non-trivial amount of space, but allow the implementation to prevent the assignment

## 7.2.2 Type Compatibility

- in Java and C#, a universal to specific assignment requires a type cast, and will generate an exception if the universal reference does not refer to an object of the casted type

```
class A {
    public void print() {
        System.out.println("class A");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        Object o = new String("hello");
        a = (A) o; // ClassCastException
    }
}
```

## 7.2.2 Type Compatibility

- in C++, it uses a **dynamic_cast** operation

```
#include <iostream>
#include <string>

class A {
public:
  virtual void print() {
    std::cout << "class A" << std::endl;
  }
};

class B : public A {
};
```

## 7.2.2 Type Compatibility

```
class C {
public:
  virtual void print() {
    std::cout << "class C" << std::endl;
  }
};

void print(A* a) {
  if (a)
    a->print();
  else
    std::cout << "cast failed" << std::endl;
}
```

## 7.2.2 Type Compatibility

```
int main() {
  A* a;
  A* b = new B;
  C* c = new C;
  a = dynamic_cast<A*>(c);
  print(a);
  a = dynamic_cast<A*>(b);
  print(a);
  delete c;
  delete b;
}
```

## 7.2.2 Type Compatibility

- in early versions of Java and C#, programmers would often create container classes that held objects of the universal reference class (Object or object)
- this idiom has become less common with the introduction of generics (discussed in 7.3.1), but it is still occasionally used for containers that hold objects of more than on class
- when an object is removed from such a container, it must be assigned (with a type cast) into a variable of an appropriate class before anything interesting can be done with it

## 7.2.2 Type Compatibility

```
import java.util.*; // for using Stack
…
Stack myStack = new Stack();
String s = "Hi, Mom";
Foo f = new Foo(); // f is of user-defined class type Foo
…
myStack.push(s);
myStack.push(f); // can push any kind of object on a stack
…
s = (String) myStack.pop();
    // type cast is required. will generate exception at
    // run-time if element at top-of-stack is not a string
```

- in a language without type tags, the assignment of a universal reference into an object of a specific reference type cannot be checked, because objects are not self-descriptive: there is no way to identify their type at run time
- the programmer must resort to an type conversion

## 7.2.3 Type Inference

- We have seen how type checking ensures that the components of an expression (e.g., the arguments of a binary operator) have appropriate types
- But, what determines the type of the overall expression?
- In many cases the answer is easy
  - the result of an arithmetic operator usually has the same type as the operands (possibly after coercing one of them if their types were not the same)
  - the result of a comparison is usually Boolean
  - the result of a function call has the type declared in the function's header
  - the result of an assignment (in languages in which assignments are expressions) has the same type as the left-hand side

## 7.2.3 Type Inference

- In a few cases, however, the answer is not obvious
  - operations on subranges and composite objects, for example, do not necessarily preserve the types of the operands
  - we will explain these cases
- Subranges and Sets
  - for arithmetic operators, a simple example of inference

```
type Atype = 0..20;
     Btype = 10..20;
var a : Atype;
    b : Btype;
```

  - what is the type of a + b?
    - usually any arithmetic operation on a subrange has the subrange's base type—in this case, integer

## 7.2.3 Type Inference

    - The compiler may try to keep track of the min and max bounds of the subrange for the result to increase the performance, then check with bounds

- Declarations
  - Ada was among the first languages to make the index of a for loop a new, local variable, accessible only in the loop
    - the language implicitly assigned it the base type of the expressions provided as bounds for the loop
  - extensions of this idea appear in several more recent languages, including Scala, C# 3.0, C++11, Go, and Swift
    - allow the programmer to omit type information from a variable declaration when the intent of the declaration can be inferred from context

## 7.2.3 Type Inference

- in C#,

```
var i = 123; // equiv. to int i = 123;
var map = new Dictionary<string, int>(); // equiv. to
    // Dictionary<string, int> map
    //                  = new Dictionary<string, int>();
```

- here the type of the right-hand side of the assignment can be used to infer the variable's type, freeing us from the need to declare it explicitly

## 7.2.3 Type Inference

- the  convenience of inference increases with complex declarations

```
auto reduce = [](list<int> L, int f(int, int), int s) {
    for (auto e : L) {
        s = f(e, s);
    }
    return s;
  };
…
int sum = reduce(my_list, [](int a, int b) { return a +
   b; }, 0);
int product = reduce(my_list, [](int a, int b) { return
   a * b; }, 1);
```

- auto keyword allows us to omit the indication of type:

```
int (*reduce)(list<int>, int(*)(int, int), int) = …
 = [](list<int> L, int f(int, int), int s) {….
```

## 7.2.3 Type Inference

- C++ goes a step further, with a decltype keyword that can be used to match the type of any existing expression
  - the decltype keyword is particularly handy in templates, where it is sometimes impossible to provide an appropriate static type name

```
template <typename A, typename B>
…
    A a;   B b;
    decltype(a + b) sum;
```

## 7.3 Parametric Polymorphism

- Languages without compile-time type inference can provide implicit parametric polymorphism-like feature if we are willing to delay type checking until run time
- In Scheme, min function would be written like this

```
(define min (lambda (a b) (if (< a b) a b)))
```

- It makes no mention of types
  - The typical Scheme implementation employs an interpreter that examines the arguments to min and determines, at run time, whether they are mutually compatible and support a < operator
  - (min 123 456) evaluates to 123; (min 3.14159 2.71828) evaluates to 2.71828
  - (min "abc" "def") produces a run-time error when evaluated because the string comparison operator is named string<?, not <

## 7.3 Parametric Polymorphism

- Similar run-time checks for object-oriented languages were pioneered by Smalltalk, and appear in Objective C, Swift, Python, and Ruby, among others
  - In these languages, an object is assumed to have an acceptable type if it supports whatever method is currently being invoked
  - In Ruby, for example, **min** is  a predefined method supported by collection classes
    - assuming that the elements of collection **C** support a comparison (**<=>** operator), **C.min** will return the minimum element:

```
[5, 9, 3, 6].min           # 3 (array)
(2..10).min                # 2 (range)
["apple", "pear", "orange"].min  # "apple"
["apple", "pear", "orange"].min {
   |a,b| a.length <=> b.length
}                                # "pear"
```

- for the final call to **min**, an alternative definition of the comparison operator is provided

- This operational style of checking (an object has an acceptable type if it supports the requested method) is sometimes known as *duck typing*
  - its name is from the notion that "if it walks like a duck and quacks like a duck, then it must be a duck"

## 7.3.1 Generic Subroutines and Classes

- The disadvantage of polymorphism in Scheme, Smalltalk, Ruby, and the like is the need for run-time checking, which incurs nontrivial costs, and delays the reporting of errors
- For other compiled languages, *explicit* parametric polymorphism (otherwise known as *generics*) allows the programmer to specify type parameters when declaring a subroutine or class
  - compiler use these parameters when declaring a subroutine or class

## 7.3.1 Generic Subroutines and Classes

- Languages that provide generics include Ada, C++ (calls them *templates*), Eiffel, Java, C#, and Scala
- overloaded min function in Ada

```
function min(x, y : integer) return integer is
begin
  if x < y then return x;
  else return y;
  end if;
end min;

function min(x, y : long_float) return long_float is
begin
  if x < y then return x;
  else return y;
  end if;
end min;
```

## 7.3.1 Generic Subroutines and Classes

- generics in Ada

```
generic
  type T is private;
  with function "<"(x, y : T) return Boolean;
function min(x, y : T) return T;

function min(x, y : T) return T is
begin
  if x < y then return x;
  else return y;
  end if;
end min;

function int_min is new min(integer, "<");
function real_min is new min(long_float, "<");
function string_min is new min(string, "<");
function date_min is new min(date, date_precedes);
```

## 7.3.1 Generic Subroutines and Classes

- In an object-oriented language, generics are most often used to parameterize entire classes
  - among other things, such classes may serve as *containers*—data abstractions whose instances hold a collection of other objects, but whose operations are generally oblivious to the type of the objects they contain
    - examples
      - stack, queue, heap, set, dictionary (mapping) abstractions implemented as lists, arrays, tress, or hash tables
    - without generics, it is possible in some languages (C and early versions of Java and C#) to define a collection (e.g., queue) of references to arbitrary objects
      - use of such a collection requires type casts that abandon compile-time checking

- Generic array-based queue in C++

```
template<class item, int max_items = 100>
class queue {
    item items[max_items];
    int next_free, next_full, num_items;
public:
    queue() : next_free(0), next_full(0),
  num_items(0) {}
    bool enqueue(const item& it) {
        if (num_items == max_items) return false;
        ++num_items; items[next_free] = it;
        next_free = (next_free + 1) % max_items;
        return true;
    }
    bool dequeue(item* it) {
        if (num_items == 0) return false;
        --num_items; *it = items[next_full];
        next_full = (next_full + 1) % max_items;
        return true;
    }
};
```

```
queue<process> ready_list;
queue<int, 50> int_queue;
```

## 7.3.1 Generic Subroutines and Classes

- We can think of generic parameters as supporting compile-time customization, allowing the compiler to create an appropriate version of the parameterized subroutine or class
- In some languages—Java and C#, for example,--generic parameters must always be types
- Other languages are more general
  - in Ada and C++, for example, a generic can be parameterized by values as well
    - in C++, this value must be a compile-time constant
    - in Ada, which supports dynamic-size arrays, its evaluation can be delayed until elaboration time

## 7.3.1 Generic Subroutines and Classes

- Implementation Options
  - generics can be implemented several ways
  - in most implementations of Ada and C++ they are purely static mechanism: all the work required to create and use multiple instances of the generic code takes place at compile time
  - in the usual case, the compiler create a *separate copy* of the code for every instance
    - if several queues are instantiated with the same set of argument, then the compiler may share the code of the **enqueue** and **dequeue** routines among them
    - a clever compiler may arrange to share the code for a queue of integers with the code for a queue of floating-point numbers, if two types happen to have the same size, which is not required

## 7.3.1 Generic Subroutines and Classes

- Java, by contrast, guarantees that *all* instances of a given generic will share the same code at run time
  - in effect, if T is a generic type parameter in Java, then objects of class T are treated as instances of the standard base class Object
  - except that the programmer does not have to insert explicit casts to use them as objects of class T
    - the compiler guarantees, statically, that the elided casts will never fail
- C# plots an intermediate course
  - Like C++, it will create specialized implementations of a generic for different primitive or value types
  - Like Java, however, it requires that the generic code itself be demonstrably type safe, independent of the arguments provided in any particular instantiation

## 7.3.1 Generic Subroutines and Classes

- Generic Parameter Constraints
  - a generic is an abstraction, it is important that its interface (the header of its declaration) provide all the information that must be known by a user of the abstraction
    - several languages, including Ada, Java, C#, Scala, OCaml, and SML, attempt to enforce this rule by *constraining* generic parameters
    - specifically, they require that the operations permitted on a generic parameter type be explicitly declared
  - In Ada, the programmer can specify the operations of a generic type parameter by means of a trailing **`with`** clause
    - without the with clause, procedure sort would be unable to compare elements of A for ordering, because type T is private—it supports only assignment, testing for equality and inequality, and a few other standard attributes

## 7.3.1 Generic Subroutines and Classes

- Java and C# employ a particularly clean approach to constraints that exploits the ability of object-oriented types to *inherit* methods from a parent type or interface
  - it allows the Java or C# programmer to require that a generic parameter support a particular set of methods
  - in Java, we might declare and use a sorting routine as follows

```
public static <T extends Comparable<T>>
  void sort(T A[]) {
    …
    if (A[i].compareTo(A[j]) >= 0) …
    …
}
…
Integer[] myArray = new Integer[50];
…
sort(myArray);
```

## 7.3.1 Generic Subroutines and Classes

- C# syntax is similar:

```
static void sort<T>(T A[]) where T :
   Icomparable {
      …
      if (A[i].compareTo(A[j]) >= 0) …
      …
}
…
int[] myArray = new int[50];
sort(myArray);
```

- C#'s compiler is smart enough to recognize that int is a primitive type, and generates a customized implementation of sort, eliminating the need for java's Integer wrapper class, and producing faster code

## 7.3.1 Generic Subroutines and Classes

- A few languages forgo explicit constraints, but still check how parameters are used
  - in C++, for example, the header of a generic sorting routine can be extremely simple:

```
template<typename T>
void sort(T A[], int A_size) { …
```

    - no mention is made of the need for a comparison operator
    - the body of a generic can attempt to perform arbitrary operations on objects of a generic parameter type
    - if the generic is instantiated with a type that does not support that operation, the compiler will announce a static semantic error

## 7.3.1 Generic Subroutines and Classes

- unfortunately, because the header of the generic does not necessarily specify which operations will be required, it can be difficult for the programmer to predict whether a particular instantiation will cause an error message
- worse, in some cases the type provided in a particular instantiation may support an operation required by the generic's code, but that operation may not do "the right thing"
- for the above example, it would work with int and double, but for character strings, < operator will compare pointers, to see which referenced character has a lower address in memory
- to avoid surprises, the programmer may choose to emulate Java or C#, encapsulating the required methods in a abstract base class from which the type T may inherit

## 7.3.1 Generic Subroutines and Classes

- Implicit Instantiation
  - because a class is a type, one must generally create an instance of a generic class (i.e., an object) before the generic can be used
  - the declaration provides a natural place to provide generic arguments

```
queue<int,50>* myQueue = new queue<int,50>(); //C++
```

  - some languages (Ada among them) also require generic subroutines to be instantiated explicitly before they can be used:

```
procedure int_sort is new sort(integer,
                               int_array, "<");
…
int_sort(my_array);
```

## 7.3.1 Generic Subroutines and Classes

- other languages (C++, Java, C# among them) do not require this
  - instead they treat generic subroutines as a form of overloading
  - given the C++ sorting routine and the following objects:

```
int ints[10];
double reals[50];
string strings[30]; // string class has operator<
```

  - we can perform the following calls without instantiating anything explicitly:

```
sort(ints, 10);
sort(reals, 50);
sort(strings, 30);
```

  - in each case, the compiler will implicitly instantiate an appropriate version of the sort routine
  - Java and C# have similar conventions

## 7.3.2 Generics in C++, Java, and C#

- C++ is by far the most ambitious of the three
  - its templates are intended for almost any programming task that requires substantially similar but not identical copies of an abstraction
- Java and C# provide generics purely for the sake of polymorphism
  - Java's design was heavily influenced by the desire for backward compatibility, not only with existing versions of the language, but with existing virtual machines and libraries
  - the C# designers, though building on an existing language, was not constrained
  - they had been planning for generics from the outset, and were able to engineer substantial new support into the .NET virtual machine

## 7.4 Equality Testing and Assignment

- For simple, primitive data types such as integers, floating-point numbers, or characters, equality testing and assignment are relatively straightforward operations with obvious semantics and implementations (bit-wise comparison or copy)
- For more complicated or abstract data types, both semantic and implementation subtleties arise
- Consider for example the problem of comparing two character strings. Should the expression s == t determine whether s and t
  - are aliases for one another?
  - occupy storage that is bit-wise identical over its full length?
  - contain the same sequence of characters?
  - would appear the same if printed?

## 7.4 Equality Testing and Assignment

- The second of these tests is probably two low level to be of interest in most programs; it suggests the possibility that a comparison might fail because of garbage in currently unused portions of the space reserved for a string
- In many cases the definition of equality boils down to the distinction between l-values and r-values:
  - in the presence of references, should expressions be considered equal only if they refer to the same object, or also if the objects to which they refer are in some sense equal?
  - *shallow* comparison
    - refer to the same object
  - *deep* comparison
    - refer to equal object

## 7.4 Equality Testing and Assignment

- In imperative programing languages, assignment operations may also be deep or shallow
- Under a reference model of variables
  - shallow assignment
    - a = b will make a refer to the object to which b refers
  - deep assignment
    - will create a copy of the object to which b refers, and make a refer to the copy
- Under a value model of variables
  - a shallow assignment will copy the value of b into a, but if the value is a pointer (or a record containing pointers), then the objects to which the pointer(s) refer will not be copied

## 7.4 Equality Testing and Assignment

- Most programming languages employ both shallow comparisons and shallow assignment
  - a few (notably Python and the various dialects of Lisp and ML) provide more than one option for comparison
- Deep assignments are relatively rare
  - used primarily in distributed computing, and in particular for parameter passing in remote procedure call (RPC) systems
- For user-defined abstractions, no single language-specified mechanism for equality testing or assignment is likely to produce the desired results in all cases
  - Languages with sophisticated data abstraction mechanisms usually allow the programmer to define the comparison and assignment operators for each new data type—or to specify that equality testing and/or assignment is not allowed

## 7.4 Equality Testing and Assignment

```
class A {
public:
    bool operator==(const A& c) {
        return (a == c.a && b == c.b);
    }

    A& operator=(const A& c) {
        a = c.a;
        b = c.b;
    }

private:
    int a;
    int b;
}
```