Wikipedia
The Free Encyclopedia

Type conversion

In computer science, **type conversion**,[1][2] **type casting**,[1][3] **type coercion**,[3] and **type juggling**[4][5] are different ways of changing an expression from one data type to another. An example would be the conversion of an integer value into a floating point value or its textual representation as a string, and vice versa. Type conversions can take advantage of certain features of type hierarchies or data representations. Two important aspects of a type conversion are whether it happens *implicitly* (automatically) or *explicitly*,[1][6] and whether the underlying data representation is converted from one representation into another, or a given representation is merely *reinterpreted* as the representation of another data type.[6][7] In general, both primitive and compound data types can be converted.

Each programming language has its own rules on how types can be converted. Languages with strong typing typically do little implicit conversion and discourage the reinterpretation of representations, while languages with weak typing perform many implicit conversions between data types. Weak typing language often allow forcing the compiler to arbitrarily interpret a data item as having different representations—this can be a non-obvious programming error, or a technical method to directly deal with underlying hardware.

In most languages, the word *coercion* is used to denote an *implicit* conversion, either during compilation or during run time. For example, in an expression mixing integer and floating point numbers (like 5 + 0.1), the compiler will automatically convert integer representation into floating point representation so fractions are not lost. Explicit type conversions are either indicated by writing additional code (e.g. adding type identifiers or calling built-in routines) or by coding conversion routines for the compiler to use when it otherwise would halt with a type mismatch.

In most ALGOL-like languages, such as Pascal, Modula-2, Ada and Delphi, *conversion* and *casting* are distinctly different concepts. In these languages, *conversion* refers to either implicitly or explicitly changing a value from one data type storage format to another, e.g. a 16-bit integer to a 32-bit integer. The storage needs may change as a result of the conversion, including a possible loss of precision or truncation. The word *cast*, on the other hand, refers to explicitly changing the *interpretation* of the *bit pattern* representing a value from one type to another. For example, 32 contiguous bits may be treated as an array of 32 booleans, a 4-byte string, an unsigned 32-bit integer or an IEEE single precision floating point value. Because the stored bits are never changed, the programmer must know low level details such as representation format, byte order, and alignment needs, to meaningfully cast.

In the C family of languages and ALGOL 68, the word *cast* typically refers to an *explicit* type conversion (as opposed to an implicit conversion), causing some ambiguity about whether this is a re-interpretation of a bit-pattern or a real data representation conversion. More important is the multitude of ways and rules that apply to what data type (or class) is located by a pointer and how a pointer may be adjusted by the compiler in cases like object (class) inheritance.

Explicit casting in various languages

Ada

Ada provides a generic library function Unchecked_Conversion.[8]

C-like languages

Implicit type conversion

Implicit type conversion, also known as *coercion* or *type juggling*, is an automatic type conversion by the compiler. Some programming languages allow compilers to provide coercion; others require it.

In a mixed-type expression, data of one or more subtypes can be converted to a supertype as needed at runtime so that the program will run correctly. For example, the following is legal C language code:

```c
double  d;
long    l;
int     i;

if (d > i)   d = i;
if (i > l)   l = i;
if (d == l)  d *= 2;
```

Although `d`, `l`, and `i` belong to different data types, they will be automatically converted to equal data types each time a comparison or assignment is executed. This behavior should be used with caution, as unintended consequences can arise. Data can be lost when converting representations from floating-point to integer, as the fractional components of the floating-point values will be truncated (rounded toward zero). Conversely, precision can be lost when converting representations from integer to floating-point, since a floating-point type may be unable to exactly represent all possible values of some integer type. For example, `float` might be an IEEE 754 single precision type, which cannot represent the integer 16777217 exactly, while a 32-bit integer type can. This can lead to unintuitive behavior, as demonstrated by the following code:

```c
#include <stdio.h>

int main(void)
{
    int i_value   = 16777217;
    float f_value = 16777216.0;
    printf("The integer is: %d\n", i_value);
    printf("The float is:   %f\n", f_value);
    printf("Their equality: %d\n", i_value == f_value);
}
```

On compilers that implement floats as IEEE single precision, and ints as at least 32 bits, this code will give this peculiar print-out:

```
The integer is: 16777217
The float is: 16777216.000000
Their equality: 1
```

Note that 1 represents equality in the last line above. This odd behavior is caused by an implicit conversion of `i_value` to float when it is compared with `f_value`. The conversion causes loss of precision, which makes the values equal before the comparison.

Important takeaways:

1. `float` to `int` causes truncation, i.e., removal of the fractional part.
2. `double` to `float` causes rounding of digit.
3. `long` to `int` causes dropping of excess higher order bits.

Type promotion

One special case of implicit type conversion is type promotion, where an object is automatically converted into another data type representing a superset of the original type. Promotions are commonly used with types smaller than the native type of the target platform's arithmetic logic unit (ALU), before arithmetic and logical operations, to make such operations possible, or more efficient if the ALU can work with more than one type. C and C++ perform such promotion for objects of boolean, character, wide character, enumeration, and short integer types which are promoted to int, and for objects of type float, which are promoted to double. Unlike some other type conversions, promotions never lose precision or modify the value stored in the object.

In Java:

```
int x = 3;
double y = 3.5;
System.out.println(x + y); // The output will be 6.5
```

Explicit type conversion

Explicit type conversion, also called type casting, is a type conversion which is explicitly defined within a program (instead of being done automatically according to the rules of the language for implicit type conversion). It is requested by the user in the program.

```
double da = 3.3;
double db = 3.3;
double dc = 3.4;
int result = (int)da + (int)db + (int)dc; // result == 9
// if implicit conversion would be used (as with "result = da + db + dc"), result would be
equal to 10
```

There are several kinds of explicit conversion.

checked

Before the conversion is performed, a runtime check is done to see if the destination type can hold the source value. If not, an error condition is raised.

unchecked

No check is performed. If the destination type cannot hold the source value, the result is undefined.

bit pattern

The raw bit representation of the source is copied verbatim, and it is re-interpreted according to the destination type. This can also be achieved via aliasing.

In object-oriented programming languages, objects can also be downcast : a reference of a base class is cast to one of its derived classes.

C# and C++

In C#, type conversion can be made in a safe or unsafe (i.e., C-like) manner, the former called *checked type cast*.[9]

```
Animal animal = new Cat();

Bulldog b = (Bulldog) animal;  // if (animal is Bulldog), stat.type(animal) is Bulldog, else
an exception
b = animal as Bulldog;         // if (animal is Bulldog), b = (Bulldog) animal, else b = null

animal = null;
b = animal as Bulldog;         // b == null
```

In C++ a similar effect can be achieved using *C++-style cast syntax*.

```
Animal* animal = new Cat;

Bulldog* b = static_cast<Bulldog*>(animal); // compiles only if either Animal or Bulldog is
derived from the other (or same)
b = dynamic_cast<Bulldog*>(animal);         // if (animal is Bulldog), b = (Bulldog*) animal,
else b = nullptr

Bulldog& br = static_cast<Bulldog&>(*animal); // same as above, but an exception will be
thrown if a nullptr was to be returned
                                              // this is not seen in code where exception
handling is avoided

delete animal; // always free resources
animal = nullptr;
b = dynamic_cast<Bulldog*>(animal);         // b == nullptr
```

Eiffel

In Eiffel the notion of type conversion is integrated into the rules of the type system. The Assignment Rule says that an assignment, such as:

```
x := y
```

is valid if and only if the type of its source expression, `y` in this case, is *compatible with* the type of its target entity, `x` in this case. In this rule, *compatible with* means that the type of the source expression either *conforms to* or *converts to* that of the target. Conformance of types is defined by the familiar rules for polymorphism in object-oriented programming. For example, in the assignment above, the type of y conforms to the type of x if the class upon which y is based is a descendant of that upon which x is based.

Definition of type conversion in Eiffel

The actions of type conversion in Eiffel, specifically *converts to* and *converts from* are defined as:

> A type based on a class CU *converts to* a type T based on a class CT (and T *converts from* U) if either
>
> > CT has a *conversion procedure* using U as a conversion type, or
> > CU has a *conversion query* listing T as a conversion type

Example

Eiffel is a fully compliant language for Microsoft .NET Framework. Before development of .NET, Eiffel already had extensive class libraries. Using the .NET type libraries, particularly with commonly used types such as strings, poses a conversion problem. Existing Eiffel software uses the string classes (such as STRING_8) from the Eiffel libraries, but Eiffel software written for .NET must use the .NET string class (System.String) in many cases, for example when calling .NET methods which expect items of the .NET type to be passed as arguments. So, the conversion of these types back and forth needs to be as seamless as possible.

```
    my_string: STRING_8                 -- Native Eiffel string
    my_system_string: SYSTEM_STRING     -- Native .NET string

        ...

            my_string := my_system_string
```

In the code above, two strings are declared, one of each different type (SYSTEM_STRING is the Eiffel compliant alias for System.String). Because System.String does not conform to STRING_8, then the assignment above is valid only if System.String converts to STRING_8.

The Eiffel class STRING_8 has a conversion procedure make_from_cil for objects of type System.String. Conversion procedures are also always designated as creation procedures (similar to constructors). The following is an excerpt from the STRING_8 class:

```
    class STRING_8
        ...
    create
        make_from_cil
        ...
    convert
        make_from_cil ({SYSTEM_STRING})
        ...
```

The presence of the conversion procedure makes the assignment:

```
            my_string := my_system_string
```

semantically equivalent to:

```
            create my_string.make_from_cil (my_system_string)
```

in which `my_string` is constructed as a new object of type `STRING_8` with content equivalent to that of `my_system_string`.

To handle an assignment with original source and target reversed:

```
            my_system_string := my_string
```

the class `STRING_8` also contains a conversion query `to_cil` which will produce a `System.String` from an instance of `STRING_8`.

```
    class STRING_8
        ...
    create
        make_from_cil
        ...
    convert
        make_from_cil ({SYSTEM_STRING})
        to_cil: {SYSTEM_STRING}
        ...
```

The assignment:

```
            my_system_string := my_string
```

then, becomes equivalent to:

```
            my_system_string := my_string.to_cil
```

In Eiffel, the setup for type conversion is included in the class code, but then appears to happen as automatically as explicit type conversion in client code. The includes not just assignments but other types of attachments as well, such as argument (parameter) substitution.

Rust

Rust provides no implicit type conversion (coercion) between primitive types. But, explicit type conversion (casting) can be performed using the `as` keyword.[10]

```
let x = 1000;
println!("1000 as a u16 is: {}", x as u16);
```

Type assertion

A related concept in static type systems is called **type assertion**, which instruct the compiler to treat the expression of a certain type, disregarding its own inference. Type assertion may be safe (a runtime check is performed) or unsafe. A type assertion does not convert the value from a data type to another.

Typescript

In Typescript, a type assertion is done by using the `as` keyword:[11]

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
```

In the above example, `document.getElementById` is declared to return an `HTMLElement`, but you know that it always return an `HTMLCanvasElement`, which is a subtype of `HTMLElement`, in this case. If it is not the case, subsequent code which relies on the behaviour of `HTMLCanvasElement` will not perform correctly, as in Typescript there is no runtime checking for type assertions.

In Typescript, there is no general way to check if a value is of a certain type at runtime, as there is no runtime type support. However, it is possible to write a user-defined function which the user tells the compiler if a value is of a certain type of not. Such a function is called **type guard**, and is declared with a return type of `x is Type`, where `x` is a parameter or `this`, in place of `boolean`.

This allows unsafe type assertions to be contained in the checker function instead of littered around the codebase.

Go

In Go, a type assertion can be used to access a concrete type value from an interface value. It is a safe assertion that it will panic (in the case of one return value), or return a zero value (if two return values are used), if the value is not of that concrete type. [12]

```
t := i.(T)
```

This type assertions tell the system that `i` is of type `T`. If it isn't, it panics.

Implicit casting using untagged unions

Many programming languages support union types which can hold a value of multiple types. *Untagged* unions are provided in some languages with loose type-checking, such as C and PL/I, but also in the original Pascal. These can be used to interpret the bit pattern of one type as a value of another type.

Security issues

In hacking, typecasting is the misuse of type conversion to temporarily change a variable's data type from how it was originally defined.[13] This provides opportunities for hackers since in type conversion after a variable is "typecast" to become a different data type, the compiler will treat that hacked variable as the new data type for that specific operation.[14]

See also

- Downcasting
- Run-time type information § C++ – dynamic cast and Java cast

- Type punning

References

1. Mehrotra, Dheeraj (2008). *S. Chand's Computer Science*. S. Chand. pp. 81–83. ISBN 978-8121929844.
2. *Programming Languages - Design and Constructs*. Laxmi Publications. 2013. p. 35. ISBN 978-9381159415.
3. Reilly, Edwin (2004). *Concise Encyclopedia of Computer Science* (https://archive.org/details/conciseencyclope0000unse_v5u2/page/82). John Wiley & Sons. pp. 82, 110 (https://archive.org/details/conciseencyclope0000unse_v5u2/page/82). ISBN 0470090952.
4. Fenton, Steve (2017). *Pro TypeScript: Application-Scale JavaScript Development*. Apress. pp. xxiii. ISBN 978-1484232491.
5. "PHP: Type Juggling - Manual" (http://php.net/manual/en/language.types.type-juggling.php). *php.net*. Retrieved 27 January 2019.
6. Olsson, Mikael (2013). *C++ Quick Syntax Reference*. Apress. pp. 87–89. ISBN 978-1430262770.
7. Kruse, Rudolf; Borgelt, Christian; Braune, Christian; Mostaghim, Sanaz; Steinbrecher, Matthias (16 September 2016). *Computational Intelligence: A Methodological Introduction*. Springer. p. 269. ISBN 978-1447172963.
8. "Unchecked Type Conversions" (https://www.adaic.org/resources/add_content/standards/95lrm/ARM_HTML/RM-13-9.html). *Ada Information Clearinghouse*. Retrieved 11 March 2023.
9. Mössenböck, Hanspeter (25 March 2002). "Advanced C#: Checked Type Casts" (http://ssw.jku.at/Teaching/Lectures/CSharp/Tutorial/Part2.pdf) (PDF). Institut für Systemsoftware, Johannes Kepler Universität Linz, Fachbereich Informatik. p. 5. Retrieved 4 August 2011. at C# Tutorial (http://ssw.jku.at/Teaching/Lectures/CSharp/Tutorial/)
10. "Casting - Rust By Example" (https://doc.rust-lang.org/rust-by-example/types/cast.html). *doc.rust-lang.org*.
11. " "Typescript documentation" " (https://www.typescriptlang.org/docs/handbook/2/everyday-types.html).
12. " "A Tour of Go" " (https://go.dev/tour/methods/15).
13. Jon Erickson *Hacking, 2nd Edition: The Art of Exploitation* 2008 1593271441 p51 "Typecasting is simply a way to temporarily change a variable's data type, despite how it was originally defined. When a variable is typecast into a different type, the compiler is basically told to treat that variable as if it were the new data type, but only for that operation. The syntax for typecasting is as follows: (typecast_data_type) variable ..."
14. Arpita Gopal *Magnifying C* 2009 8120338618 p. 59 "From the above, it is clear that the usage of typecasting is to make a variable of one type, act like another type for one single operation. So by using this ability of typecasting it is possible for create ASCII characters by typecasting integer to its ..."

External links

- Casting in Ada (http://www.adapower.com/index.php?Command=Class&ClassID=FAQ&CID=354)
- Casting in C++
- C++ Reference Guide (https://web.archive.org/web/20160709112746/http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=285) Why I hate C++ Cast Operators, by Danny Kalev

- Casting in Java (https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.5)
- Implicit Conversions in C# (http://msdn.microsoft.com/en-us/library/aa691280(v=vs.71).aspx)
- Implicit Type Casting at Cppreference.com (http://cppreference.com/wiki/language/implicit_cast)
- Static and Reinterpretation castings in C++ (http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/cast.html)
- Upcasting and Downcasting in F#

-