Imperative programming

Statements

Tamás Kozsik et al

Eötvös Loránd University

October 11, 2022

ELTE
EÖTVÖS LORÁND
TUDOMÁNYEGYETEM

Table of contents

Statements

- Simple statements
  - Variable declaration
  - Assignment
  - Function call
  - Returning from functions
- Control structures
  - Sequence
  - Branching
  - Loops

Variable declarations

- Create all variables before their first usage
- It is worth declaring here

```
double m;
int n = 3;
char cr = '\r', lf = '\n';
int i = 1, j;
int u, v = 3;
```

Expression statement

- Expression evaluation (with side-effect)
- Typical example: assignments

```
<statement> ::= <expression>;
              | ...
```

```
n = 1;
x *= y;
c++;
n > 0 ? --n : ++n;      /* Not idiomatic! */
```

Return

- There can be multiple `return` statements in a function
- No `return` ≡ empty `return` (`void`)

```
int twice(int x) {
  return 2 * x;
}

bool isPrime(int x) {
  for (int i = 2; i <= x / 2; ++i)
    if (x % i == 0)
      return false;
  return true;
}

printf("42 twice: %d\n", twice(42));

if (isPrime(11))
  printf("11 is a prime\n");
```

Function call

- Declared return type, corresponding `return` statement(s)
- Only side-effect: `void` return type, empty `return`

Pure function

```
unsigned long fact(int n)
{
  unsigned long result = 1L;
  int i;
  for (i = 2; i <= n; ++i)
    result *= i;
  return result;
}
```

Only side-effect

```
void printSquares(int n)
{
  int i;
  for (i = 1; i <= n; ++i) {
    printf("%d\n",i*i);
  }
  return;   /* Can be omitted */
}
```

Mixed behavior

```
printf("%d\n", printf("%d\n", 42));
```

Empty statement

A statement doing nothing

```
;
```

Infinite loop

```
int i = 0;
while (i < 10);
  printf("%d\n");
```

First negative number in an array

```
int nums[] = {3, 6, 1, 45, -1, 4};

for (int i = 0; i < 6 && nums[i] < 0; ++i);

for (int i = 0; i < 6 && nums[i] < 0; ++i) {
}
```

Control structures

- Sequence
- Branching
- Loops
  - Testing – Forward testing – Backward testing
  - Stepping
- Non-structured transfer of control
  - `return`
  - `break`
  - `continue`
  - `goto`

Structured programming

- Sequence, branching, loops
- Every algorithm can be implemented by these
- More readable, easier to reason about their correctness
- Possibly use only these!

Sequence

- Writing statements after each other
- Semicolon
- Compound statement

```
<statement>      ::= {<statement-list>}
                   | ...
<statement-list> ::= ""
                   | <statement> <statement-list>
```

Body of control structures

- One statement
- Can be a compound statement

```
int arr[10];

for (int i = 0; i < 10; ++i)
  arr[i] = i + 1;

int pos = 0;
while (pos < 10)
{
  printf("%d\n", arr[pos]);
  ++pos;
}
```

Branching

- `if - else` structure
- `else` branch is optional

Idióma

```
if (x > 0)
  y = x;
else if (y > 0)
  x = y;
else
  x = y = x * y;
```

Conventional indentation

```
if (x > 0)
  y = x;
else
  if (y > 0)
    x = y;
  else
    x = y = x * y;
```

Curly braces are useful

Idióma

```
if (x > 0) {
  y = x;
} else if (y > 0) {
  x = y;
} else {
  x = y = x * y;
}
```

Conventional indentation

```
if (x > 0) {
  y = x;
} else {
  if (y > 0) {
    x = y;
  } else {
    x = y = x * y;
  }
}
```

Dangling else

Wrote this

```
if (x == 1)
  if (y == 2)
    printf("hello");
else
  printf("world");
```

Means this

```
if (x == 1)
  if (y == 2)
    printf("hello");
  else
    printf("world");
```

Wanted this

```
if (x == 1) {
  if (y == 2)
    printf("hello");
} else
  printf("world");
```

See...

goto-fail (Apple) link!

switch-case-break statement

Based on integer type, compile-time constants

```
switch (dayOf(date()))
{
  case 0:  strcpy(name, "Sunday");    break;
  case 1:  strcpy(name, "Monday");    break;
  case 2:  strcpy(name, "Tuesday");   break;
  case 3:  strcpy(name, "Wednesday"); break;
  case 4:  strcpy(name, "Thursday");  break;
  case 5:  strcpy(name, "Friday");    break;
  case 6:  strcpy(name, "Saturday");  break;
  default: strcpy(name, "illegal value");
}
```

Overflow

```
switch (month)
{
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12: days = 31;
           break;
  case 2:  days = 28 + (isLeapYear(year) ? 1 : 0);
           break;
  default: days = 30;
}
```

Non-trivial overflow

```
switch (getKey())
{
  case 'q': jump = 1;
  case 'a': moveLeft();
            break;
  case 'e': jump = 1;
  case 's': moveRight();
            break;
  case ' ': openDoor();
}
```

`switch` and structural programming

Considered as structural

- `break` at the end of each branch
- Same statements for multiple branches

Non-structural construction

- Non-trivial overflow
- E.g. no `break` at all

while loop

```
<while-stmt> ::= while (<expression>) <statement>
```

```
while (i > 0)
{
  printf("%d\n", i);
  --i;
}
```

```
while (i > 0)
  printf("%d\n", i--);
```

do-while loop

<do-while-stmt> ::= do <statement> while (<expression>);

Typical example

```
char command[LENGTH];
do {
  read_data(command);
  if (strcmp(command, "START") == 0) {
    printf("start\n");
  } else if (strcmp(command, "STOP") == 0) {
    printf("stop\n");
  }
} while (strcmp(command, "QUIT") != 0);
```

Transformations

Under what circumstances is it true?

`do` σ `while (`ε`);` $\equiv$ σ `while (`ε`)` σ

Under what circumstances is it true?

`do` σ `while (`ε`);`

$\equiv$

`int new_var = 1; ... while (new_var) {` σ`; new_var =` ε`; }`

Previous example transformed

```
char command[LENGTH];
int new_var = 1;
...
while (new_var) {
  read_data(command);
  if (strcmp(command, "START") == 0 ) {
    ...
  } else if (strcmp(command, "STOP") == 0 ) {
    ...
  }
  new_var = (strcmp(command, "QUIT") != 0);
}
```

Refactoring

```
char command[LENGTH];
int stay_in_loop = 1;
...
while (stay_in_loop) {
  read_data(command);
  if (strcmp(command, "START") == 0) {
    ...
  } else if (strcmp(command, "STOP") == 0) {
    ...
  } else if (strcmp(command, "QUIT") == 0) {
    stay_in_loop = 0;
  }
}
```

for loop

```
<for-stmt> ::= for (<optional-expression>;
                    <optional-expression>;
                    <optional-expression>)
                <statement>
<optional-expression> ::= "" | <expression>
```

(initialization; condition; iteration)

Example

```
unsigned char c;
for (c = 0; c < 256; ++c)
{
  printf("%d\t%c\n", c, c);
}
```

Infinite loop

```
while (1) ...

for (;;) ...
```

Transformations

Can always be done

`while (`ε`)` σ $\Rightarrow$ `for (;` ε`;)` σ

Under what circumstances is it true?

`for (`ι`;` ε`;` λ`)` σ $\Rightarrow$ ι`; while (`ε`) {` σ`;` λ`; }`

Control structures of structured programming

- Compound statement
- Branching
  - `if-else`
  - `switch-case-break`
- Loops
  - Testing loops
    - Forward testing (`while`)
    - Backward testing (`do-while`)
  - Stepping (`for`)

Non structured transfer of control

- `return`
- `break`
- `continue`
- `goto`

break statement

- Exits the innermost loop (or `switch`)

```
for (int i = 0; i < 10; ++i)
{
  if (i == 5)
    break;

  printf("%d", i);      /* 0 1 2 3 4 */
}
```

continue statement

- Finishes the innermost loop body
- Executes iteration step at `for` loop

```
for (int i = 0; i < 10; ++i)
{
  if (i == 5)
    continue;

  printf("%d", i);      /* 0 1 2 3 4 6 7 8 9 */
}
```

goto statement

- Jumps to a given label inside a function

```
<statement> ::= ...
              | goto <label>
              | <label>: <statement>
<label> ::= <identifier>
```

Find a zero element in a matrix

goto-val

```
int matrix[SIZE][SIZE];
...
int found = 0;
int i, j;
for (i=0; i<SIZE; ++i) {
  for (j=0; j<SIZE; ++j) {
    if (matrix[i][j] == 0) {
      found = 1;
      goto end_of_search;
    }
  }
}
/* --i; --j; */
end_of_search:;
```

Szabályosan

```
int matrix[SIZE][SIZE];
...
int found = 0;
int i= -1, j;
while (i < SIZE - 1 && !found) {
  j = -1;
  while (j < SIZE - 1 && !found) {
    if (matrix[i + 1][j + 1] == 0) {
      found = 1;
    }
    j++;
  }
  i++;
}
```