

Imperative programming

4. Statements

Zoltán Porkoláb

Statements

- Null statement -- skip
- Expression statement
- Compound statement (block)

```
void f(void)
{
    // function body is always a block
    i=0; // expression statement
    while ( i < 10 ) // while statement
    {
        // compound statement
        if ( i % 2 ) // if-else statement
            ; // null statement
        else
            printf("%d\n",i); // expression statement
        ++i; // expression statement
    }
}
```

Statements

- Null statement -- skip
- Expression statement
- Compound statement (block)

```
void f(void)
{
    // function body is always a block
    i=0; // expression statement
    while ( i < 10 ) { // while statement + compound statement
        if ( i % 2 ) // if-else statement
            ; // null statement
        else
            printf("%d\n",i); // expression statement
        ++i; // expression statement
    }
}
```

if-else statement

- All non-zero conditions are true
- Non-null pointers are true
- Dangling else belongs the closes if

```
void f(int x, int y)
{
    if ( x < 10 )
        if ( y > 5 )
            printf("x < 10 and y > 5\n");
    else
        printf("x < 10 and y <= 5\n");
}
```

if-else statement

- All non-zero conditions are true
- Non-null pointers are true
- Dangling else belongs to the closes if

```
void f(int x, int y)
{
    if ( x < 10 )
        if ( y > 5 )
            printf("x < 10 and y > 5\n");
        else
            printf("x < 10 and y <= 5\n");
}
```

if-else statement

- All non-zero conditions are true
- Non-null pointers are true
- Dangling else belongs to the closes if

```
void f(int x, int y)
{
    if ( x < 10 )
        if ( y > 5 )
            printf("x < 10 and y > 5\n");
        else
            printf("x < 10 and y <= 5\n");
}
```

if-else statement

- All non-zero conditions are true
- Non-null pointers are true
- Dangling else belongs the closes if

```
void f(int x, int y)
{
    if ( x < 10 ) {
        if ( y > 5 ) {
            printf("x < 10 and y > 5\n");
        }
        else {
            printf("x < 10 and y <= 5\n");
        }
    }
}
```

if-else statement

- All non-zero conditions are true
- Non-null pointers are true
- Dangling else belongs the closes if

```
void f(int x, int y)
{
    if ( x < 10 )
    {
        if ( y > 5 )
        {
            printf("x < 10 and y > 5\n");
        }
        else
        {
            printf("x < 10 and y <= 5\n");
        }
    }
}
```


if-else statement

```
if ( x < 10  &&  y > 5 ) {
    printf("x < 10 and y > 5");
}
else {
    if ( x < 10  &&  y <= 5 ) {
        printf("x < 10 and y <= 5");
    }
    else {
        if ( x >= 10  &&  y > 5 ) {
            printf("x >= 10 and y >5");
        }
        else {
            if ( x >= 10  &&  y <= 5 ) {
                printf("x >= 10 and y <= 5");
            }
            else {
                printf("impossible");
            }
        }
    }
}
}
```

if-else statement

```
if ( x < 10  &&  y > 5 ) {  
    printf("x < 10 and y > 5");  
}  
else if ( x < 10  &&  y <= 5 ) {  
    printf("x < 10 and y <= 5");  
}  
else if ( x >= 10  &&  y > 5 ) {  
    printf("x >= 10 and y >5");  
}  
else if ( x >= 10  &&  y <= 5 ) {  
    printf("x >= 10 and y <= 5");  
}  
else {  
    printf("impossible");  
}
```

switch statement

- Selection on a value (“case label”)
- All labels must be unique
- Control fall through the next statement unless **break** is used
- The break exits switch and continues on the statement after switch
- Optional default statement

```
void f(int i)
{
    switch ( i % 2 )
    {
        case 0: printf("even\n");    break;
        case 1: printf("odd\n");    break;
    }
}
```

switch statement

```
void print_day( int day_of_week )
{
    switch ( day_of_week )
    {
        case 1: printf( "Sunday" );      break;
        case 2: printf( "Monday" );     break;
        case 3: printf( "Tuesday" );    break;
        case 4: printf( "Wednesday" );  break;
        case 5: printf( "Thursday" );   break;
        case 6: printf( "Friday" );     break;
        case 7: printf( "Saturday" );   break;
    }
}
```

switch statement

```
void print_day( int day_of_week )
{
    switch ( day_of_week )
    {
        default: printf("Bad value"); break;
        case 1: printf("Sunday"); break;
        case 2: printf("Monday"); break;
        case 3: printf("Tuesday"); break;
        case 4: printf("Wednesday"); break;
        case 5: printf("Thursday"); break;
        case 6: printf("Friday"); break;
        case 7: printf("Saturday"); break;
    }
}
```

switch statement

- ```
void print_day(int day_of_week)
{
 switch (day_of_week)
 {
 default: printf("Bad value"); break;
 case 2: printf("Monday"); break;
 case 3: printf("Tuesday"); break;
 case 4: printf("Wednesday"); break;
 case 5: printf("Thursday"); break;
 case 6: printf("Friday"); break;
 case 1:
 case 7: printf("Weekend"); break;
 }
}
```

# switch statement

```
void print_day(int day_of_week)
{
 switch (day_of_week)
 {
 default: printf("Bad value"); break;
 case 2: printf("Monday"); break;
 case 3: printf("Tuesday"); break;
 case 4: printf("Wednesday"); break;
 case 5: printf("Thursday"); break;
 case 6: printf("Friday"); break;
 case 1: [[fallthrough]];
 case 7: printf("Weekend"); break;
 }
}
```

# while statement

- Most simple loop with **pre-testing** the condition
- All non-zero values are true
- Non-null pointer values are true
- The **break** statement exits the loop and continues on the statement after
- The break jumps out only one level, from the innermost loop
- The **continue** statement continues to the end of the loop body

```
int i = 0;
while (i < 10)
{
 if (i % 2)
 {
 printf("%d is odd\n", i);
 }
 ++i;
}
```



# while statement

```
void print_first_odd(int t[], int sz)
{
 int i = 0;

 while (i < sz)
 {
 if (t[i] % 2)
 {
 printf("the first odd is %d\n", t[i]);
 }
 ++i;
 }
}
```

# while statement

```
void print_first_odd(int t[], int sz)
{
 int i = 0;
 int first_found = 0;
 while (i < sz && !first_found)
 {
 if (t[i] % 2)
 {
 printf("the first odd is %d\n", t[i]);
 first_found = 1;
 }
 ++i;
 }
}
```

# while statement

```
void print_first_odd(int t[], int sz)
{
 int i = 0;

 while (i < sz)
 {
 if (t[i] % 2)
 {
 printf("the first odd is %d\n", t[i]);
 break;
 }
 ++i;
 }
}
```

# while statement

```
#include <stdio.h>

int main() // copy standard input to standard output
{
 int ch;
 ch = getchar(); // read the next char from stdin
 while (EOF != ch) // EOF is a special value from stdio.h
 {
 putchar(ch); // print ch to stdout
 ch = getchar(); // read the next char from stdin
 }
 return 0;
}
```

# while statement

```
#include <stdio.h>

int main() // copy standard input to standard output
{
 int ch;

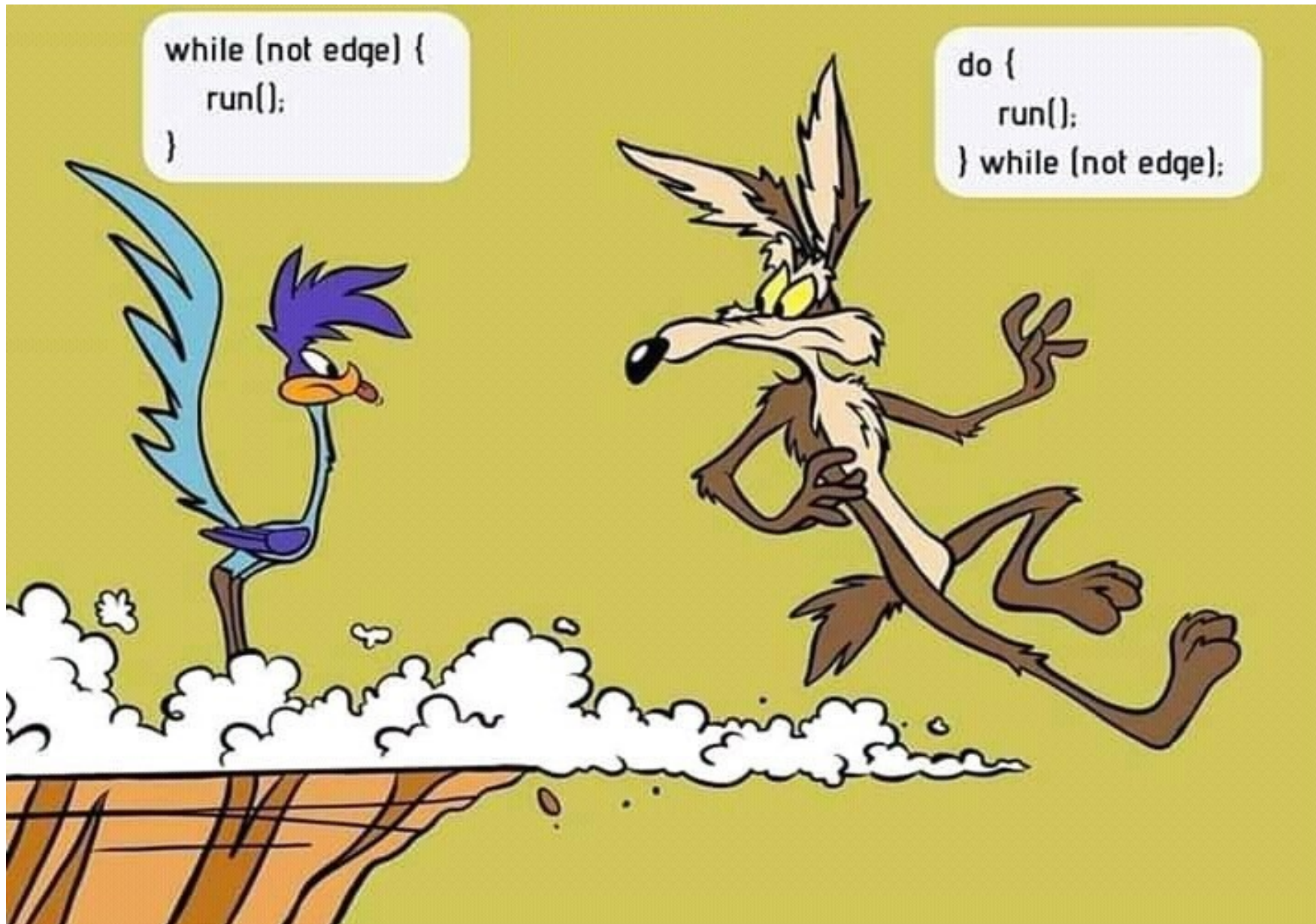
 while (EOF != (ch = getchar())) // read ch until EOF
 {
 putchar(ch); // print ch to stdout
 }
 return 0;
}
```

# do-while statement

- Loop with **post-testing** the condition
- Always execute the loop body at least once
- Otherwise, same as the **while** statement

```
int i = 0;
do
{
 if (i % 2)
 {
 printf("%d is odd\n", i);
 }
 ++i;
}
while (i < 10);
```

# do-while statement



# for statement

- Loop with **pre-testing** the condition
- Three optional parts: initialization; loop condition; increment
- Initialization is executed only once, before the first test of condition
- Increment is executed after the loop body and the **continue** statement, (but not after **break**)
- Otherwise, same as the **while** statement

```
int i = 0;
for (i = 0; i < 10; ++i)
{
 if (i % 2)
 {
 printf("%d is odd\n", i);
 }
}
```



# for statement

- The initialization part can be a declaration since C99
- The declared variable has block scope to the for loop (not visible after)

```
// int i = 0;
for (int i = 0; i < 10; ++i)
{
 if (i % 2)
 {
 printf("%d is odd\n", i);
 }
}
```

```
// i is not exists here
```

# for statement

- The initialization part can be a declaration since C99
- The declared variable has block scope to the for loop (not visible after)

```
// int i = 0;
for (int i = 0; i < 10; ++i)
{
 if (i % 2)
 {
 printf("%d is odd\n", i);
 }
}

{
 int i = 0;
 while (i < 10)
 {
 if (i % 2)
 {
 printf("%d is odd\n", i);
 }
 ++i
 }
}
```

# for statement

```
for (i = 0; i < 10; ++i)
{
 f(&i);
}
```

# for statement

- The initialization part can be omitted

```
// i is already 0
for (; i < 10; ++i)
{
 f(&i);
}
```

# for statement

- The initialization part can be omitted
- The increment part can be omitted

```
for (; i < 10;)
{
 f(&i); // f() may increment i
}
```

# for statement

- The initialization part can be omitted
- The increment part can be omitted
- The condition part can be omitted, meaning **always true**

```
for (; ;)
{
 f(&i); // f() may call exit()
}
```

# break in loop

```
void print_first_odd(int t[], int sz)
{
 for (int i = 0; i < sz; ++i)
 {
 if (t[i] % 2)
 {
 printf("the first odd is %d at index %d\n", t[i], i);
 break;
 }
 }
}
```

# continue in loop

```
void print_second_odd(int t[], int sz)
{
 int first = 1;
 for (int i = 0; i < sz; ++i)
 {
 if (t[i] % 2)
 {
 if (first)
 {
 first = 0;
 continue;
 }
 printf("the second odd is %d at index %d\n", t[i], i);
 break;
 }
 }
}
```



# continue in loop

```
void print_nth_odd(int t[], int sz, int n)
{
 int cnt = 1; // first is n == 1
 for (int i = 0; i < sz; ++i)
 {
 if (t[i] % 2)
 {
 if (cnt < n)
 {
 ++cnt;
 continue;
 }
 printf("the %dth odd is %d at index %d\n", n, t[i], i);
 break;
 }
 }
}
```

# return statement

- Unconditionally returns from the function
- Returning from **main( )** finishes the program
- Non-void functions should return an expressions  
(But not all compilers check this)
- The value to return is converted to the return type of the function

```
int return_first_odd(int t[], int sz) // wrong!
{
 for (int i = 0; i < sz; ++i)
 {
 if (t[i] % 2)
 {
 return t[i];
 }
 }
}
```

# return statement

- Unconditionally returns from the function
- Returning from **main( )** finishes the program
- Non-void functions should return an expressions  
(But not all compilers check this)
- The value to return is converted to the return type of the function

```
int return_first_odd(int t[], int sz)
{
 for (int i = 0; i < sz; ++i)
 {
 if (t[i] % 2)
 {
 return t[i];
 }
 }
 return 0; // 0 is not odd, caller can recognize
}
```

# return statement

- Unconditionally returns from the function
- Returning from **main()** finishes the program
- Non-void functions should return an expressions  
(But not all compilers check this)
- The value to return is converted to the return type of the function

```
int* return_first_odd(int t[], int sz) // with pointer
{
 for (int i = 0; i < sz; ++i)
 {
 if (t[i] % 2)
 {
 return &t[i];
 }
 }
 return NULL; // NULL pointer, caller can recognize
}
```

# goto statement

- Unconditionally jumps to a **label**
  - The **goto** statement and the **label** should be in the same function
  - Variables jumped over have undefined value
  - Must not jump into the scope of variadic length array (VLA)
- 
- DO NOT USE GOTO!

# Sample: change vowels

- Read the standard input and copy all characters to standard output
- Converting each lowercase vowels to the next one (cyclically)
- All other characters remain the same

```
$./a.out
```

```
Hello world, this is a sample program. This program
changes the vowels aeiou to the next one.
```

```
Hillu wurld, thos os e sempli prugrem. Thos prugrem
chengis thi vuwils eioua tu thi nixt uni.
```

```
$
```

# Sample: change vowels

- Read the standard input and copy all characters to standard output
- Converting each vowels to the next one (cyclically)

```
#include <stdio.h>

char change(char ch);

int main()
{
 int ch;

 while (EOF != (ch = getchar()))
 {
 putchar(change(ch));
 }
 return 0;
}
```

# Sample: change vowels

```
char change(char ch) // wrong!
{
 if ('a' == ch)
 ch = 'e';
 if ('e' == ch)
 ch = 'i';
 if ('i' == ch)
 ch = 'o';
 if ('o' == ch)
 ch = 'u';
 if ('u' == ch)
 ch = 'a';
 return ch;
}
```

- Prints the wrong output



# Sample: change vowels

```
char change(char ch)
{
 if ('a' == ch)
 ch = 'e';
 else if ('e' == ch)
 ch = 'i';
 else if ('i' == ch)
 ch = 'o';
 else if ('o' == ch)
 ch = 'u';
 else if ('u' == ch)
 ch = 'a';
 return ch;
}
```

- Prints the correct output, but not maintainable, readable.

# Sample: change vowels

```
char change(char ch) // wrong!
{
 switch (ch)
 {

 case 'a': ch = 'e';
 case 'e': ch = 'i';
 case 'i': ch = 'o';
 case 'o': ch = 'u';
 case 'u': ch = 'a';
 }
 return ch;
}
```

- Prints the wrong output.

# Sample: change vowels

```
char change(char ch)
{
 switch (ch)
 {
 default : break;
 case 'a' : ch = 'e'; break;
 case 'e' : ch = 'i'; break;
 case 'i' : ch = 'o'; break;
 case 'o' : ch = 'u'; break;
 case 'u' : ch = 'a'; break;
 }
 return ch;
}
```

- Prints the correct output, fast, but not too much maintainable.

# Sample: change vowels

```
char change(char ch)
{
 char from[5] = {'a', 'e', 'i', 'o', 'u'};
 char to[5] = {'e', 'i', 'o', 'u', 'a'};

 int i = 0;

 while (i < 5)
 {
 if (from[i] == ch)
 return to[i];
 ++i;
 }
 return ch;
}
```

- Prints the correct output, more flexible than the earlier
- The from and to array can be read from input

# Sample: change vowels

```
char change(char ch)
{
 static const char from[5] = {'a', 'e', 'i', 'o', 'u'};
 static const char to[5] = {'e', 'i', 'o', 'u', 'a'};

 size_t i = 0; // unsigned int

 for (i = 0; i < sizeof(from)/sizeof(from[0]); ++i)
 {
 if (from[i] == ch)
 return to[i];
 }
 return ch;
}
```

- Static variables initialized only once
- Const variables are immutable
- `sizeof(t)/sizeof(t[0])` is automatically adjusted to the #elements of the array

# Sample: change vowels

```
char change(unsigned char ch)
{
 static const char table[256] = { /* ... */ };
 return table[ch];
}
```

- Initialize the table with characters corresponding to incoming **char** as index
- Fastest solution
- Still flexible: table can be read from input
- Parameter is **unsigned char** to ensure non-negative indexes