Exercise 5 – Floating point representation

Informatik I für Mathematiker und Physiker (HS 2015)

Yeara Kozlov

Slides courtesy of Kaan Yücer & Endri Dibra

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatik
Computer Science

Agenda

- HW #3 feedback
- Floating points
  - Representation
  - Standard
  - Good practice
  - Comparison
- Functions - PRE and POST Conditions

Agenda

- **HW #3 feedback**
- Floating points
  - Representation
  - Standard
  - Good practice
  - Comparison
- Functions - PRE and POST Conditions

HW #3 Feedback

- for loops
- while loops
- modulus

for Loop

```
for (int i=0; i < 4; i++) {
        cout << "i is: " << i;
}
```

- Declare and initialize loop variable: `int i=0;`
- Check continue condition ***BEFORE***: `i < 4;`
- Increment ***AFTER*** loop `i++`

while Loop

- Declare and initialize loop variable: `int i=0;`
- Check continue condition ***BEFORE***: `i < 4;`
- Increment ***AFTER*** loop `i++`

```
int i=0;
while (i < 4) {
      cout << "i is: " << i;
      i++;
}
```

Loop choice

- minimal code
- easy to understand

for	Some counting is done, but the counter is not needed after the loop. e.g. repeat something n times
while	The loop condition depends on variables which already exist before the loop. e.g. decrease x until it's a power of 5
do	The loop condition depends on variables which are obtained in the loop body. e.g. execute std::cin >> x until x > 3

Loop choice

- n can vary according to input - for example, number of digits

for	Some counting is done, but the counter is not needed after the loop. e.g. repeat something n times
while	The loop condition depends on variables which already exist before the loop. e.g. decrease x until it’s a power of 5
do	The loop condition depends on variables which are obtained in the loop body. e.g. execute std::cin >> x until x > 3

Modulus

- What happens here?

```
std::cout << 9 % 9 << std::endl;
```

```
int res = digit % 2;
if ( res == 0)
  std::cout << 0 << std::endl;
else
  std::cout << res << std::endl;
```

- Modulus always returns a numeric value:

```
std::cout << res << std::end;
```

Agenda

- HW #3 feedback
- **Floating points**
  - **Representation**
  - Standard
  - Good practice
  - Comparison
- Functions - PRE and POST Conditions

Binary representation of non-decimal numbers

Representing 7.25 as a binary decimal:

decimal	8	4	2	1	1/2	1/4	1/8	
powers of 2	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	
decimal, linear	$0*2^3$	$+ 1*2^2$	$+ 1*2^1$	$+ 1*2^0$	$+0*2^{-1}$	$+1*2^{-2}$	$+0*2^{-3}$	= 7.25
binary	0	1	1	1	0	1	0	= 111.010

Binary representation of floating point numbers

- Converting **1.9**$_{\text{dec}}$ to binary

x	b_i	$x-b_i$
1.9	1	0.9
1.8	1	0.8
1.6	1	0.6
1.2	1	0.2
0.4	0	0.4
0.8	0	0.8
1.6	1	

1.6 beginning of repetitive part.

Final representation:

$$1.1\overline{1100}$$

Binary representation (Examples)

Compute the binary expansion of the following decimal numbers:

- 0.25
- 1.52
- 1.3
- 11.1

- 0.25 = 1 / 4 = 2^-2
- 0.01

Binary representation (Examples)

1.52

$1.\overline{10000101000111101011}$

				1.52	$\rightarrow$	b_0	$= 1$
$2(1.52-1)$	$=$	$2 \cdot 0.52$	$=$	1.04	$\rightarrow$	b_{-1}	$= 1$
$2(1.04-1)$	$=$	$2 \cdot 0.04$	$=$	0.08	$\rightarrow$	b_{-2}	$= 0$
$2(0.08-0)$	$=$	$2 \cdot 0.08$	$=$	0.16	$\rightarrow$	b_{-3}	$= 0$
$2(0.16-0)$	$=$	$2 \cdot 0.16$	$=$	0.32	$\rightarrow$	b_{-4}	$= 0$
$2(0.32-0)$	$=$	$2 \cdot 0.32$	$=$	0.64	$\rightarrow$	b_{-5}	$= 0$
$2(0.64-0)$	$=$	$2 \cdot 0.64$	$=$	1.28	$\rightarrow$	b_{-6}	$= 1$
$2(1.28-1)$	$=$	$2 \cdot 0.28$	$=$	0.56	$\rightarrow$	b_{-7}	$= 0$
$2(0.56-0)$	$=$	$2 \cdot 0.56$	$=$	1.12	$\rightarrow$	b_{-8}	$= 1$
$2(1.12-1)$	$=$	$2 \cdot 0.12$	$=$	0.24	$\rightarrow$	b_{-9}	$= 0$
$2(0.24-0)$	$=$	$2 \cdot 0.24$	$=$	0.48	$\rightarrow$	b_{-10}	$= 0$
$2(0.48-0)$	$=$	$2 \cdot 0.48$	$=$	0.96	$\rightarrow$	b_{-11}	$= 0$
$2(0.96-0)$	$=$	$2 \cdot 0.96$	$=$	1.92	$\rightarrow$	b_{-12}	$= 1$
$2(1.92-1)$	$=$	$2 \cdot 0.92$	$=$	1.84	$\rightarrow$	b_{-13}	$= 1$
$2(1.84-1)$	$=$	$2 \cdot 0.84$	$=$	1.68	$\rightarrow$	b_{-14}	$= 1$
$2(1.68-1)$	$=$	$2 \cdot 0.68$	$=$	1.36	$\rightarrow$	b_{-15}	$= 1$
$2(1.36-1)$	$=$	$2 \cdot 0.36$	$=$	0.72	$\rightarrow$	b_{-16}	$= 0$
$2(0.72-0)$	$=$	$2 \cdot 0.72$	$=$	1.44	$\rightarrow$	b_{-17}	$= 1$
$2(1.44-1)$	$=$	$2 \cdot 0.44$	$=$	0.88	$\rightarrow$	b_{-18}	$= 0$
$2(0.88-0)$	$=$	$2 \cdot 0.88$	$=$	1.76	$\rightarrow$	b_{-19}	$= 1$
$2(1.76-1)$	$=$	$2 \cdot 0.76$	$=$	1.52	$\rightarrow$	b_{-20}	$= 1$

Binary representation (Examples)

Compute the binary expansion of the following decimal numbers: 1.3

$$
\begin{array}{rcccccl}
 & & & & 1.3 & \longrightarrow & b_0 = 1 \\
2(1.3-1) & = & 2\cdot 0.3 & = & 0.6 & \longrightarrow & b_{-1} = 0 \\
2(0.6-0) & = & 2\cdot 0.6 & = & 1.2 & \longrightarrow & b_{-2} = 1 \\
2(1.2-1) & = & 2\cdot 0.2 & = & 0.4 & \longrightarrow & b_{-3} = 0 \\
2(0.4-0) & = & 2\cdot 0.4 & = & 0.8 & \longrightarrow & b_{-4} = 0 \\
2(0.8-0) & = & 2\cdot 0.8 & = & 1.6 & \longrightarrow & b_{-5} = 1 \\
2(1.6-1) & = & 2\cdot 0.6 & = & 1.2 & \longrightarrow & b_{-6} = 1 \\
 & & & & & \vdots &
\end{array}
$$

$$1.0\overline{1001}$$

Binary representation (Examples)

Compute the binary expansion of the following decimal numbers: 11.1

$$
\begin{array}{rclclcl}
 & & & & 1.1 & \rightarrow & b_0 = 1 \\
2(1.1-1) & = & 2\cdot 0.1 & = & 0.2 & \rightarrow & b_{-1} = 0 \\
2(0.2-0) & = & 2\cdot 0.2 & = & 0.4 & \rightarrow & b_{-2} = 0 \\
2(0.4-0) & = & 2\cdot 0.4 & = & 0.8 & \rightarrow & b_{-3} = 0 \\
2(0.8-0) & = & 2\cdot 0.8 & = & 1.6 & \rightarrow & b_{-4} = 1 \\
2(1.6-1) & = & 2\cdot 0.6 & = & 1.2 & \rightarrow & b_{-5} = 1 \\
2(1.2-1) & = & 2\cdot 0.2 & = & 0.4 & \rightarrow & b_{-6} = 0 \\
 & & & & \vdots & &
\end{array}
$$

$$1011.0\overline{0011}$$

Agenda

- HW #3 feedback
- **Floating points**
  - Representation
  - **Standard**
  - Good practice
  - Comparison
- Functions - PRE and POST Conditions

Fixed Floating Point Representation

- finite subset of $\mathbb{R}$
- defined by 4 numbers:
  - the base: $2 \leq \beta \in \mathbb{N}$
  - the precision: $1 \leq \rho \in \mathbb{N}$
  - the smallest exponent: $e_{min} \in \mathbb{Z}$
  - the largest exponent: $e_{max} \in \mathbb{Z}$
- the real numbers represented by this system:

$$s \cdot \sum_{i=0}^{\rho-1} d_i \beta^{-i} \cdot \beta^e$$

$s \in \{-1,1\}$

$d_i \in \{0, \dots, \beta - 1\}$ for all i

$e \in \{e_{min}, \dots, e_{max}\}$

Fixed Floating Point Representation

$\mathcal{F}(\beta, \rho, e_{min}, e_{max})$

$$s \cdot \sum_{i=0}^{\rho-1} d_i \beta^{-i} \cdot \beta^e$$

or in other form: $\pm d_0 . d_1 \ldots d_{\rho-1} \cdot \beta^e$

sign

exponent

significand (=mantissa)

- **Example:** $\beta = 10$, number 0.1

  → $1.0 \cdot 10^{-1}$ or $0.1 \cdot 10^0$ or $0.01 \cdot 10^1, \ldots$

$\mathcal{F}^*(\beta, \rho, e_{min}, e_{max})$

normalized form (same, but $d_0 \neq 0$)

- **Example:** $\beta = 10$, number 0.1

  → $1.0 \cdot 10^{-1}$ ~~or $0.1 \cdot 10^0$ or $0.01 \cdot 10^1, \ldots$~~

Creating 10-bit floating point type

- Goal:
  - store as many different non-integer values as possible,
  - into a finite and small unit (10 bits in our case)

Creating 10-bit floating point type

9.	8.	7.	6.	5.	4.	3.	2.	1.	0.
exp4	*exp3*	*exp2*	*exp1*	*mant5*	*mant4*	*mant3*	*mant2*	*mant1*	*digit*

$\boxed{\mathcal{F}(2,6,e_{min},e_{max})}$ $e_{min} = ?$ $e_{max} = ?$

1. if exponent bits represent an `unsigned int` value →

   $e_{min} = 0000_{bin} = 0_{dec}$ and $e_{max} = 1111_{bin} = 15_{dec}$

2. if exponent bits represent an `signed int` value →

   $e_{min} = 1000_{bin} = -\mathbf{8}_{dec}$ and $e_{max} = 0111_{bin} = \mathbf{7}_{dec}$

$\boxed{\mathcal{F}(2,6,-8,7)}$ smallest: $0.00001 \cdot 2^{1000}$ largest: $1.11111 \cdot 2^{0111}$

Creating 10-bit floating point type

1. let's use one bit as sign to get negative numbers

   → 1 bit space needed

2. switch to normalized scientific form

   → digit before decimal point is always 1

   → no need to explicitly store

   → 1 bit space frees up

$\mathcal{F}^*(2,6,-8,7)$

9.	8.	7.	6.	5.	4.	3.	2.	1.	0.
exp4	*exp3*	*exp2*	*exp1*	*mant5*	*mant4*	*mant3*	*mant2*	*mant1*	*~~digit~~ sign*

Examples at this point:

- smallest: $-1.11111 \cdot 2^{0111}$, largest: $+1.11111 \cdot 2^{0111}$
- one random number from the middle range: $-1.01001 \cdot 2^{1001}$

Creating 10-bit floating point type

- How to represent 0?

  The closest at this point: $\pm 1.00000 \cdot 2^{1000} = \pm \frac{1}{256}$

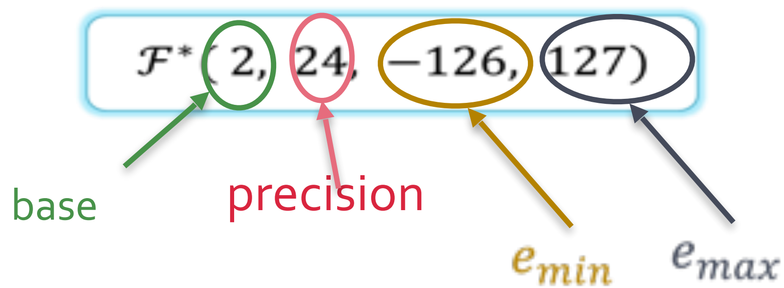
→ let's "sacrifice" one value from the range of exponent ($1000_{bin} = -8_{dec}$)

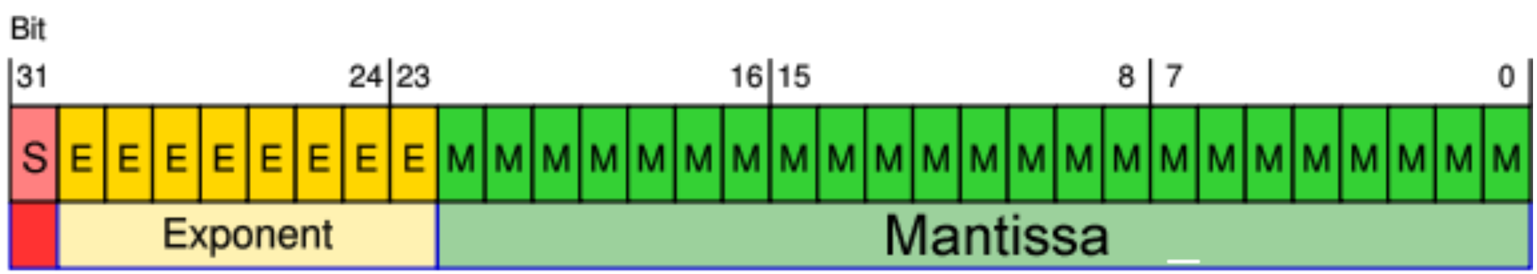
9.	8.	7.	6.	5.	4.	3.	2.	1.	0.
exp4	*exp3*	*exp2*	*exp1*	*mant5*	*mant4*	*mant3*	*mant2*	*mant1*	~~*digit*~~ *sign*

- if exp == 1000_{bin} → different rule when interpreting the number
  - mant == 00000_{bin} means 0 mant == 00001_{bin} means $+\infty$
  - mant == 00010_{bin} means $-\infty$ mant == 00011_{bin} means NaN (not a number)

IEEE 754 Standard Representation

- internal representation of **`float`**



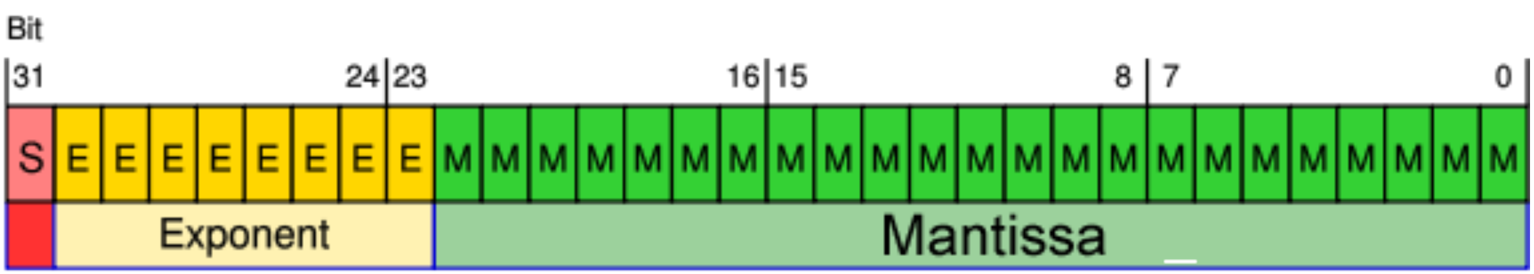


IEEE 754 Standard Representation

- internal representation of **`double`**

$$\mathcal{F}^*(2, 53, -1022, 1023)$$

base precision e_{min} e_{max}



IEEE 754 - min and max

- $+1.11111111111111111111111 * 2^{127}$
- More elegant calculation: $\left(1-\left(\frac{1}{\beta}\right)^{\rho}\right)\beta^{e_{max}+1}$

$\mathcal{F}^*(2,\ 24,\ -126,\ 127)$

$$\left(1-\left(\frac{1}{2}\right)^{24}\right)2^{127+1} = 2^{128} - 2^{104}$$

$\mathcal{F}^*(2,53,-1022,1023)$

$$\left(1-\left(\frac{1}{2}\right)^{53}\right)2^{1023+1} = 2^{1024} - 2^{971}$$

Binary representation of 0.1

- the decimal system is based on two prime numbers: `2` and `5` (`10=2*5`)
- the binary system is based on `2`
- *(the hexadecimal system is based on* `2`*, too:* `16=2*2*2*2`*)*
- `0.1 = 1/10 = 1/(2*5)`

→ no finite representation in binary (neither in hexadecimal)

Agenda

- HW #3 feedback
- Floating points
  - Representation
  - Standard
  - **Good practice**
  - Comparison
- Functions - PRE and POST Conditions

Guidelines

Guideline 1:

«Do **not** test two floating point numbers for **equality**, if at least one of them was rounded before.»

Guideline 1 – Example

Guideline 1:

«Do **not** test two floating point numbers for **equality**, if at least one of them was rounded before.»

This is `false`

Example:

```
float a = 0.1f;
if (10*a == 1.0f)
    std::cout << "no output\n";
```

Guideline 1 – Example

Guideline 1:

> «Do **not** test two floating point numbers for **equality**, if at least one of them was rounded before.»

This is `false`

Example:

```
float a = 0.1f;
if (10*a == 1.0f)
    std::cout << "no output\n";
```

Problem:

`0.1f` not representable

Guideline 1 – Example

Guideline 1:

> «Do **not** test two floating point numbers for **equality**, if at least one of them was rounded before.»

This is `false`

Example:

```
float a = 0.1f;
if (10*a == 1.0f)
    std::cout << "no output\n";
```

Problem:

`0.1f` not representable

$$
\begin{array}{rcll}
 & 0.1 & = & \overbrace{1.100110011001100110011001}^{\text{24bit}}1\ldots \cdot 2^{-4} \\
\text{(rounding)} & \rightarrow 0.099\ldots & = & 1.10011001100110011001101 \cdot 2^{-4}
\end{array}
$$

Guidelines

Guideline 1:

«Do **not** test two floating point numbers for **equality**, if at least one of them was rounded before.»

Guideline 2:

«**Avoid** the **addition** of numbers of extremely **different sizes**!»

Guideline 2 – Example

Guideline 2:

«**Avoid** the **addition** of numbers of extremely **different sizes**!»

Example:

```
float a = 67108864.0f + 1.0f;

if (a > 67108864.0f)
    std::cout << "This is not output ... \n";
```

Guideline 2 – Example

Guideline 2:

«**Avoid** the **addition** of numbers of extremely **different sizes**!»

Problem:

Significand too short

Example:

```
float a = 67108864.0f + 1.0f;

if (a > 67108864.0f)
    std::cout << "This is not output ... \n";
```

Guideline 2 – Example

Guideline 2:

«**Avoid** the **addition** of numbers of extremely **different sizes**!»

Problem:

Significand too short

Example:

```
float a = 67108864.0f + 1.0f;

if (a > 67108864.0f)
    std::cout << "This is not output ... \n";
```

$$
\begin{array}{rcl}
67108864 & = & \overbrace{1.000000000000000000000000}^{\text{24bit}} \cdot 2^{26} \\
+1 & = & 0.00000000000000000000000001 \cdot 2^{26} \\
\hline
67108865 & = & 1.00000000000000000000000001 \cdot 2^{26}
\end{array}
$$

Guideline 2 – Example

Guideline 2:

«**Avoid** the **addition** of numbers of extremely **different sizes**!»

Problem:

Significand too short

Example:

```
float a = 67108864.0f + 1.0f;

if (a > 67108864.0f)
    std::cout << "This is not output ... \n";
```

$$
\begin{array}{rrll}
 & 67108864 = & \overbrace{1.000000000000000000000000}^{\text{24bit}} & \cdot 2^{26} \\
 & +1 = & 0.00000000000000000000000001 & \cdot 2^{26} \\
\hline
 & 67108865 = & 1.00000000000000000000000001 & \cdot 2^{26} \\
\text{(rounding)} & \rightarrow 67108864 = & 1.000000000000000000000000 & \cdot 2^{26}
\end{array}
$$

Agenda

- HW #3 feedback
- Floating points
  - Representation
  - Standard
  - Good practice
  - **Comparison**
- Functions - PRE and POST Conditions

The Comparison Problem

- Given `fp1` and `fp2` of type `float` or `double`.

- Guideline 1:

  «Do **not** test two floating point numbers for **equality**, if at least one of them was rounded before.»

The Comparison Problem

- Given `fp1` and `fp2` of type `float` or `double`.

- Guideline 1:

  «Do **not** test two floating point numbers for **equality**, if at least one of them was rounded before.»

- Thus `fp1 == fp2` should be **avoided**.

The Comparison Problem

- How can we compare instead?

The Comparison Problem

- How can we compare instead?

- First idea:
  Allow for small differences!

Given: tolerance value $c > 0$.

`fp1` **"equals"** **`fp2`** whenever **`|fp1 - fp2| < c`**

(Remark: | … | means absolute value. In C++ it's not available using vertical bars.)

The Comparison Problem

Given: tolerance value `c > 0`.

`fp1` "equals" `fp2` whenever **`|fp1 - fp2| < c`**

- Examples (`c` is `0.001`):
  - `fp1 = 10.0` and `fp2 = 12.0`

(Remark: on this slide = is meant in the mathematical sense.)

The Comparison Problem

Given: tolerance value `c > 0`.

`fp1` "equals" `fp2` whenever **`|fp1 - fp2| < c`**

- Examples (`c` is `0.001`):
  - `fp1 = 10.0` and `fp2 = 12.0`

    `|10.0 - 12.0| = 2.0`

(Remark: on this slide = is meant in the mathematical sense.)

The Comparison Problem

Given: tolerance value `c > 0`.

`fp1` "equals" `fp2` whenever **`|fp1 - fp2| < c`**

- Examples (`c` is `0.001`):
  - `fp1 = 10.0` and `fp2 = 12.0`

    `|10.0 - 12.0| = 2.0 > c`

    Thus: **not "equal"**

(Remark: on this slide = is meant in the mathematical sense.)

The Comparison Problem

Given: tolerance value `c > 0`.

`fp1` "equals" `fp2` whenever **`|fp1 - fp2| < c`**

- Examples (`c` is `0.001`):
  - `fp1 = 10.0` and `fp2 = 12.0`

    `|10.0 - 12.0| = 2.0 > c`

    Thus: **not "equal"**
  - `fp1 = 10.0` and `fp2 = 10.000013`

(Remark: on this slide = is meant in the mathematical sense.)

The Comparison Problem

Given: tolerance value `c > 0`.

`fp1` "equals" `fp2` whenever **`|fp1 - fp2| < c`**

- Examples (`c` is `0.001`):
  - `fp1 = 10.0` and `fp2 = 12.0`

    `|10.0 - 12.0| = 2.0 > c`

    Thus: **not "equal"**

  - `fp1 = 10.0` and `fp2 = 10.000013`

    `|10.0 - 10.000013| = 0.000013`

(Remark: on this slide = is meant in the mathematical sense.)

Agenda

- HW #3 feedback
- Floating points
  - Representation
  - Standard
  - Good practice
  - Comparison
- **Functions - PRE and POST Conditions**

Functions - Structure

return type function name argument type argument

```
int main ( int argc, char* argv[] )
{
        cout << "Hello, World!" << endl;
        return 0;
}
```

argument list

function body

- **function name:** name of the function
- **function body:** statements to be executed
- **argument:** variable whose value is **passed into function body** from the outside
- **argument type**: type of the argument
- **return value:** value that is **passed to the outside** after function call
- **return type:** type of the return value (`void` if there is no return value)

Functions - Advantages

- Readability

```
int main()
{
cout << “Please enter two integers: ";
int a,b;
cin >> a >> b;
int res = (a + b) * (a + b);
cout << “The result is: " << res << endl;
return 0;
}
```

Functions - Advantages

- Readability

```
int square_of_sum(int i1, int i2)
{
     int r = i1 + i2;
return r*r;
}

int main()
{
cout << “Please enter two integers: ";
int a,b;
cin >> a >> b;
int res = square_of_sum(a,b);
cout << “The result is: " << res << endl;
return 0;
}
```

Functions - Advantages

- Code re-use

```
int square_of_sum(int i1, int i2)
{
     int r = i1 + i2;
return r*r;
}

int main()
{
cout << “Please enter four integers: ";
int a,b,c,d;
cin >> a >> b >> c >> d;
int res1 = square_of_sum(a,b);
int res2 = square_of_sum(c,d);
cout << “the results are: " << res1 << “,” << res2 << endl;
return 0;
}
```

Functions - Scope

- Scope = validity/visibility of a variable

```
int nonsense(int input)
{
    return input * x;
}

int main()
{
    int x = 3;
    if (x == 3)
    {
        int y = 15;
    }

    int z = nonsense(y);
    std::cout << z << std::endl;
    return 0;
}
```

Errors?

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Informatik
Computer Science

Functions - Scope

- Scope = validity/visibility of a variable

```
int nonsense(int input)
{
    return input * x;
}

int main()
{
    int x = 3;
    if (x == 3)
    {
        int y = 15;
    }

    int z = nonsense(y);
    std::cout << z << std::endl;
    return 0;
}
```

x is **not visible** here;

only visible inside «main» function

y is **not valid** outside { }

we cannot use **y** as an argument

for function «nonsense».

Scope of **x**

Functions - PRE and POST conditions

- Abstraction

```
double cos (double x) {... }

int main()
{
    float x = 0.1;
    float cosx = cos(x);
}
```

- Computation of the value of cos is unknown

Functions - Arguments and Return Value

```
int     square_of_sum(int i1, int i2)



res1 = square_of_sum(   a,        b);
```

- Both are optional

```
void print_message()
{
    std::cout << “This is also a function“ << std::endl;
}
```

- Invocation of a function must have ()

```
print_message();
```

- **Declaration must be before invocation**
- #include files contain function declarations

Functions - PRE and POST conditions

- Abstraction

```
double cos (double x) {... }

int main()
{
    float x = 0.1;
    float cosx = cos(x);
}
```

- Computation of the value of cos is unknown
- x - radians or degrees?

Functions - PRE and POST conditions

- Abstraction

```
// PRE: x represents an angle expressed in radians. // One radian is
equivalent to 180/PI degrees.

// POST: Returns cosine of x in [-1,1].
double cos (double x) {... }

int main()
{
    float x = 0.1;
    float cosx = cos(x);
}
```

- Computation of the value of cos is unknown
- x - radians or degrees?
- Solution: use PRE and POST conditions

Exercise 1

Find **PRE- and POST-conditions** for this function.

```
double f (const double i,
          const double j,
          const double k)
{
  if (i > j)
    if (i > k)
      return i;
    else
      return k;
  else
    if (j > k)
      return j;
    else
      return k;
}
```

(From: Script Exercise 79)

Exercise 1

PRE-Condition:

(not needed)

POST-Condition:

```
// POST: return value is
//       the maximum of
//       i,j and k
```

```
double f (const double i,
          const double j,
          const double k)
{
  if (i > j)
    if (i > k)
      return i;
    else
      return k;
  else
    if (j > k)
      return j;
    else
      return k;
}
```

(From: Script Exercise 79)

Exercise 1

Find **PRE- and POST-conditions** for this function.

```
double g (const int i, const int j)
{
  double r = 0.0;
  for (int k = i; k <= j; ++k)
    r += 1.0 / k;
  return r;
}
```

(From: Script Exercise 79)

Exercise 1

```
double g (const int i, const int j)
{
  double r = 0.0;
  for (int k = i; k <= j; ++k)
    r += 1.0 / k;
  return r;
}
```

PRE-Condition: `// PRE: 0 not contained in {i, ..., j}`

POST-Condition: `// POST: return value is the sum`
`//          1/i + 1/(i+1) + ... + 1/j`

(From: Script Exercise 79)

Exercise 2

Find **3 mistakes** in this program.

```
# include <iostream>

double f (const double x) {
   return g(2.0 * x);
}

bool g (const double x) {
   return x % 2.0 == 0;
}

void h () {
   std::cout << result;
}

int main () {
   const double result = f(3.0);
   h();

   return 0;
}
```

(From: Script Exercise 82)

Exercise 2

Problem 1: g() not yet known

scope of g starts later

```
# include <iostream>

double f (const double x) {
   return g(2.0 * x);
}

bool g (const double x) {
   return x % 2.0 == 0;
}

void h () {
   std::cout << result;
}

int main () {
   const double result = f(3.0);
   h();

   return 0;
}
```

(From: Script Exercise 82)

Exercise 2

Problem 1: g () not yet known

scope of `g` starts later

```
# include <iostream>

double f (const double x) {
   return g(2.0 * x);
}

bool g (const double x) {
   return x % 2.0 == 0;
}

void h () {
   std::cout << result;
}

int main () {
   const double result = f(3.0);
   h();

   return 0;
}
```

Problem 2: Modulo

no modulo for `double`

(From: Script Exercise 82)

Exercise 2

```
# include <iostream>

double f (const double x) {
   return g(2.0 * x);
}

bool g (const double x) {
   return x % 2.0 == 0;
}

void h () {
   std::cout << result;
}

int main () {
   const double result = f(3.0);
   h();

   return 0;
}
```

Problem 1: g() not yet known

scope of g starts later

Problem 2: Modulo

no modulo for double

Problem 3: h() does not «see» result

result is out-of-scope

(From: Script Exercise 82)

Exercise 3

- Fix the **problems** in the following functions.
- Then add suitable **PRE- and POST-conditions**.

```
bool is_even (const int i)
{
   if (i % 2 == 0) return true;
}
```

(From: Script Exercise 80)

Exercise 3

- Problem: just a return value for even inputs

1. Function:

```
bool is_even (const int i)
{
    if (i % 2 == 0) return true;
}
```

(From: Script Exercise 80)

Exercise 3

- Problem: just a return value for even inputs
- Fix: e.g. direct return of `i % 2 == 0`

1. Function:

```
bool is_even (const int i)
{
    if (i % 2 == 0) return true;
}
```

```
bool is_even (const int i)
{
    return (i % 2 == 0);
}
```

(From: Script Exercise 80)

Exercise 3

- Problem: just a return value for even inputs
- Fix: e.g. direct return of `i % 2 == 0`

1. Function:

```
bool is_even (const int i)
{
    if (i % 2 == 0) return true;
}
```

```
bool is_even (const int i)
{
    return (i % 2 == 0);
}
```

PRE-Condition: (not needed)

POST-Condition:

```
// POST: return value is true if and only
//       if i is even
```

(From: Script Exercise 80)

Exercise 3

- Fix the **problems** in the following functions.
- Then add suitable **PRE- and POST-conditions**.

2. Function:

```
double inverse (const double x) {
   double result;
   if (x != 0.0)
        result = 1.0 / x;
   return result;
}
```

(From: Script Exercise 80)

Exercise 3

- Í

2. Function:

```
double inverse (const double x) {
    double result;
    if (x != 0.0)
        result = 1.0 / x;
    return result;
}
```

(From: Script Exercise 80)

Exercise 3

- Problem: no return value for `x=0`
- Fix: `x != 0.0` as PRE-condition (and `assert`)

2. Function:

```
double inverse (const double x) {
    double result;
    if (x != 0.0)
        result = 1.0 / x;
    return result;
}
```

```
// PRE: x != 0.0
// POST: ...
double inverse (const double x) {
    assert(x != 0.0);
    return 1.0 / x;
}
```

(From: Script Exercise 80)

Exercise 3

- Problem: no return value for `x=0`
- Fix: `x != 0.0` as PRE-condition (and `assert`)

2. Function:

```
double inverse (const double x) {
    double result;
    if (x != 0.0)
        result = 1.0 / x;
    return result;
}
```

```
// PRE: x != 0.0
// POST: ...
double inverse (const double x) {
    assert(x != 0.0);
    return 1.0 / x;
}
```

PRE-Condition: `// PRE: x != 0.0`

POST-Condition: `// POST: return value is 1/x`

(From: Script Exercise 80)

Exercise 3

Another solution:

Exercise 3

Another solution:

`else` with special return value

```
double inverse (const double x)
{
    double result;
    if (x != 0.0)
         result = 1.0 / x;
    else
         result = 0.0;
    return result;
}
```

(From: Script Exercise 80)

Exercise 3

Another solution:

`else` with special return value

```
double inverse (const double x)
{
    double result;
    if (x != 0.0)
        result = 1.0 / x;
    else
        result = 0.0;
    return result;
}
```

PRE-Condition: (not needed)

POST-Condition:

```
// POST: return value is 1/x if x!=0.0
//       return value is 0.0 else
```

(From: Script Exercise 80)

Exercise 4

- What is the **output** of this program?
- You can neglect possible over- or underflows for this exercise.

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4

```
i * f(i) * f(f(i))
```

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4

```
i * f(i) * f(f(i))
```

```
f(i)
```

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4

```
i * f(i) * f(f(i))
```

```
i*i
```

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4

```
i * (i*i) * f(f(i))
```

```
i*i
```

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4

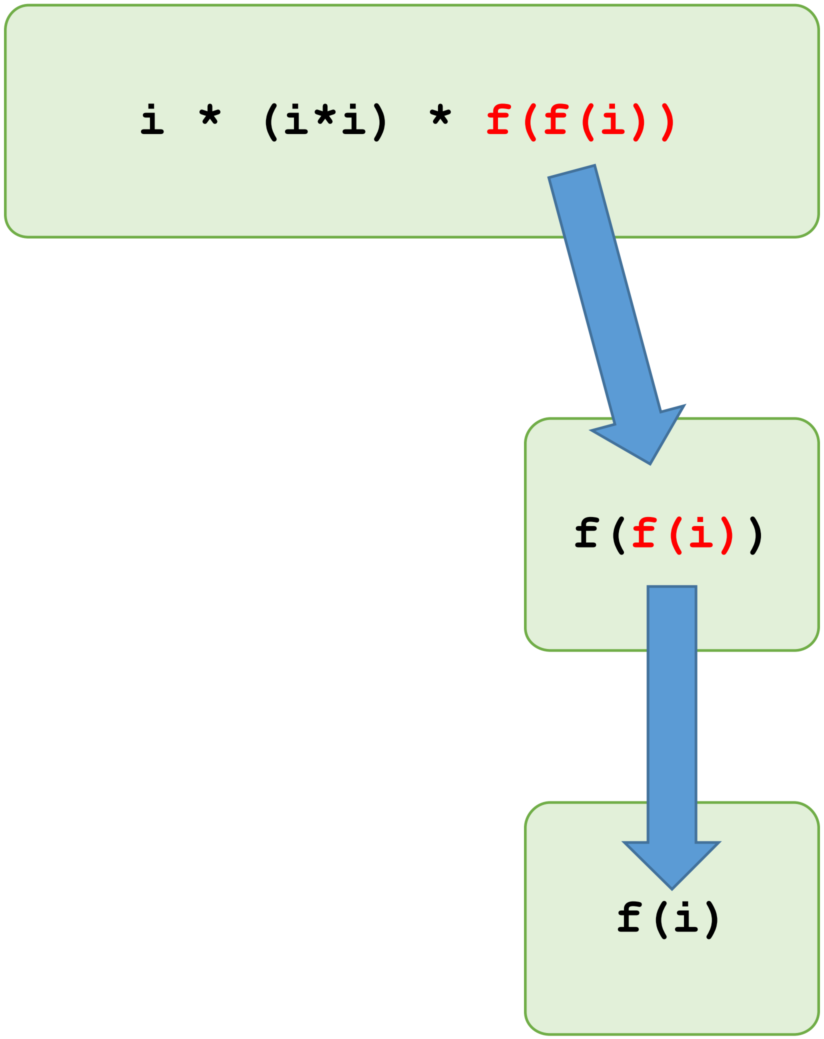
```
i * (i*i) * f(f(i))
```

```
f(f(i))
```

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

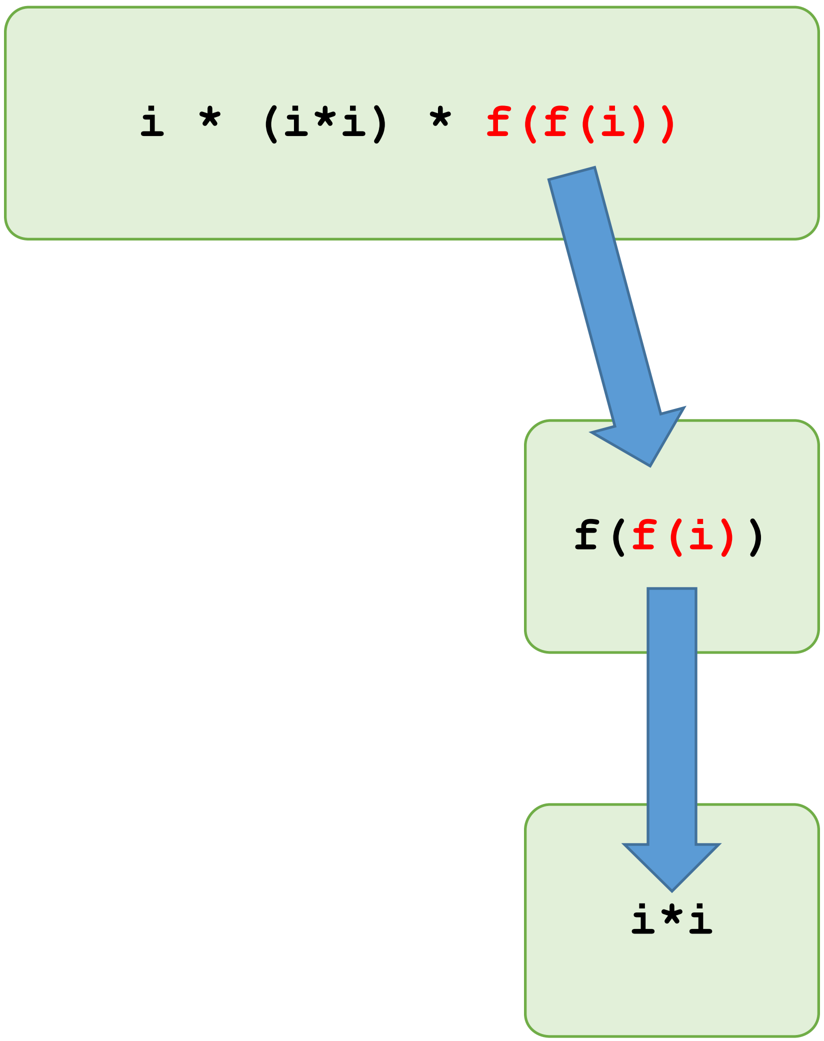
Exercise 4




```cpp
#include <iostream>

int f (const int i) {
      return i * i;
}

int g (const int i) {
      return i * f(i) * f(f(i));
}

void h (const int i) {
      std::cout << g(i) << "\n";
}

int main () {
      int i;
      std::cin >> i;
      h(i);
      return 0;
}
```

(From: Script Exercise 81)

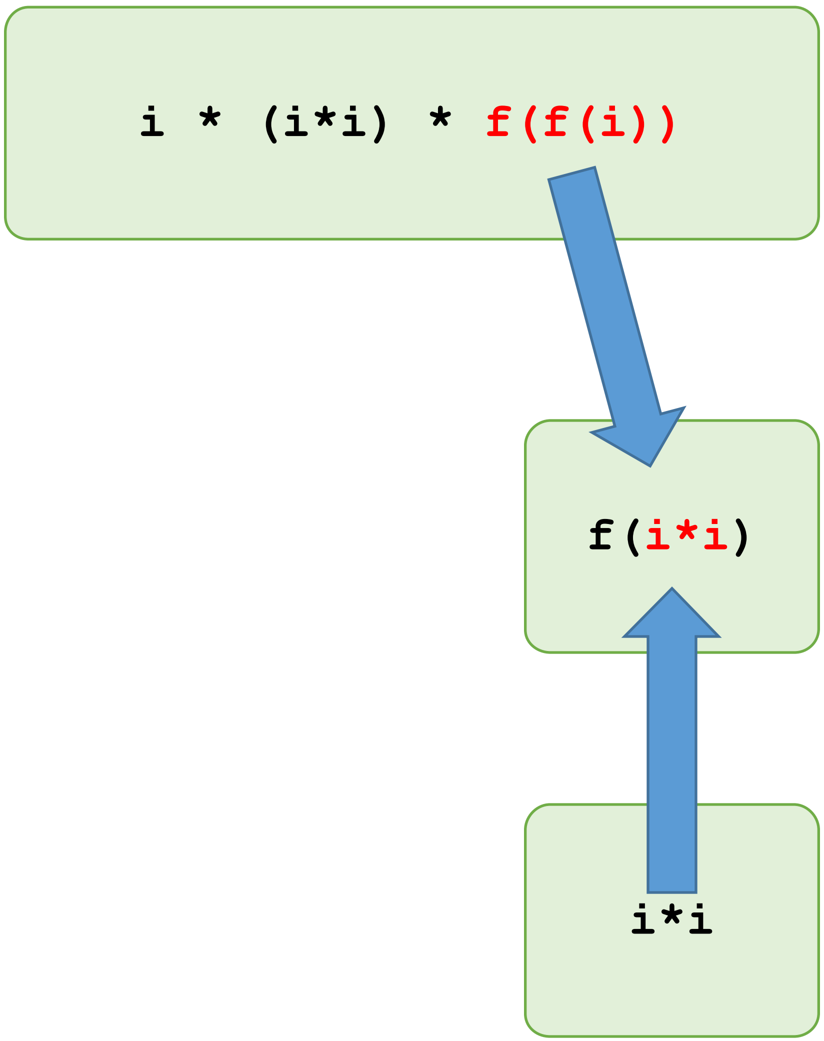
Exercise 4




```cpp
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4




```cpp
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4

```
i * (i*i) * f(f(i))
```

```
f(i*i)
```

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4

```
i * (i*i) * f(f(i))
```

```
(i*i)*(i*i)
```

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4

```
i * (i*i) * ((i*i)*(i*i))
```

```
(i*i)*(i*i)
```

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4

```
i * (i*i) * ((i*i)*(i*i))
```

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

Exercise 4

```
i * (i*i) * ((i*i)*(i*i))
```

This is
i^7

```
#include <iostream>

int f (const int i) {
     return i * i;
}

int g (const int i) {
     return i * f(i) * f(f(i));
}

void h (const int i) {
     std::cout << g(i) << "\n";
}

int main () {
     int i;
     std::cin >> i;
     h(i);
     return 0;
}
```

(From: Script Exercise 81)

The Comparison Problem

Given: tolerance value `c > 0`.

`fp1` "equals" `fp2` whenever **`|fp1 - fp2| < c`**

- Examples (`c` is `0.001`):
  - `fp1 = 10.0` and `fp2 = 12.0`

    `|10.0 - 12.0| = 2.0 > c`

    Thus: **not "equal"**
  - `fp1 = 10.0` and `fp2 = 10.000013`

    `|10.0 - 10.000013| = 0.000013 < c`

    Thus: **"equal"**

(Remark: on this slide = is meant in the mathematical sense.)

Exercise

Complete the following function:

```
// POST: returns true if and only if
//         |x - y| < tol
bool equals (const double x, const double y,
             const double tol) {
   ...
}
```

Exercise

For example:

```
// POST: returns true if and only if
//         |x - y| < tol
bool equals (const double x, const double y,
             const double tol) {
  double diff = x - y;
  if (diff < 0)
    diff *= -1; // absolute value
  return diff < tol;
}
```