

## 1.4 TYPE CONVERSIONS IN C++

C++ allows us to treat a value of one type as being of another type. This is called **type conversion**.

In C++ there are two kinds of type conversions:

- implicit conversion
- explicit conversion

### Implicit conversion

Implicit conversion occurs **automatically** when we have operands of different types in an operation. The compiler automatically considers both data to be of the same type in order to perform the operation.

The implicit conversion is only done if the operands are of **compatible**. If the types are not compatible we get a **syntax error**!

### Examples

#### Conversion from int to double


```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int n = 5;
7      double x;
8
9      /// conversie de la int la double
10     x = n;
11
12     cout << "n = " << n << endl;
13     cout << "x = " << x << endl;
14     return 0;
15 }
```

```

n = 5
x = 5
```

## Conversion from double to int

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int n;
7      double x = 5.75;
8
9      /// conversie de la double la int
10
11     n = x;
12
13     cout << "n = " << n << endl;
14     cout << "x = " << x << endl;
15     return 0;
16 }
```



```
n = 5
x = 5.75
```

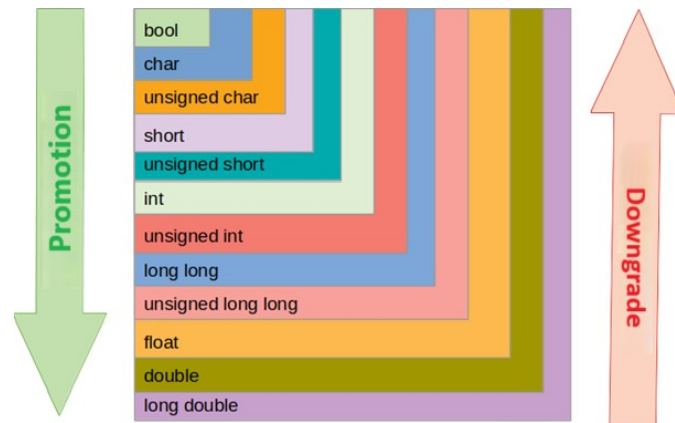
Since an *int* data type stores **integers**, the decimals of the x value **were lost** during the conversion.

## Data loss during conversion

As we saw in the previous example, type conversion can result in data being lost (**truncated**). This is not necessarily bad, but we need to be aware of this situation.

In fact, the conversion is of two kinds:

- **promotion** – conversion from a lower data type to a higher one; **no data loss!**
- **downgrade** – conversion from a higher data type to a lower one; **may cause data loss!**



In the first example above there was a promotion from `int` to `double` and in the second example there was a demotion from `double` to `int`. The second one led to the loss of some data.

For example, the result of the expression `2 + 1.5` will be `3.5`, not `3`. The value `2` is promoted to `double`, not `1.5` downgraded to `int`, which would have meant data loss!

## The explicit conversion operator

It is a unary operator and it has high priority.

Its result is the result of the expression, considered by the `TYPE` data type.

It has the syntax: `(TYPE) phrase` or `TYPE (phrase)`. In the second variant, `TYPE` must be a one-word type name. Thus, the expression `unsigned int(expression)` is wrong.

e.g.:

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      char c='A';
7      cout << (int) c << endl; // 65
8      cout << char(97) << endl; // a
9  }
```

## Rules about conversion

1. If a negative integer is converted to an *unsigned* type the result will be the base 2's complement of the original value, due to the way integer values are represented. Thus, `-1` becomes the highest value of the *unsigned* type, `-2` becomes the second highest value, etc.
2. When converting a numeric value to the `bool` type, the result will be:

- **true** (if the initial value is non-zero)
  - **false** (if the initial value is 0)
3. When converting a value of type *float*, *double* to an *integer* type, the value will be **truncated**, losing the decimal part. If the result does not fall within the bounds of the integer type, the behavior of the program becomes unpredictable.

The above rules apply to both **implicit** and **explicit** conversion.

### Conversion example (following the rules) - arithmetic mean

Suppose we want to determine the arithmetic mean of three integers. Of course, the answer is the sum of the numbers divided by 3, but keep in mind that the division of two integers is the quotient of division. To get the correct average we will need to do some conversions.

#### 1. Implicit conversion

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a , b, c;
7      cin >> a >> b >> c;
8      int S = a + b + c;
9      /// cout << S / 3; // wrong - integer division
10     cout << S / 3.0;
11 }
```

Implicit conversion of S to *double* (promotion) occurred.

#### 2. Implicit conversion

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a , b, c;
7      cin >> a >> b >> c;
8      int S = a + b + c;
9      cout << 1.0 * S / 3;
10 }
```

Implicit conversion of S, then 3, to *double* (promotions) occurred.

#### 3. Explicit conversion

```

main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a , b,  c;
7      cin >> a >> b >> c;
8      int S = a + b + c;
9      cout << (double)S / 3;
10 }

```

First the value of *S* is explicitly converted to *double*, then the implicit conversion to *double* of 3 takes place.

## Avoiding type overflow

In many situations we have operations with data of type *int*, but the result exceeds the maximum (or minimum) limit of this type. This is how overflow occurs. The solution is to use conversions that perform the operations in a wider data type, for example *long long*.

e.g.:

```

main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int n = 1000000;
7      cout << n * n << endl; /// possible -727379968 - overflow
8      cout << 1LL * n * n << endl; /// correct 1000000000000
9      cout << (long long) n * n << endl; /// correct
10 }

```

## Pay attention to!

- Where possible, the compiler performs implicit conversions by promotion.
- When performing the conversions, the precedence of the operators must be taken into account. For example, the sequence below does not get the correct result.

