Санкт-Петербургский
государственный
университет

Structuring Code

Aleksandr Logunov

SPbSU

Objectives

- Simplifying the process of debugging.
- Making your code more understandable.

Managing dependencies

When your mobile phone or laptop doesn’t turn on, the problem can be in:

- battery
- motherboard
- one of cables
- power button
- ...

Managing dependencies

knowledge of structure + disassembling

↓

quick dealing with problems

Managing dependencies

The same holds for programming:

- bad understanding of code and its structure → need to analyze each line
- good understanding, structuring → possibility of considering each part of code separately

Managing dependencies

Example task: you have information about n people.

Your goals are:

1. Compute the number of people employed.
2. Compute the sum of ages of all people.

Managing dependencies

```
person a[n];
int employed = 0;
for (int i = 0; i < n; i++) {
    read(a[i]);
    if (a[i].isEmployed) employed++;
}
int sumAges = 0; write(employed);
for (int i = 0; i < n; i++)
    sumAges += a[i].age;
write(sumAges);
```

Managing dependencies

```
person a[n];
int employed = 0;
for (int i = 0; i < n; i++) {
    read(a[i]);
    if (a[i].isEmployed) employed++;
}
int sumAges = 0; write(employed);
for (int i = 0; i < n; i++)
    sumAges += a[i].age;
write(sumAges);
```

Non-structured code

Managing dependencies

```
person a[n];
int employed = 0;
int sumAges = 0;
for (int i = 0; i < n; i++)
    read(a[i]);
for (int i = 0; i < n; i++)
    if (a[i].isEmployed) employed++;
for (int i = 0; i < n; i++)
    sumAges += a[i].age;
write(employed);
write(sumAges);
```

Well structured code

Managing dependencies

```
person a[n];
int employed = 0;                          }
int sumAges = 0;                           }  initData()
for (int i = 0; i < n; i++)                }
     read(a[i]);                           }  readData()
for (int i = 0; i < n; i++)                }
     if (a[i].isEmployed) employed++;      }  countEmployed()
for (int i = 0; i < n; i++)                }
     sumAges += a[i].age;                  }  countSumAges()
write(employed);                           }
write(sumAges);                            }  writeAnswers()
```

Managing dependencies

```
person a[n];
int employed, sumAges;
initData();
readData();
countEmployed();
countSumAges();
writeAnswers();
```

Managing dependencies

Observations:

- code is more readable when logical blocks of code don’t mix;

Managing dependencies

Observations:

- code is more readable when logical blocks of code don’t mix;
- number of blocks depends on a lot of factors;

Managing dependencies

Observations:

- code is more readable when logical blocks of code don’t mix;
- number of blocks depends on a lot of factors;
- often solution is not unique;

Managing dependencies

Observations:

- code is more readable when logical blocks of code don’t mix;
- number of blocks depends on a lot of factors;
- often solution is not unique;
- sometimes code is not working just because its idea is not correct.

Readability

Consider the following line of code:

Readability

Consider the following line of code:

```
p = q * 60 + r;
```

What do you think this line does?

Readability

Consider the following line of code:

```
p = q * 60 + r;
```

What do you think this line does?

Answer: computes time (in minutes) from the midnight.

Readability

Consider the following line of code:

```
read(q, r);
p = q * 60 + r;
write(“Time = “, p);
```

Certainly, other lines could help you.
But why not to make the line of code readable on its own?

Readability

Consider the following line of code:

```
time = hours * 60 + minutes;
```

Now, it is immediate what this line does.

Invariants and conditions

Task: compute the sum of all integers i such that $1 \leq i^3 < 5000$.

Invariants and conditions

Task: compute the sum of all integers i such that $1 \leq i^3 < 5000$.

Solution: use loops!

Pre-condition loop

```
sum = 0;
i = 1;
while (i * i * i < 5000) {
        sum = sum + i;
        i = i + 1;
}
```

Property: pre-condition states what must be true before **entering** a loop.

Post-condition loop

```
sum = 0;
i = 1;
do {
            sum = sum + i;
            i = i + 1;
} while (i * i * i < 5000);
```

Property: post-condition states what must be true before **continuing** a loop (so at least one iteration is performed!).

Loop invariants

What happens if 5000 → 1?

There are no numbers i such that $1 \le i^3 < 1$, so the answer equals 0.

Loop invariants

What happens if 5000 → 1?

There are no numbers i such that $1 \leq i^3 < 1$,
so the answer equals 0.

Pre-condition: sum = **0**
Post-condition: sum = **1**

Loop invariants

The second program fails because **loop invariant** is violated.

Loop invariant: assertion that is always preserved
in the loop body.

Loop invariants

In our task it makes sense to consider two invariants:

1. $1 \le i^3 < 5000$
2. sum equals sum of all i’s seen so far.

What if 5000 → 1?

Loop invariants

Type – Invariant	Pre-cond	Post-cond
$1 \le i^2 < 1$	OK	FAIL
$sum = \dots$	OK	OK

Loop invariants: conclusion

We should set invariants carefully and keep an eye on them.

Summary

We wanted to:

- understand code better;
- simplify debugging.

Ways to do it:

- divide your code into blocks;
- give meaningful names to variables;
- preserve invariants.