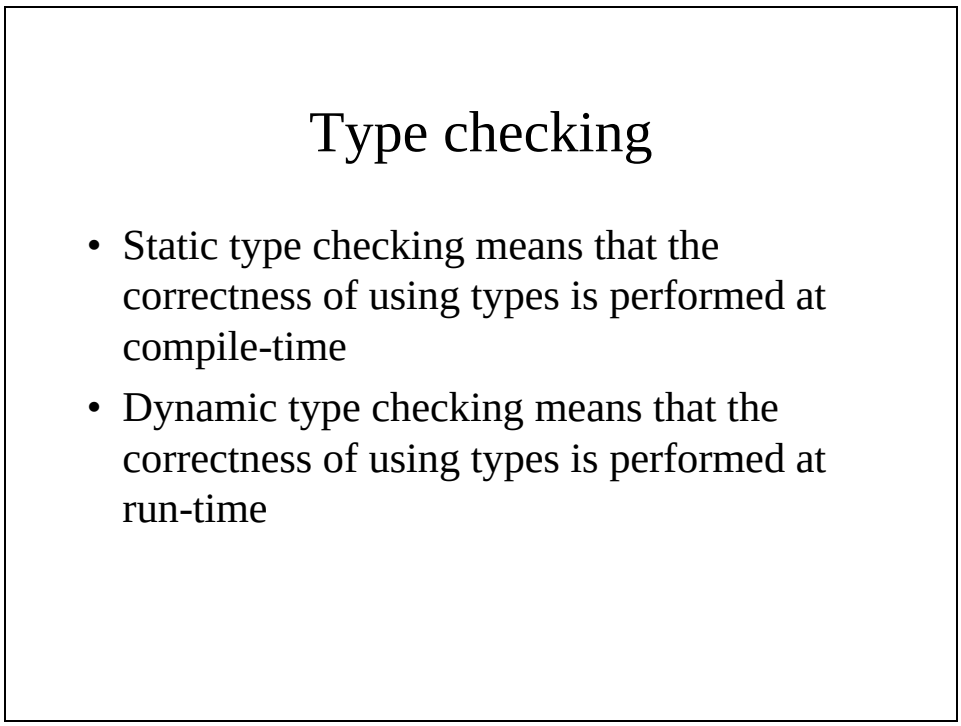
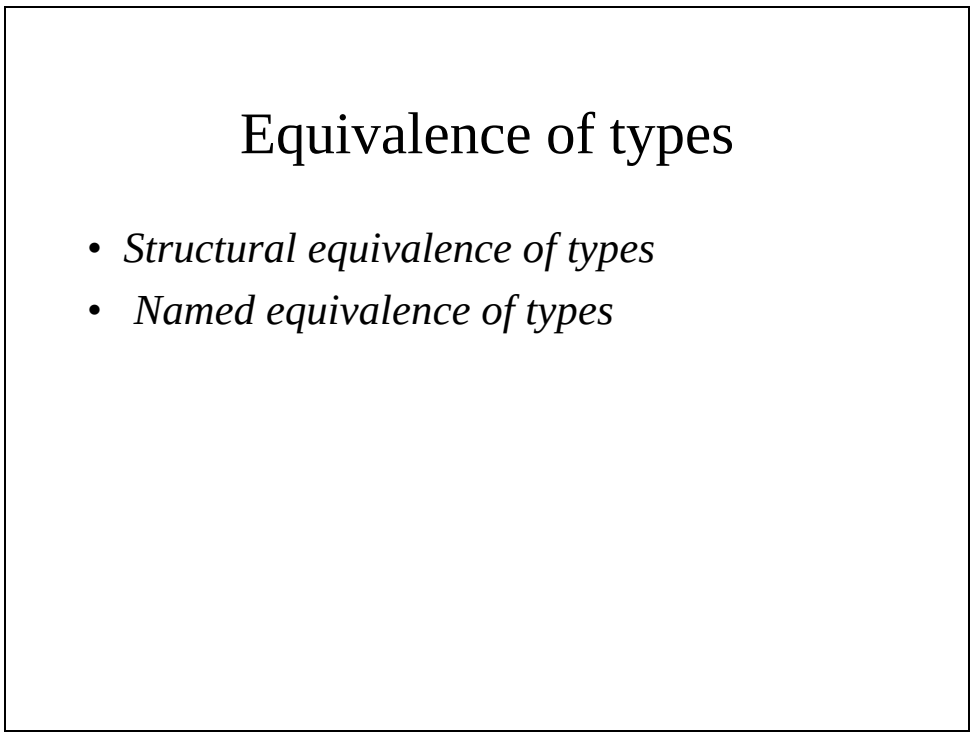


Type checking

If type checking is performed at compile-time, one may speak about *static type checking*; otherwise, that is, type checking is performed at run-time, a term *dynamic type checking* is applied. In principle, type checking can always be performed at run-time if the information about types of values in program is accessible in executing code. Obviously, dynamic type checking, compared to static one, leads to the increase of size and execution time of the object program and lowers the reliability of the compiled code. A programming language is called a *language with static type checking* or *strongly typed language*, if the type of each expression can be determined at compile-time, thereby guaranteeing that the type-related errors cannot occur in object program. Pascal is an example of a strongly typed language. However, even for Pascal some checks can be performed only dynamically. For instance, consider the following definitions:

```
table: array [0..255] of char;
i: integer;
```

A compiler cannot guarantee that the argument `i` in the expression `table[i]` is actually in the specified boundaries, that is, not less than zero and not greater than 255. In some cases, such checking can be performed by techniques like data flow analysis, but it is far from being always possible. It is clear that this sample demonstrates a common situation, namely, the control of slice arguments, occurring in the greater part of programming languages. Of course, this checking is usually performed dynamically.



Equivalence of types

The essential part of type checking is controlling *equivalence of types*. It is extremely important for a compiler to check the equivalence of types quickly.

Structural equivalence of types. Two types are called *structurally equivalent*, if both are either the same primitive type, or constructed by applying the same constructor to a structurally equivalent types. In other words, two types are structurally equivalent if and only if they are identical. We can check whether two types are structurally equivalent by using the following function:

```
bool sequiv (s, t)
{
	if (s and t – two identical primitive types)
	{
		return true;
	}
	else if (s == array (s1, s2) && t == array (t1, t2))
	{
		return sequiv (s1, t1) && sequiv (s2, t2);
	}
	else if (s == s1*s2 && t == t1*t2)
	{
		return sequiv (s1, t1) && sequiv (s2, t2);
	}
	else if (s==pointer (s1) && t == pointer (t1))
	{
		return sequiv (s1, t1);
	}
	else if (s==proc (s1, s2) && t == proc (t1, t2))
	{
		return sequiv (s1, t1) && sequiv (s2, t2);
	}
	else
	{
		return false;
	}
}
```

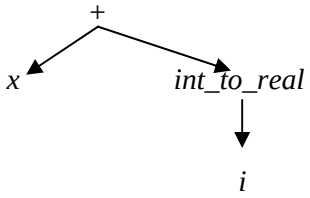
Type conversions

- Implicit conversions or coercions
- Explicit type conversions

Type conversions

Consider a formula $x+i$, where x is a variable of a **`real`** type, i is an **`integer`** variable. Since representations of integer and real numbers in memory are different, different instructions are used to handle integer and real values, and, commonly, there are no instructions with operands of different types, a compiler must convert one operand to the type of another, or convert both to some appropriate type.

Language specifications define what conversions are legal and what are necessary. If an integer value is assigned to a real variable, the conversion to the type of the assignment target is performed. However, the conversion of a real value to an integer is, generally, incorrect. In the case, a compiler usually converts an integer to a real. Type checking pass inserts conversions into the internal representation of the source program. For example, the following tree represents formula $x+i$ after type checking pass:



A conversion is called *implicit* if it is inserted automatically by the compiler. In many languages, implicit conversions, or *coercions*, are allowed only in situations when the loss of information cannot occur, for example, an integer can be safely converted to a real, whereas the reverse conversion is potentially dangerous with respect to information loss, and, therefore, extremely undesirable.

The conversion is called *explicit* if it is explicitly specified by a programmer. Explicit conversions resemble calls of functions defined on types. Sometimes type conversions are actually performed on expressions and are true functions, for example, Pascal provides built-in functions for type conversions: `ord` converts a character to an integer, `chr` performs the reverse conversion, and so on. Implicit calls are even more wide spread. The C programming language, for example, implicitly converts ASCII characters to integers in arithmetical expressions.