GlobalLogic®

Basecamp: C/Embedded

2

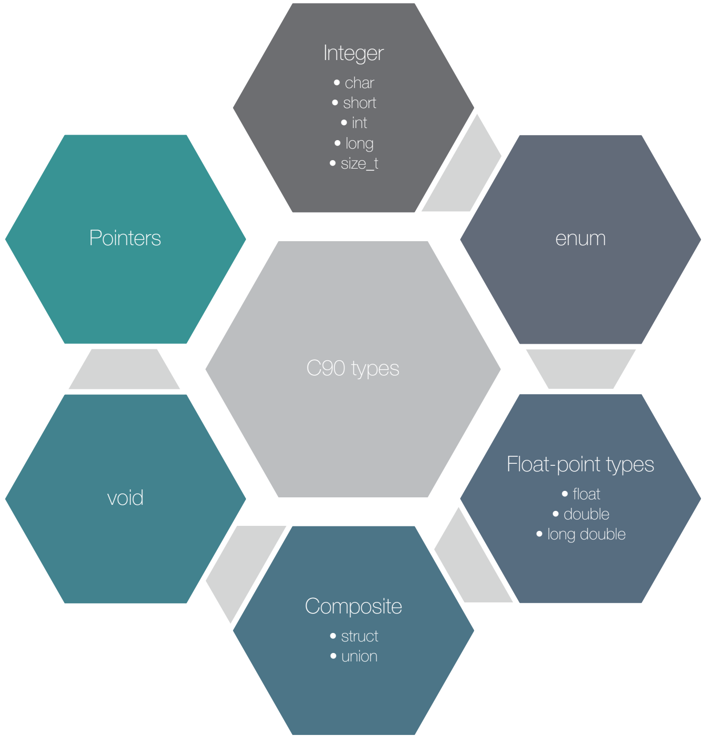
Agenda

1. Homework review
2. Types
3. if, switch-case
4. for, while
5. break, continue, goto
6. Typical patterns

Types in C

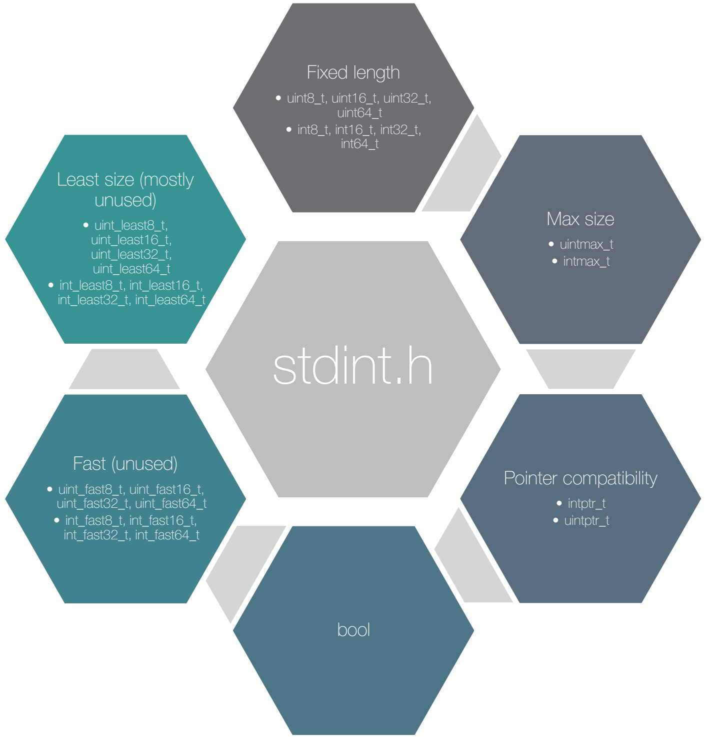
C90 types

- C90 basic types get more support in C99
- Integer types have extension with sign specification: signed and unsigned
  - signed long = long with sign bit
  - unsigned = unsigned int = int without sign bit
- Additional types
  - size_t
  - offset_t
  - ptrdiff_t
- Float is usually software emulated type, while double requires hardware support
- char is signed by default



C99 stdint.h

- C99 adds a bunch of types especially interesting for embedded and protocol specification
- Fixed length types are widely used
- Max size and pointer compatibility types sometimes are used in a cross-compile environment
- Native **bool** support was added
- Fast and least types are mostly unused


Fixed length
• uint8_t, uint16_t, uint32_t, uint64_t
• int8_t, int16_t, int32_t, int64_t
Least size (mostly unused)
• uint_least8_t, uint_least16_t, uint_least32_t, uint_least64_t
• int_least8_t, int_least16_t, int_least32_t, int_least64_t
Max size
• uintmax_t
• intmax_t
stdint.h
Fast (unused)
• uint_fast8_t, uint_fast16_t, uint_fast32_t, uint_fast64_t
• int_fast8_t, int_fast16_t, int_fast32_t, int_fast64_t
Pointer compatibility
• intptr_t
• uintptr_t
bool

sizeof

Importance of sizes

- sizeof measures type size in the number of chars
- sizeof operator can be applied both to the type and the variable. Both applications are compile-time evaluated
- When designing embedded hardware and communication protocols with it developer must know exact size of the type he is using
- sizes of fixed size types is encoded in the name, i.e. sizeof(uin32_t) strictly equals to 4 bytes

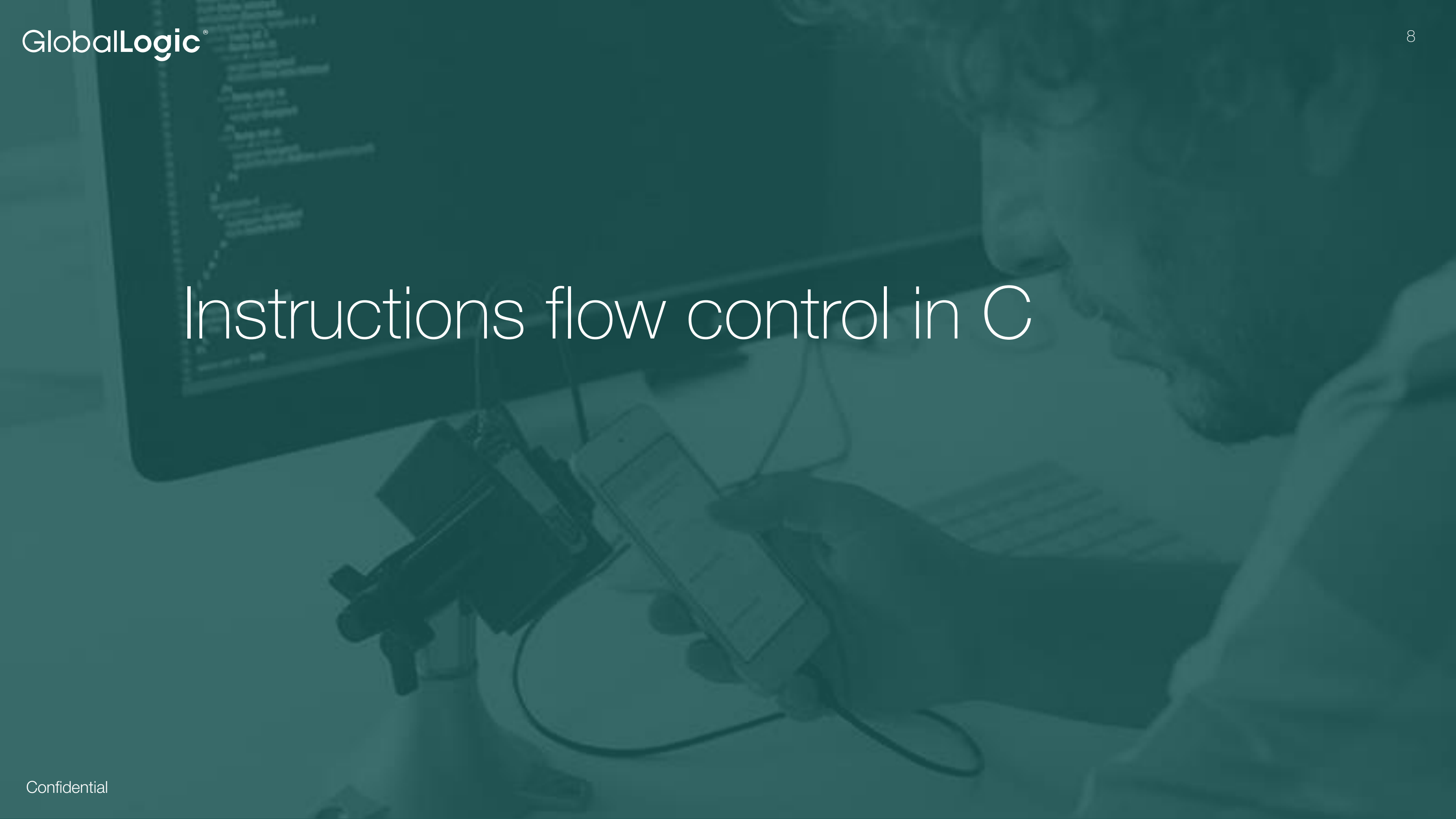
Sizes of types

- char < short <= int <= long < long long
  - x86/arm (compiler defined)
    - sizeof(char) = 1
    - sizeof(short) = 2
    - sizeof(int) = sizeof(long) = 4
    - sizeof(long long) = 8
  - x64/arm64 (compiler defined)
    - sizeof(char) = 1
    - sizeof(short) = 2
    - sizeof(int) = 4
    - sizeof(long) = sizeof(long long) = 8

Implicit integer casts

- C has some implicit casts:
  - Integer is cast to a bigger same signedness type, if operation against larger type is performed
  - bool is implicitly casted to integer values 0 or 1
  - Vice versa, integer used in logical expression is casted backward (0 is for false, everything else is true)
- Additionally void* has implicit cast, but we'll talk about it later

```
int i;
char c = 3;
char c2 = 126;
bool b = true;

i = c + c2; /* causes overflow */
i = (int)c + c2; /* c2 implicitly casted to int */
i
```



Instructions flow control in C

GlobalLogic

if

- Most common construction ever used for imperative logic control
- No differences in C++
- **Py:**
  - Parentheses are mandatory
  - No elif form, instead another branching operator is used: switch-case
  - As usual, white space doesn't play any role in the instruction flow organization
- **J:** pay attention
  - If requires logically evaluated expression in the clause, however all of the integer types are implicitly cast to bool, so condition might contain any arithmetic, and even pointers

```
if (logical expression)
    statement;

if (logical expression) {
    statement 1;
    statement 2;
} else {
    statement 1;
    statement 2;
}
```

else

- else in nested if belongs to the last if (6.8.4.1/3)
- **Py**: common logic mislead to the right, don't pay attention to the leading whitespace, only scopes define syntax
- Good style:
  - all of the conventions recommend to put else on the same depth, as the corresponding if
  - most of the conventions require explicit scope for the nested if statement, if it contains else
  - most of the conventions recommends to align a scope closing parenthesis on the same depth as the corresponding if
  - most of the conventions recommend to skip semicolon after closing parenthesis

```
if (expression)
    if (expression2)
        statement1;
else //*<- belongs to the last if
    statement2;

/* Good style: */
if (expression) {
    if (expression2)
        statement1;
    else
        statement2;
}
```

if coding style

/* Linux kernel coding style */

```
if (logical expression) {
        statement 1;
        statement 2;
} else
        statement 3;

if (logical expression)
        statement 4;
```

/* MISRA */

```
if (logical expression)
{
        statement 1;
        statement 2;
}
else
{
        statement 3;
}

if (logical expression)
{
        statement 4;
}
```

if nesting

- Nesting **if** more than 3 levels deep is forbidden in most of the conventions
- If you have a deeply nested **if**, probably you need to optimize some part of your function, or made a logical errors somewhere
- Solutions
  - Make your functions shorter, do more functions. A recommended size for a function is shorter than one screen
  - In case you have multiple **return**s allowed try to linearize sanity checks
  - In case you have a **goto** allowed as well as multiple **return** within **if**, try to apply exception patterns
  - Evaluate logical expressions, used in if beforehand and recombine **if**s

```
/* Bad example */
if (packet_is_allowed) {
    total_size += packet.len;
    if (total_size < SOME_CONSTANT) {
        if (packet.type == CONTROL_PACKET) {
            counter++;
            if (counter > 3) {
                printf("Too many control packets");
                errors++;
                if (errors > 10) {
                    return -1;
                }
            }
        } else
            forward_packet();
    }
}
```

switch-case

- **Py**: a replacement for if-elif-else construction
- Generally, switch-case is a very powerful construction
- To exit switch-case you need to use break
- *Fall-through* is allowed, however it's not recommended.
- In case you need a *fall-through*, please, leave an explicit comment
- Repeating cases (see example, CONSTANT 3 and CONSTANT4) is a civilized and common used way of *fall-through* organization
- It's highly recommended to have a default clause in every switch

```
switch (integer_value) {
    case CONSTANT1:
        statement;
        break;

    case CONSTANT2:
        statement;
        /* fall through */

    case CONSTANT3:
    case CONSTANT4:
        statement;
        break;

    default:
        statement;
        break;
}
```

switch-case: other

- **J:** pay attention, that due to implicit integer cast, you can use a pointer in a switch clause, or a character
- There is a GCC extension for ranges support, see example
- switch-case is especially powerful while combined with enums

```
switch (character) {
    case ‘A’ … ‘Z’:
        printf(“letters”);
        break;

    case ‘0’ … ‘9’:
        printf(“digits”);
        break;

    default:
         printf(“This should not happen”);
        break;
}
```

for

- A most generic loop, for combines initialization, checking for the logical clause and after every loop change
- **C++:** type definition in the middle of the scope is a C++ syntax sugar, so common structures like
  `for (auto i = 0; i < N; i++)`
  are naturally forbidden in C
- **C++:** there is no foreach extension in C
- **J:** common true-false check in for has implicit integer cast in the same way, as if had

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

for (; i > 3; i--)
     statement;

/* Common mistake */
bool b;
for (b = true, i=0; b = true; i++)
     b = some_function(i);
```

while

- While is another loop type in C
- While has two forms, do-while is least used as there is little use cases when loop should be performed at least once
- **J:** common true-false check in for has implicit integer cast in the same way, as if had

```
while (condition) {
    statement1;
    statement2;
    statement3;
}

do {
    statement1;
    statement2;
    statement3;
} while (condition);
```

break and continue

- break and continue are the operators for loop cycle skipping and exit
- **J, Py:** break in C can break only the last loop
- continue might be forbidden by your code convention as a form of the goto and inconsistent loop checking logic, however it's widely used to skip some loops processing

```
for (i = 0; i < 10; i++) {
    statement;

    if (condition)
        continue;

    statement;

    if (condition)
        break;

    statement;
}
```

goto

- goto jumps from the certain instruction to another place in the code
- goto is a direct mapping of a near jump in the assembly, therefore it’s supported in C
- goto doesn’t leave traces in your stack traces, therefore the code with goto is usually harder to debug
- goto combined with loops, especially jumps to the loops scopes and within them is very unbovious
- goto is usually forbidden in most of the code conventions, however it might participate in some very useful behavioral patters

```
    if (condition) {
        statement1;
        goto label;
    }

    statement2;

    if (condition) {
        statement3;
        goto label;
    }

    statement4;

label:
    statement5;
```

Typical behavioral patterns

Nested ifs vs control variable

- There is a common approach to convert nested if with error checking to a common variable approach
- It’s especially useful, when working in both deep nesting and goto forbidding conventions, like MISRA

```
if ( is_ampdu(skb) )
{
    if ( reserve_buffer(skb->len) && \
            ampdu_session_in_progress(dev) )
    {
        if ( ! ampdu_session_in_progress(dev) )
        {
            return -EBUSY;
        }

        if ( enqueue_packet_for_transmit(dev, skb) )
        {
            return trigger_ampdu_transmission(dev);
        }
    }
}

return -EAGAIN;
```

Nested ifs vs control variable (#2)

- Such linearization looks a little bit bulky, however if the temporary variable is not used in other places the resulting machine code will be optimal

- This kind of code has greater readability, as developer spends lesser efforts on reading of a linear construction comparable to complex if-else clauses

```
int rc = 0;

if ( is_ampdu(skb) )
{
      rc = reserve_buffer(skb->len);
}

if ( rc == 0 )
{
      if ( ! ampdu_session_in_progress(dev) )
      {
            return -EBUSY;
      }

      rc = enqueue_packet_for_transmit(dev, skb);
}

if ( rc == 0 )
{
      rc = trigger_ampdu_transmission(dev);
}

return rc;
```

Multiple return vs goto

- returns are very similar to goto
- Sometimes, you need to perform some cleanup before return from the function
- If you have multiple returns with multiple cleanup and at the same time goto is allowed in your code convention, you can consider removing cleanup code duplication.

Multiple return vs goto

```
static int e100_exec_cmd(struct nic *nic, u8 cmd, dma_addr_t dma_addr)
{
        unsigned long flags;
        unsigned int i;
        int err = 0;
        spin_lock_irqsave(&nic->cmd_lock, flags);
        /* Previous command is accepted when SCB clears */
        for (i = 0; i < E100_WAIT_SCB_TIMEOUT; i++) {
                // some actions
        }
        if (unlikely(i == E100_WAIT_SCB_TIMEOUT)) {
                spin_unlock_irqrestore(&nic->cmd_lock, flags);
                return -EAGAIN;
        }
        // some actions

        spin_unlock_irqrestore(&nic->cmd_lock, flags);
        return 0;
}
```

```
static int e100_exec_cmd(struct nic *nic, u8 cmd, dma_addr_t dma_addr)
{
        unsigned long flags;
        unsigned int i;
        int err = 0;
        spin_lock_irqsave(&nic->cmd_lock, flags);
        /* Previous command is accepted when SCB clears */
        for (i = 0; i < E100_WAIT_SCB_TIMEOUT; i++) {
                // some actions
        }
        if (unlikely(i == E100_WAIT_SCB_TIMEOUT)) {
                err = -EAGAIN;
                goto err_unlock;
        }
        // some actions

err_unlock:
        spin_unlock_irqrestore(&nic->cmd_lock, flags);
        return err;
}
```

goto for exceptions

- Exceptions are extremely useful concepts from OOD
- **C**: (sic!) The basic idea is the following:
    - In general work flow, imperative code is executed once per statement
    - If if fails, it stops execution of other commands in the function and directly jumps outside of the scope
    - Then a check for exception handler presence occurs, if there is no exception handler detected, a next jump issued
    - When exception catched by a higher level code, it starts processing of the information saved in the exception
- As C doesn't have support for OOP, but generally idea of failsafe function exit is interesting, there are standard approaches for exceptions implementations on top of goto
- Of course, it's possible only if you have goto allowed by your convention

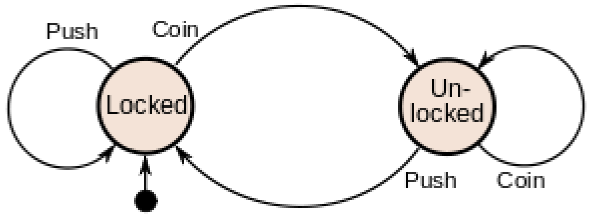
goto for exceptions

```
static int e100_up(struct nic *nic)
{
        int err;
        if ((err = e100_rx_alloc_list(nic)))
                return err;
        if ((err = e100_alloc_cbs(nic))) {
                e100_rx_clean_list(nic);
                return err;
        }
        if ((err = e100_hw_init(nic))) {
                e100_clean_cbs(nic);
                e100_rx_clean_list(nic);
                return err;
        }
        if ((err = request_irq(/* some arguments */))) {
                del_timer_sync(&nic->watchdog);
                e100_clean_cbs(nic);
                e100_rx_clean_list(nic);
                return err;
        }
        // some code
        return 0;
}
```

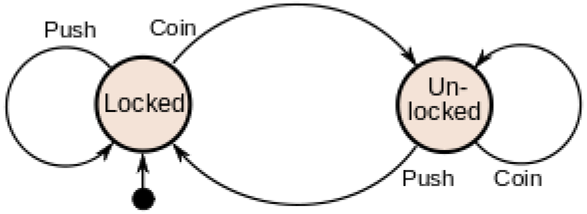
```
static int e100_up(struct nic *nic)
{
        int err;
        if ((err = e100_rx_alloc_list(nic)))
                return err;
        if ((err = e100_alloc_cbs(nic)))
                goto err_rx_clean_list;
        if ((err = e100_hw_init(nic)))
                goto err_clean_cbs;
        if ((err = request_irq(/* some arguments */)))
                goto err_no_irq;
        // some code
        return 0;

err_no_irq:
        del_timer_sync(&nic->watchdog);
err_clean_cbs:
        e100_clean_cbs(nic);
err_rx_clean_list:
        e100_rx_clean_list(nic);
        return err;
}
```

State machines idea

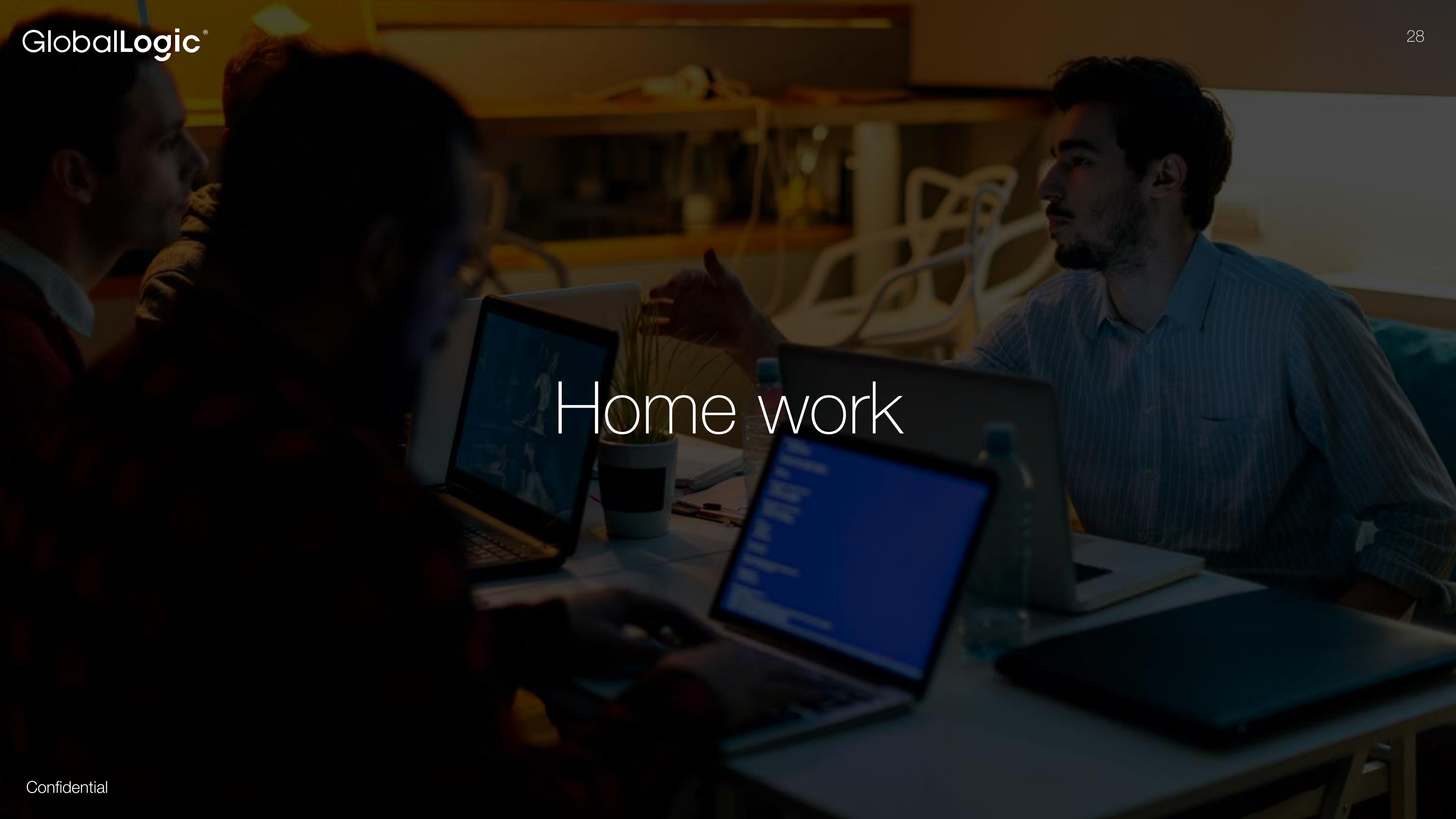
- Finite-state machines is the concept from the models of computation
- Covering of state machine theory is out of the scope, however the general idea is very simple:
    - System has some state
    - When some action occurs, the state can be changed to another one
    - When state is changed, something happens and system will react on next actions in a different way
- switch-case clauses combined with some constant values is very common for state machine organization
- Usually there is a variable containing state machine state and a set of constants describing other states



switch-case for state machines



```
int state = 0; /* 0 is for locked, 1 is for unlocked */
void f(int action) /** action: 0 is for push, 1 is for coin*/
{
        switch (state) {
                case 0:
                        if (action == 1) {
                                unlock();
                                state = 1;
                        }
                        break;
                case 1:
                        if (action == 0) {
                                lock();
                                state = 0;
                        }
                        break;
                default:
                        printf(“Unknown state”);
                        break;
}
```



Home work

Quiz (light)

Fill in the blanks in each of the following statements.

a) The __________ statement, when executed in a repetition statement, causes the next iteration of the loop to be performed immediately.

b) The __________ statement, when executed in a repetition statement or a switch, causes an immediate exit from the statement.

c) The __________ is used to test a particular variable or expression for each of the constant integral values it may assume.

State whether the following are true or false. If the answer is false, explain why.

a) The default case is required in the switch selection statement.

b) The break statement is required in the default case of a switch selection statement.

c) The expression (x > y && a < b) is true if either x > y is true or a < b is true.

d) An expression containing the || operator is true if either or both of its operands is true.

Write a statement or a set of statements to accomplish each of the following tasks:

a) Sum the odd integers between 1 and 99 using a for statement. Assume the integer variables sum and count have been defined.

b) Print the value 333.546372 in a field width of 15 characters with precisions of 1, 2, 3, 4 and 5. Left justify the output. What are the five values that print?

c) Calculate the value of 2.5 raised to the power of 3 using the pow function. Print the result with a precision of 2 in a field width of 10 positions. What is the value that prints?

d) Print the integers from 1 to 20 using a while loop and the counter variable x. Assume that the variable x has been defined, but not initialized. Print only five integers per line.

e) Repeat Exercise 4.3 (d) using a for statement.

Home task (light)

- Write a program that reads in two integers and determines and prints if the first is a multiple of the second.
- Write a program that inputs three different integers from the keyboard, then prints the sum, the average, the product, the smallest and the largest of these numbers.
- The formulas for calculating BMI are

```
                    weightInPounds × 703
BMI = -----------------------------------------
              heightInInches × heightInInches
```

or

```
                    weightInKilograms
BMI = -----------------------------------------
            heightInMeters × heightInMeters
```

Create a BMI calculator application that reads the user's weight in pounds and height in inches (or, if you prefer, the user's weight in kilograms and height in meters), then calculates and displays the user's body mass index. Also, the application should display the following information from the Department of Health and Human Services/National Institutes of Health and evaluate user's BMI:

BMI VALUES
Underweight: less than 18.5
Normal: between 18.5 and 24.9
Overweight: between 25 and 29.9
Obese: 30 or greater

Yours: 55.5 - Obese

- (Palindrome Tester) A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a program that reads in a five-digit integer and determines whether or not it's a palindrome.
- (Find the Smallest) Write a program that finds the smallest of several integers. Assume that the first value read specifies the number of values remaining.
- (Factorial) Write a program that estimates the value of the mathematical constant e by using the formula:

```
         1     1     1           1
e = 1 + --- + --- + --- + … + --- + …
         1!    2!    3!          n!
```

- (Calculating the Value of π) Calculate the value of π from the infinite series

```
         4     4     4     4     4
π = 4 - --- + --- - --- + --- - --- + ...
         3     5     7     9     11
```

Harder tasks

Quiz

Nope, still no

Home task

Nope, still no

Clean code is important

C doesn’t restricts you on the way you put spaces and the way you names the identifiers.
If the code is supposed to be read at least twice, you have to write it clean, especially if the code will be shared