9. Functions II

Stepwise Refinement, Scope, Libraries and Standard Functions

Stepwise Refinement

A simple technique to solve complex problems


Stepwise Refinement

- Solve the problem step by step. Start with a coarse solution on a high level of abstraction (only comments and abstract function calls)
- At each step, comments are replaced by program text, and functions are implemented (using the same principle again)
- The refinement also refers to the development of data representation (more about this later).
- If the refinement is realized as far as possible by functions, then partial solutions emerge that might be used for other problems.
- Stepwise refinement supports (but does not replace) the structural understanding of a problem.

Example Problem

Find out if two rectangles intersect!
Coarse Solution

(include directives omitted)

```c
int main()
{
    // input rectangles
    // intersection?
    // output solution
    return 0;
}
```

Refinement 1: Input Rectangles

Width \( w \) and height \( h \) may be negative.

```c
int main()
{
    std::cout << "Enter two rectangles [x y w h each] \n";
    int x1, y1, w1, h1;
    std::cin >> x1 >> y1 >> w1 >> h1;
    int x2, y2, w2, h2;
    std::cin >> x2 >> y2 >> w2 >> h2;
    // intersection?
    // output solution
    return 0;
}
```
Refinement 2: Intersection? and Output

```cpp
int main()
{
    // input rectangles ✓
    bool clash = rectangles_intersect (x1,y1,w1,h1,x2,y2,w2,h2);
    if (clash)
        std::cout << "intersection!\n";
    else
        std::cout << "no intersection!\n";
    return 0;
}
```

Refinement 3: Intersection Function...

```cpp
bool rectangles_intersect (int x1, int y1, int w1, int h1,
int x2, int y2, int w2, int h2)
{
    return false; // todo
}
```

Refinement 3: Intersection Function... with PRE and POST

```cpp
// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles,
// where w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1) and
//       (x2, y2, w2, h2) intersect
bool rectangles_intersect (int x1, int y1, int w1, int h1,
int x2, int y2, int w2, int h2)
{
    return false; // todo
}
```

Function main ✓
Refinement 4: Interval Intersection

Two rectangles intersect if and only if their $x$ and $y$-intervals intersect.

\[
(x_1, y_1) \quad w_1 \quad h_1 \\
(x_2, y_2) \quad w_2 \quad h_2
\]

\[
[x_1, x_1 + w_1] \\
[y_1, y_1 + h_1]
\]

\[
[x_2, x_2 + w_2] \\
[y_2, y_2 + h_2]
\]

Refinement 4: Interval Intersections

// PRE: \((x_1, y_1, w_1, h_1), (x_2, y_2, w_2, h_2)\) are rectangles, where
// \(w_1, h_1, w_2, h_2\) may be negative.
// POST: returns true if \((x_1, y_1, w_1, h_1), (x_2, y_2, w_2, h_2)\) intersect
bool rectangles_intersect (int x1, int y1, int w1, int h1, int x2, int y2, int w2, int h2)
{
    return intervals_intersect (x1, x1 + w1, x2, x2 + w2)
        && intervals_intersect (y1, y1 + h1, y2, y2 + h2);
}

Refinement 4: Interval Intersections

// PRE: \([a1, b1], [a2, b2]\) are (generalized) intervals,
// with \([a,b] := [b,a]\) if \(a>b\)
// POST: returns true if \([a1, b1],[a2, b2]\) intersect
bool intervals_intersect (int a1, int b1, int a2, int b2)
{
    return false; // todo
}

Refinement 5: Min and Max

// PRE: \([a1, b1], [a2, b2]\) are (generalized) intervals,
// with \([a,b] := [b,a]\) if \(a>b\)
// POST: returns true if \([a1, b1],[a2, b2]\) intersect
bool intervals_intersect (int a1, int b1, int a2, int b2)
{
    return max(a1, b1) >= min(a2, b2)
        && min(a1, b1) <= max(a2, b2); ✓
}
Refinement 5: Min and Max

// POST: the maximum of x and y is returned
int max (int x, int y)
{
    if (x>y) return x; else return y;
}

// POST: the minimum of x and y is returned
int min (int x, int y)
{
    if (x<y) return x; else return y;
}

// PRE: [a1, b1], [a2, h2] are (generalized) intervals,
// with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
bool intervals_intersect (int a1, int b1, int a2, int b2)
{
    return std::max(a1, b1) >= std::min(a2, b2)
        && std::min(a1, b1) <= std::max(a2, b2);
}

int main ()
{
    std::cout << "Enter two rectangles [x y w h each]";
    int x1, y1, w1, h1;
    std::cin >> x1 >> y1 >> w1 >> h1;
    int x2, y2, w2, h2;
    std::cin >> x2 >> y2 >> w2 >> h2;
    bool clash = rectangles_intersect (x1,y1,w1,h1,x2,y2,w2,h2);
    if (clash)
        std::cout << "intersection!\n";
    else
        std::cout << "no intersection!\n";
    return 0;
}
Where can a Function be Used?

```cpp
#include<iostream>

int main()
{
    std::cout << f(1);  // Error: f undeclared
    return 0;
}

int f (int i) // Scope of f starts here
{
    return i;
}
```

Scope of a Function

- is the part of the program where a function can be called
- is defined as the union of all scopes of its declarations (there can be more than one)

`declaration` of a function: like the definition but without `{...}`.

```cpp
double pow (double b, int e);
```

This does not work...

```cpp
#include<iostream>

int main()
{
    std::cout << f(1);  // Error: f undeclared
    return 0;
}

int f (int i) // Scope of f starts here
{
    return i;
}
```

...but this works!

```cpp
#include<iostream>

int main()
{
    std::cout << f(1);  // Gueltigkeitsbereich von f ab hier
    return 0;
}

int f (int i)
{
    return i;
}
```
Forward Declarations, why?

Functions that mutually call each other:

```c
int g(...); // forward declaration
int f(...) // f valid from here
{
g(...) // ok
}
int g(...)
{
f(...) // ok
}
```

Reusability

- Functions such as `rectanges` and `pow` are useful in many programs.
- “Solution”: copy-and-paste the source code
- Main disadvantage: when the function definition needs to be adapted, we have to change all programs that make use of the function

Level 1: Outsource the Function

```c
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e)
{
    double result = 1.0;
    if (e < 0) { // b^e = (1/b)^(−e)
        b = 1.0/b;
        e = −e;
    }
    for (int i = 0; i < e; ++i)
        result *= b;
    return result;
}
```

Level 1: Include the Function

```c
// Prog: callpow2.cpp
// Call a function for computing powers.
#include <iostream>
#include "math.cpp"

int main()
{
    std::cout << pow( 2.0, −2) << "\n";
    std::cout << pow(1.5, 2) << "\n";
    std::cout << pow(5.0, 1) << "\n";
    std::cout << pow(−2.0, 9) << "\n";

    return 0;
}
```
Disadvantage of Including

- #include copies the file (math.cpp) into the main program (callpow2.cpp).
- The compiler has to (re)compile the function definition for each program.
- This can take long for many and large functions.

Level 2: Separate Compilation

of math.cpp independent of the main program:

```cpp
double pow(double b, int e)
{
    ...
}
```

g++ -c math.cpp

math.cpp

math.o

Level 2: Separate Compilation

Declaration of all used symbols in so-called header file.

```cpp
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e);
```

#include <iostream>
#include "math.h"

```cpp
int main()
{
    std::cout << pow(2,-2) << "\n";
    return 0;
}
```

callpow3.cpp

callpow3.o

Level 2: Separate Compilation

of the main program, independent of math.cpp, if a declaration of math is included.

```cpp
#include <iostream>
#include "math.h"
int main()
{
    std::cout << pow(2,-2) << "\n";
    return 0;
}
```

callpow3.cpp

callpow3.o
The linker unites...

... what belongs together

<table>
<thead>
<tr>
<th>001110101100101010</th>
<th>001110101100101010</th>
</tr>
</thead>
<tbody>
<tr>
<td>math.o</td>
<td>callpow3.o</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00010</th>
<th>Funktion pow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000111010</td>
<td></td>
</tr>
<tr>
<td>0101011011010001</td>
<td></td>
</tr>
<tr>
<td>1001011111001010</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00010</th>
<th>Funktion main</th>
</tr>
</thead>
<tbody>
<tr>
<td>111100001101010001</td>
<td></td>
</tr>
<tr>
<td>11111110</td>
<td></td>
</tr>
<tr>
<td>1000111010</td>
<td></td>
</tr>
<tr>
<td>010101101011010001</td>
<td></td>
</tr>
<tr>
<td>1001011111001010</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>001110101100101010</th>
<th>00010</th>
<th>Funktion main</th>
</tr>
</thead>
<tbody>
<tr>
<td>111100001101010001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000111010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010101101011010001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001011111001010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

math.o + callpow3.o = Executable callpow

Availability of Source Code?

Observation

math.cpp (source code) is not required any more when the math.o (object code) is available.

Many vendors of libraries do not provide source code. Header files then provide the only readable informations.

„Open Source” Software

- Source code is generally available.
- Only this allows the continued development of code by users and dedicated “hackers”.
- Even in commercial domains, “open source” gains ground.
- Certain licenses force naming sources and open development. Example GPL (GNU General Public License)
- Known open source software: Linux (operating system), Firefox (browser), Thunderbird (email program)…
Libraries

- Logical grouping of similar functions
  - pow
  - exp
  - log
  - sin
  - math.h
  - math.cpp

Name Spaces...

// ifeemath.h
// A small library of mathematical functions
namespace ifee {
  // PRE: e >= 0 || b != 0.0
  // POST: return value is b^e
  double pow (double b, int e);
  ....
  double exp (double x);
  ...
}

...Avoid Name Conflicts

#include <cmath>
#include "ifeemath.h"

int main()
{
  double x = std::pow (2.0, -2); // <cmath>
  double y = ifee::pow (2.0, -2); // ifeemath.h
}

Name Spaces / Compilation Units

In C++ the concept of separate compilation is independent of the concept of name spaces
In some other languages, e.g. Modula / Oberon (partially also for Java) the compilation unit can define a name space.
Functions from the Standard Library

- help to avoid re-inventing the wheel (such as with `ifee::pow`);
- lead to interesting and efficient programs in a simple way;
- guarantee a quality standard that cannot easily be achieved with code written from scratch.

Prime Number test with `sqrt`

$n \geq 2$ is a prime number if and only if there is no $d$ in $\{2, \ldots, n-1\}$ dividing $n$.

```cpp
unsigned int d;
for (d=2; n % d != 0; ++d);
```

This works because `std::sqrt` rounds to the next representable `double` number (IEEE Standard 754).

Other mathematical functions (`std::pow`,...) are almost as exact in practice.

Prime Number Test with `sqrt`

```cpp
// Test if a given natural number is prime.
#include <iostream>
#include <cassert>
#include <cmath>

int main ()
{
    // Input
    unsigned int n;
    std::cout << "Test if n>1 is prime for n =? ";
    std::cin >> n;
    assert (n > 1);

    // Computation: test possible divisors d up to sqrt(n)
    unsigned int bound = std::sqrt(n);
    unsigned int d;
    for (d = 2; d <= bound && n % d != 0; ++d);

    // Output
    if (d <= bound)
        // d is a divisor of n in \{2,...,\lfloor \sqrt{n}\rfloor\}
        std::cout << n << " = " << d << " ∗ " << n / d << ".
    else
        // no proper divisor found
        std::cout << n << " is prime.
    return 0;
}
```
void swap (int x, int y) {
    int t = x;
    x = y;
    y = t;
}

int main(){
    int a = 2;
    int b = 1;
    swap (a, b);
    assert (a==1 && b==2); // fail! 😞
}

// POST: values of x and y are exchanged
void swap (int& x, int& y) {
    int t = x;
    x = y;
    y = t;
}

int main(){
    int a = 2;
    int b = 1;
    swap (a, b);
    assert (a==1 && b==2); // ok! 😊
}

Sneak Preview: Reference Types

- We can enable functions to change the value of call arguments.
- Not a new concept for functions but rather a new class of types

Reference types (e.g. `int&`)