Type conversions

Michael Nowak

Texas A&M University

Notes

Overview

Notes

Overview

Type conversions
- Narrowing conversions
- Widening conversions
- Narrowing conversions vs. widening conversions

Coercion in expressions
- Implicit type conversion

Casts in expressions
- Explicit type conversion

Safe type conversions vs. unsafe type conversion in C++
- Safe type conversions
- Unsafe type conversions

Errors in expressions

References

Notes

Overview

Notes

Narrowing conversions

- A `narrowing conversion` converts a value to a type that cannot store even approximations of all of the values of the original type
- For example, converting a `double` to a `float` is a `narrowing conversion` in C++ because the rangle of `double` is much larger than that of `float`

Notes

Overview

Notes

Widening conversions

- A `widening conversion` converts a value to a type that can include at least approximations of all of the values of the original type
- For example, converting an `int` to a `double` is a `widening conversion`

Notes

Overview

Notes

Narrowing conversions vs. widening conversions

- `Narrowing conversions` are not always safe – sometimes the magnitude of the converted value is changed in the process
  - For example, converting the `double 1.5E25` to an `int` will result in a value that is not in any way related to the original value
- `Widening conversions` are nearly always `safe`, meaning that the approximate magnitude of the converted value is maintained
  - In C++, an `int` to `float` conversion is a `widening conversion`; some precision may be lost
    - For example, in many cases, integers are stored in 32 bits, which allows at least 9 decimal digits of precision
    - However, on my system `float` is also stored as 32 bits, with only about seven digits of precision (b/c of the space used for the sign and exponent)
    - We can lose precision when performing this conversion

Notes

Overview

Notes

Coercion in expressions

- One design decision concerning arithmetic expressions is whether an operator can have operands of different types
- Languages, such as C++, that allow `mixed-mode expressions`, must define conventions for implicit operand type conversions
  - This is because the underlying computer hardware does not have binary operations that take operands of different types
- When writing `mixed-mode expressions` in C++ using operands of the primitive built-in types, C++ converts the `narrower` type to the `wider` type

Notes

Overview

Notes

Implicit type conversion

- `Coercion` is an implicit type conversion that is initiated by the compiler or runtime system
- When two operands are not of the same type and that is legal in the language, the compiler must choose one of them to be `coerced` and generate the code for that `coercion`
- A broad range of `coercions` reduces the benefits of `type checking`

Notes

Implicit type conversion

- For example, consider the following:
```
int a;
double b, c, d;
...
d = b * a;
```
- Assume the second operand of `operator*` was suppose to be `c`, but we accidentally typed `a`
- Considering that `mixed-mode expressions` are legal in C++, the compiler would not detect this error
- Instead, it would insert code to coerce the value of the int operand to a `double`
- Had `mixed-mode expressions` been illegal in C++, this error would have been detected through the language's static type checking and reported as an error

Notes

Overview

Notes

Overview

Notes

Explicit type conversion

- Most languages, including C++, provide some capability for doing `explicit type conversion`s, both widening and narrowing
- In C++, such conversions are called `casts`
- To specify that a cast is to be performed by the compiler, we can write
  `static_cast<type_to_cast_to>(value_to_cast)`
  - This is helpful when we would like to perform `floating-point division` in an expression that contains two `int`egers

Notes

Explicit type conversion

- `explicit type conversion`s are helpful when we would like to perform `floating-point division` in an expression that contains two `int`egers
- For instance, in C++, when `int a = 9` and `int b = 5`,
  - `a / b` evaluates to `1` rather than the `1.8` we might have hoped for
  - To get the code mathematically correct, either `a` or `b` must be casted from an `int`eger to a `double`
  - This is as easy as writing the expression as `static_cast<double>(a) / b`
  - Of course, `b` will be coerced to a `double` before the expression is evaluated

Notes

Overview

Notes

Overview

Notes

`Safe type conversions` in C++

- A `safe type conversion` always converts the value to an equal value or to the best approximation of an equal value; this includes:
  - `bool` to `char`
  - `bool` to `int`
  - `bool` to `double`
  - `char` to `int`
  - `char` to `double`
  - `int` to `double`

Notes

Overview

Notes

`Unsafe type conversions` in C++

- In addition to `safe type conversion`, C++ also allows `unsafe type conversion`
- That is, a value of the respective type is converted into a value of another type that does not equal the original value
  - `double` to `int`
  - `double` to `char`
  - `double` to `bool`
  - `int` to `char`
  - `int` to `bool`
  - `char` to `bool`
- `Unsafe type conversion` are frequently a problem: often we do not suspect that an unsafe conversion is taking place

Notes

Overview

Notes

Errors in expressions

- A number of errors can present during expression evaluation
- Limitations of computer arithmetic and limitations of arithmetic are responsible for some of the errors we experience
- For instance, one of most common error occurs when the result of an operation cannot be represented in object where it is to be stored
  - This is called `overflow` or `underflow` depending on whether the result is too large or small

Notes

Overview

Notes

References

- Sebesta, R. W. (2016). *Concepts of programming languages* (11th ed.). Pearson Education.
- Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Addison-Wesley.

Notes