1 Integer in Fortran

1.1 Objective

Understand how **integer overflow** occurs when computing large values such as factorials and how the choice of integer `kind` affects the maximum representable number.

1.2 Background

Fortran supports different integer sizes (often 2, 4, and 8 bytes). A small type (e.g. 16-bit) can only hold values up to about 3×10^4, a 32-bit up to about 2×10^9, and a 64-bit up to about 9×10^{18}, see table 1.2. When the value exceeds this range, it *overflows*, wrapping to a wrong (often negative) number.

Table 1: Common `INTEGER` kinds in Fortran (kind values and ranges are compiler dependent). *Kind 16 (128-bit) is a non-standard extension available only in some compilers such as gfortran.

Kind value	**Typical storage**	**Approx. decimal range**
1	1 byte (8-bit)	$-128 \dots 127$
2	2 bytes (16-bit)	$-32\,768 \dots 32\,767$
4	4 bytes (32-bit)	$-2\,147\,483\,648 \dots 2\,147\,483\,647$
8	8 bytes (64-bit)	$-9.22 \times 10^{18} \dots 9.22 \times 10^{18}$
16*	16 bytes (128-bit)	$\sim \pm 1.7 \times 10^{38}$

1.3 Example Program

```
program factorial_overflow
    implicit none
    integer(kind=2) :: f2 ! 2-byte integer (usually 16-bit)
    integer(kind=4) :: f4 ! 4-byte integer (usually 32-bit)
    integer(kind=8) :: f8 ! 8-byte integer (usually 64-bit)
    integer :: n

    print *, " n", "  int16", "         int32", "                  int64"
    print *, "------------------------------------------------------------"

```

```
    f2 = 1
    f4 = 1
    f8 = 1

    do n = 1, 25
        if (n > 1) then
            f2 = f2 * n
            f4 = f4 * n
            f8 = f8 * n
        end if
        print *, n, f2, f4, f8
    end do
end program factorial_overflow
```

1.4 Expected Observations

- `integer(kind=2)` (16-bit) will overflow after $7! = 5040$.
- `integer(kind=4)` (32-bit) overflows after $12! = 479001600$.
- `integer(kind=8)` (64-bit) can hold up to $20!$ ($\approx 2.43 \times 10^{18}$) but fails at $21!$.

1.5 Discussion Points

- Overflow wraps around to negative or strange values.
- Choosing the correct integer `kind` is crucial for large numbers.
- You can request a type with enough digits using:

```
integer, parameter :: ik = selected_int_kind(18) ! at least 18 decimal digits
integer(kind=ik) :: big
```

1.6 Exercise

1. Compile and run the program.
2. Identify at which n each integer type overflows.
3. Modify the program to use `selected_int_kind` and test larger factorials.

2 Real numbers in fortran

2.1 Floating-Point Representation and the Mantissa

$$\overbrace{-}^{\text{sign}}\ \overbrace{0.1234567890123456}^{\text{mantissa (significand)}} \times 10^{\overbrace{23}^{\text{exponent}}}$$

Figure 1: Example of decimal (base-10) floating-point form: sign, mantissa (significand), and exponent.

Computers store real numbers in a binary *floating-point* format similar to scientific notation. A number is written in the general form

$$x = \pm m \times b^e$$

where

- m is the **mantissa** (also called the *significand*),
- b is the base (usually 2 in computers),
- e is the **exponent**.

Fortran `REAL` types typically use IEEE 754 floating-point representation:

- The **sign bit** stores whether the number is positive or negative.
- The **exponent field** stores e, which shifts the decimal (binary) point.
- The **mantissa field** stores the significant digits of the number.

2.1.1 Why the Mantissa Matters

The mantissa determines the *precision* of the number — how many significant decimal digits you can trust. For example:

- Single precision (`real(kind=4)`) mantissa $\approx$ 24 binary bits $\Rightarrow$ about 7 decimal digits.
- Double precision (`real(kind=8)`) mantissa $\approx$ 53 binary bits $\Rightarrow$ about 15–16 decimal digits.

- Quad precision (`real(kind=16)`) mantissa ≈ 113 binary bits $\Rightarrow$ about 33–34 decimal digits.

A larger mantissa means you can represent numbers with more significant digits without rounding error.

2.1.2 Example

If you assign in Fortran:

```
real(kind=4) :: a
a = 1.0/3.0
print *, a
```

you will see that after about 7 digits the value stops being exact (e.g. 0.33333334). With `real(kind=8)` the printed digits stay accurate to about 15 decimal places.

2.1.3 Key Points for Students

- The mantissa controls **precision** (number of reliable digits), not the range of exponents.
- Overflow is governed by the exponent size; rounding error and precision loss are governed by the mantissa size.
- Choose a `REAL` kind with a mantissa long enough for your problem (use `selected_real_kind`).

2.2 Objective

Understand how different `REAL` kinds affect **precision and numerical range**. Observe **overflow** and **loss of precision** when values exceed the type limits.

2.3 Background

Fortran provides several floating-point types, usually:

- `real(kind=4)` — single precision (about 7 decimal digits, up to $\pm 10^{38}$)
- `real(kind=8)` — double precision (about 15–16 digits, up to $\pm 10^{308}$)
- `real(kind=16)` — quad precision (about 33 digits, up to $\pm 10^{4932}$, non-standard)

Choosing the correct kind is essential for both accuracy and avoiding overflow/underflow.

2.4 Example Program

```
program real_precision
    implicit none
    real(kind=4) :: r4 ! single precision (~7 digits)
    real(kind=8) :: r8 ! double precision (~15 digits)
    real(kind=16) :: r16 ! quad precision (~33 digits, if supported)
    integer :: i

    print *, "Step  |    real(4)          real(8)                    real(16)"
    print *, "--------------------------------------------------------------------"

    r4 = 1.0
    r8 = 1.0
    r16 = 1.0

    do i = 1, 1000
        r4 = r4 * 10.0
        r8 = r8 * 10.0
        r16 = r16 * 10.0
        if (mod(i,50) == 0) then
            print *, i, r4, r8, r16
        end if
    end do
end program real_precision
```

2.5 Expected Observations

- `real(kind=4)` overflows (becomes `Inf` or strange) around 10^{38}.
- `real(kind=8)` works until about 10^{308} but has only ~ 15 digits of precision.
- `real(kind=16)` (if supported) extends range and precision up to $\sim 10^{4932}$.
- Very small numbers may **underflow** to zero.

2.6 Requesting a Suitable Kind

Instead of hardcoding `4` or `8`, you can ask the compiler:

```
integer, parameter :: dp = selected_real_kind(p=15, r=300)
real(kind=dp) :: x
```

This requests at least 15 decimal digits of precision and exponent range $\pm 10^{300}$.

2.7 Typical REAL Kinds

Kind	Storage	Decimal digits	Approx. range
4	4 bytes (single)	∼7	$\pm 10^{\pm 38}$
8	8 bytes (double)	∼15	$\pm 10^{\pm 308}$
16*	16 bytes (quad)	∼33	$\pm 10^{\pm 4932}$

Table 2: Common `REAL` kinds in Fortran. *Kind 16 is a non-standard extension available only in some compilers (e.g. gfortran).

2.8 Exercises

1. Compile and run the example program. Note when each type prints `Inf` or stops changing meaningfully.
2. Try dividing repeatedly by 10.0 to see **underflow**.
3. Use `selected_real_kind` to request 30 digits and observe if your compiler supports `kind=16`.

2.9 Loss of Significance and Machine Epsilon

Floating–point numbers have limited precision: when a number becomes very small compared to another, adding it may no longer change the sum. This experiment shows how, when a value gets smaller than the machine precision (*epsilon*), adding it to 1.0 no longer increases the result.

2.9.1 Example Program

```fortran
program compare_reals
    implicit none
    real(kind=8) :: big, small, sum
    integer :: i

    big = 1 ! start with 1.0
    small = 1 ! also start at 1.0

    print *, " i", "       small", "               big", "               big+small"
    print *, "
    --------------------------------------------------------------
    "

    do i = 1, 60
        sum = big + small
        print *, i, small, big, sum

        if (sum <= big) then
            print *, "==> At step ", i, " adding SMALL no longer increases BIG."
            exit
        end if

        small = small / 2 ! decrease small each loop
    end do
end program compare_reals
```

What to Observe

- The value of `small` is halved in each loop.
- At first, `big + small` increases.
- Eventually, `big + small` equals `big`, meaning `small` is below the machine precision.

Key Idea

- Floating–point numbers have a finite number of bits in the **mantissa**.
- There exists a smallest representable increment above 1.0, called the *machine epsilon*.

- Any smaller number added to 1.0 is ignored due to rounding.