Maciej Sobieraj

Lecture 4

Outline

1. Flow control (continued), more data types and computer logic
   1. **Loops**
   2. Computer logic
2. Aggregating data into arrays
   1. switch - the different face of ‘if’
   2. Vectors: why do you need them?
3. Quiz

The "while" loop

- We want to ask you a strange question: how long do you usually take to wash your hands? Don't think about it, just answer. Well, when your hands are very dirty, you wash them for a very long time.

```
while my hands are dirty

    I am washing my hands;
```

The “while” loop

- We use the word “*while*” instead of “*if*”. The semantic difference, however, is **more important**: when the condition is met, *if* performs its statement **once**; *while* **repeats the execution** as long as the condition evaluates to “*true*”

```
while(conditional_expression)

    statement;
```

The “while” loop

- Let’s make a few observations:
  - if you want *while* to execute **more than one** statement, you must use a **block**
  - an instruction or instructions executed inside the loop are called the **loop's body**;
  - if the condition is “*false*” (equal to zero) as early as when it’s tested for the first time, the **body is not executed** even once;
  - the body should be able to change the condition value, because if the condition is true at the beginning, the body might **run continuously to infinity**.

The “while” loop

```
while(conditional_expression) {
        statement_1;
        statement_2;
        :
        :
        statement_n;
}
```

The “while” loop

- Here’s an example of a loop that can’t finish its execution.

```
while(1) {
    printf("I am stuck inside a loop");
}
```

The “while” loop

- **We can declare the variable and assign the value at the same time**.

```
int variable = 0;
```

- The part of the declaration placed on the right side of the = sign is called an **initiator**.

```
float PI = 3.1415;
double PI2 = 2.0 * PI;
```

The “while” loop

```
#include <stdio.h>

int main(void) {
	/* temporary storage for the incoming numbers */
	int number;

	/* we will store the currently greatest number here */
	int max = -100000;

	/* get the first value */
	scanf("%d",&number);

	/* if the number is not equal to -1 we will continue */
	while(number != -1) {

		/* is the number greater than max? */
		if(number > max)

			/* yes – update max */
			max = number;

		/* get next number */
		scanf("%d",&number);

	}

	/* print the largest number */
	printf("The largest number is %d \n",max);

	/* finish the program successfully */
	return 0;
}
```

The “while” loop in some examples

```
/*   program reads a sequence of numbers and counts how many numbers
   are even and how many odd;
   program terminates when a zero is entered     */

#include <stdio.h>

int main(void) {
   /* we will count the numbers here */
   int Evens = 0, Odds = 0;

   /* we will store the incoming numbers here */
   int Number;

   /* read first number */
   scanf("%d",&Number);

   /* 0 terminates execution */
   while(Number != 0) {
      /* check if the number is odd */
      if(Number % 2)
         /* increase „odd" counter */
         Odds++;
      else
         /* increase „even" counter */
         Evens++;
      /* read next number */
      scanf("%d",&Number);
   }
   /* print results */
   printf("Even numbers: %d\n",Evens);
   printf("Odd numbers: %d\n",Odds);
   return 0;
}
```

The “while” loop in some examples

- These two forms are equivalent

```
while(Number != 0) { ... }

while(Number) { ... }
```

- Also these

```
if(Number % 2 == 1) ...

if(Number % 2) ...
```

The “while” loop in some examples

- There are two things that can be written more compactly.
  - First, the condition of the *while* loop.

```c
#include <stdio.h>

int main(void) {
    int counter = 5;

    while(counter != 0) {
        puts("I am an awesome program");
        counter--;
    }
    return 0;
}
```

The “while” loop in some examples

- Another change requires some knowledge of how the post-decrement works.

```
#include <stdio.h>

int main(void) {
    int counter = 5;

    while(counter) {
        puts("I am an awesome program");
        counter--;
    }
    return 0;
}
```

The “while” loop in some examples

- This is the simplest form of the program

```c
#include <stdio.h>

int main(void) {
    int counter = 5;

    while(counter--)
        puts("I am an awesome program");
    return 0;
}
```

The “do” loop, or do it at least once

- The *while* loop has two important features:
  - it checks the condition **before** entering the body;
  - the body will not be entered if the condition is false.
- There’s another loop in the “C” language which acts as a mirror image of the *while* loop. We say so because in that loop:
  - the condition is **checked at the end** of body execution;
  - the loop's body is executed **at least once**, even if the condition is not met.

The “do” loop, or do it at least once

```
do
   statement;
while(condition);

do {
   statement_1;
   statement_2;
   :
   :
   statement_n;
} while(condition);
```

The “do” loop, or do it at least once

- Let’s go back to the program to search for the largest number.

```
#include <stdio.h>

int main(void) {
   int number;
   int max = -100000;
   int counter = 0;

   do {
      scanf("%d",&number);
      if(number != -1)
         counter++;
         if(number > max)
            max = number;
   } while (number != -1);
   if(counter)
       printf("The largest number is %d \n",max);
   else
      printf("Are you kidding? You haven't entered any number!");
   return 0;
}
```

“for” - the last loop

- The last kind of loop available in the “C” language comes from the observation that sometimes it’s more important to **count the “turns”** of a loop than to **check the conditions**.

“for” - the last loop

- We can distinguish three independent elements here:
  - initiation of the counter→red colour;
  - checking the condition→green colour;
  - modifying the counter→blue colour.

```
int i;

i = 0;

while (i < 100) {
   /* the body goes here
*/

    i++;

}
```

“for” - the last loop

- We can provide something like a generalized scheme for these kinds of loops

```
initialization;
while (checking) {
   /* the body goes
here */

   modifying;

}
```

“for” - the last loop

- All three decisive parts are gathered together. The loop is clear and easy to read.

```
for( i = 0;  i < 100;  i++ ) {
    /* the body goes here */
}
```

“for” - the last loop

- The variable used for counting the loop’s turns is often called a **control variable**.

```
for( i = 0;  i < 100;  i++ ) {
    /* the body goes here */
}
```

“for” - the last loop

- The *for* loop has an interesting **singularity**. If we **omit any** of its three components, it’s **presumed that there is a 1 there instead**.
  - One of the consequences of this is that a loop written in this way is an infinite loop

```
for( ; ; ) {
    /* the body goes here */
}
```

“for” - the last loop

- Let’s look at a short program whose task is to write some of the first powers of 2.

```
#include <stdio.h>

int main(void) {
   int exp;
   int pow = 1;

   for(exp = 0; exp < 16; exp++) {
   printf("2 to the power of %d is %d\n",exp,pow);
      pow *= 2;
   }
   return 0;
}
```

break and continue – the loop's spices

- The developer could be faced with the following choices:
  - it appears that it’s **unnecessary to continue** the loop as a whole; we should refrain from executing the loop’s body and go further;
  - it appears that we need to **start the condition testing** without completing the execution of the current turn.

break and continue – the loop's spices

- These two instructions are:
  - *break* - **exits the loop immediately** and unconditionally ends the loop’s operation; the program begins to execute the nearest instruction after the loop's body;
  - *continue* – behaves as if the program has suddenly reached **the end of the body; the end of the loop’s body is reached**, the control variable is modified (in the case of *for* loops), and the condition expression is tested.
- Both these words are keywords.

break and continue – the loop's spices

```
#include <stdio.h>

int main(void) {
   int number;
   int max = -100000;
   int counter = 0;

   for( ; ; ){
      scanf("%d",&number);
      if(number == -1)
         break;
      counter++;
      if(number > max)
         max = number;
   }
   if(counter)
       printf("The largest number is %d \n",max);
   else
      printf("Are you kidding? You haven't entered any number!");
   return 0;
}
```

break and continue – the loop's spices

```
#include <stdio.h>

int main(void) {
  int number;
  int max = -100000;
  int counter = 0;

  do {
    scanf("%d",&number);
    if(number == -1)
      continue;
    counter++;
    if(number > max)
      max = number;
  } while (number != -1);
  if(counter)
     printf("The largest number is %d\n",max);
  else
    printf("Are you kidding? You haven't entered any number!");
  return 0;
}
```

Outline

1. Flow control (continued), more data types and computer logic
    1. Loops
    2. **Computer logic**
2. Aggregating data into arrays
    1. switch - the different face of ‘if’
    2. Vectors: why do you need them?
3. Quiz

Computers and their logic

- Have you noticed that the conditions we’ve used so far have been very simple, not to say – quite primitive? The conditions we use in real life are much more complex. Let's look at the sentence:

  If we have some free time, and the weather is good, we will go for a walk.

- We’ve used the conjunction “*and*”, which means that going for a walk depends on the simultaneous fulfillment of the two conditions. In the language of logic, the connection of conditions is called a **conjunction**.

Computers and their logic

- And now another example:

  If you are in the mall or I am in the mall, one of us will buy a gift for Mom.

  The appearance of the word "*or*" means that the purchase depends on at least one of these conditions. In logic terms, this is called a **disjunction**.

Pride && Prejudice

- The logical conjunction operator in the "C" language is a **digraph** *&&* (*ampersand ampersand*).

```
Counter > 0 && Value == 100
```

Pride && Prejudice

- The result provided by the *&&* operator can be determined on the basis of the **truth table**.

left	right	left&&right
false	false	false
false	true	false
true	false	false
true	true	true

To be || not to be

- The disjunction operator is the digraph | | (*bar bar*). It’s a binary operator with a lower priority than *&&*

left	right	left||right
false	false	false
false	true	true
true	false	true
true	true	true

To be || not to be

- In addition, there's another operator that can be used to construct conditions. It's a unary operator performing a **logical negation**.

arg	! arg
false	true
true	false

Some logical expressions

```
Variable > 0        !(Variable <= 0)
Variable != 0       !(Variable == 0)
```

Some logical expressions

- You may remember **De Morgan's laws** from school. They say that:

  The negation of a conjunction is the disjunction of the negations.
  The negation of a disjunction is the conjunction of the negations.

```
!(p && q) == !p || !q

!(p || q) == !p && !q
```

How to deal with single bits

- the following snippet will assign a value of 1 to the *j* variable if *i* is not zero; otherwise, it will be 0 (why?).

```
int i,j;

j = !!i;
```

How to deal with single bits

- **Bitwise operators**
  - & (*ampersand*) bitwise conjunction
  - | (*bar*) bitwise disjunction
  - ~ (*tilde*) bitwise negation
  - ^ (*caret*) bitwise exclusive or
- Let's make it easier:
  - & requires exactly two “1s” to provide “1” as the result
  - | requires at least one “1” to provide “1” as the result
  - ^ requires only one “1” to provide “1” as the result
  - ~ (is one argument and) requires “0” to provide “1” as the result

How to deal with single bits

left	right	left&right	left|right	left^right
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

arg	~arg
0	1
1	0

How to deal with single bits

- Let’s have a look at an example of the difference in operation between logical and bit operations. Let’s assume that the following declaration has been performed:
  - **int** i = 15, j = 22;
- If we assume that the *ints* are stored with 32 bits, the bitwise image of the two variables will be as follows

i: 00000000000000000000000000001111
j: 00000000000000000000000000010110

How to deal with single bits

- The declaration is given:
  - **int** log = i *&&* j;
- We're dealing with a logical conjunction. Let's trace the course of the calculations. Both variables *i* and *j* are not zeros so will be deemed to represent "*true*".

log: 00000000000000000000000000000001

How to deal with single bits

- Now the bitwise operation – here it is:
  - **int** bit = i & j;
- The & operator will operate with each pair of corresponding bits separately, producing the values of the relevant bits of the result.

bit: 00000000000000000000000000000110

How to deal with single bits

- Let's try the negation operators now. First the logical one:
  - **int** logneg = !i;
- The *logneg* variable will be set to 0 so its image will consist of zeros only.
  The bitwise negation goes here:
  - **int** bitneg = ~i;

logneg: 00000000000000000000000000000000

bitneg: 11111111111111111111111111110000

How to deal with single bits

- Each of the previous two-argument operators can be used in their **abbreviated** forms.

x = x & y;	x &= y;
x = x | y;	x |= y;
x = x ^ y;	x ^= y

How to deal with single bits

- The variable stores information about the various aspects of system operation. Each bit of the variable stores one *yes/no* value.

```
int FlagRegister;
```

```
FlagRegister: 0000000000000000000000000000x000
```

- Only one of these bits is yours – bit number three

How to deal with single bits

- You may face the following tasks:
  - **check the state** of your bit – you want to find out the value of your bit; comparing the whole variable to zero will not do anything, because the remaining bits can have completely unpredictable values, but we can use the following conjunction property:
    - x&1=x
    - x&0=0

How to deal with single bits

- If we apply the *&* operation to the *FlagRegister* variable along with the following bit image:
    - 00000000000000000000000000001000
- (note the "1" at your bit's position) we obtain one of the following bit strings as a result:
    - 0000000000000000000000000000**1**000 if your bit was set to “1”
    - 0000000000000000000000000000**0**000 if your bit was reset to “0”

How to deal with single bits

- A sequence of zeros and ones whose task is to grab the value or to change the selected bits is called **a bitmask.** Let's try to build a bitmask to **detect the state** of your bit. It should point to the **third** bit. That bit has the weight of $2^3 = 8$. A suitable mask could be created by the following declaration:
  - **int** TheMask = 8;

How to deal with single bits

- We can also make a sequence of instructions depending on the state of your bit

```
if(FlagRegister & TheMask) {
   /* my bit is set */
} else {
   /* my bit is reset */

}
```

How to deal with single bits

- You may face the following tasks:
  - Reset your bit – you assign a zero to the bit while all the other bits must remain unchanged; we'll use the same property of the conjunction as before, but we'll use a slightly different mask – exactly as below:
    - 111111111111111111111111111111**0**111

```
FlagRegister = FlagRegister & ~TheMask;

FlagRegister &= ~TheMask;
```

How to deal with single bits

- You may face the following tasks:
  - **Set** your bit – you assign a "*one*" to your bit while all the remaining bits must remain unchanged; we'll use the following disjunction's property:
    - x|1=1
    - x|0=x

```
FlagRegister = FlagRegister | TheMask;

FlagRegister |= TheMask;
```

How to deal with single bits

- You may face the following tasks:
  - Negate your bit – you replace a “*one*” with a “*zero*” and a “*zero*” with a “*one*”. We’ll use an interesting property of the *xor* operator:
    - x ^ 1 = !x
    - x ^ 0 = x

```
FlagRegister = FlagRegister ^ TheMask;

FlagRegister ^= TheMask;
```

How to deal with single bits

- The “C” language offers yet another operation relating to single bits: **shifting**.
  - This is applied only to **integer** values and you mustn't use floats as arguments for it.
  - shifting a value one bit to the left corresponds to multiplying it by 2;
  - respectively, shifting one bit to the right is like dividing by 2

How to deal with single bits

- Bit shifting can be:
  - Logical, if all the bits of the variable are shifted; shifting takes place when you apply it to the unsigned integers;
  - Arithmetic, if the shift omits the **sign bit** – in two's complement notation, the role of the **sign bit** is played by the **highest bit** of a variable; if it’s equal to "1", the value is treated as a negative; this means than the arithmetic shift cannot change the sign of the shifted value.

How to deal with single bits

- The shift operators in the “C” language are a pair of digraphs, << and >>, clearly suggesting in which direction the shift will act.

```
Value << Bits

Value >> Bits
```

How to deal with single bits

- Let's assume the following declarations exist:
  - **int** Signed = -8, VarS;
  - **unsigned** Unsigned = 6, VarU;
- Take a look at these shifts:

```
/* equivalent to division by 2 –> VarS == -4 */
VarS = Signed >> 1;

/* equivalent to multiplication by 4 –> VarS == -32 */
VarS = Signed << 2;

/* equivalent to division by 4 –> VarU == 1 */
VarU = Unsigned >> 2;

/* equivalent to multiplication by 2 –> VarU == 12 */
VarU = Unsigned << 1;
```

How to deal with single bits

- Both operators can be used in the shortcut form as below:
  - Signed >>= 1; /* division by 2 */
  - Unsigned <<= 1; /* multiplication by 2 */

How to deal with single bits

! ~ (type) ++ -- + -	unary
* / %	
+ -	binary
<< >>	
< <= > >=	
== !=	
&	
|	
&&	
||	
= += -= *= /= %= &= ^= |= >>= <<=	

Outline

1. Flow control (continued), more data types and computer logic
   1. Loops
   2. Computer logic
2. **Aggregating data into arrays**
   1. **switch - the different face of ‘if’**
   2. Vectors: why do you need them?
3. Quiz

Case and switch vs. if

- There are no obstacles to using and maintaining code like that, but there are a few **disadvantages**:
  - The longer the cascade, the harder it is to read and understand what it’s indented for.
  - Amending the cascade is also hard: it's hard to add a new branch to it and it's hard to remove any previously created branch.

Case and switch vs. if

```
if(i == 1)
    puts("Only one?");
else if(i == 2)
    puts("I want more");
else if(i == 3)
    puts("Not bad");
else
    puts("OK");
```

- But the “C” language offers a way to make selections easier.

Case and switch vs. if

- The new instruction is called *switch*. So how does it work?
  - First, the value of the expression enclosed inside the parenthesis after the *switch* keyword is **evaluated**.
  - Then the block is searched in order to **find a literal** preceded by the *case* keyword which is equal to the value of the expression.
  - When this case is found, the instructions behind the colon are **executed**. If there's a *break* among them, the execution of the *switch* statement is terminated.

Case and switch vs. if

```
switch(i) {
	case 1: puts("Only one?"); break;
	case 2: puts("I want more"); break;
	case 3: puts("Not bad"); break;
	case 4: puts("OK");
}
```

Case and switch vs. if

- We’re allowed to place more than one *case* in front of any branch

```
switch(i) {
        case 1: puts("Only one?"); break;
        case 2: puts("I want more"); break;
        case 3:
        case 4: puts("OK");
}
```

Case and switch vs. if

- We can also assume that our program does not have an opinion when *i* values are different from the ones specified so far.
- We can put **default** case.

```
switch(i) {
	case 1: puts("Only one?"); break;
	case 2: puts("I want more"); break;
	case 3:
	case 4: puts("OK"); break;
	default: puts("Don't care");
}
```

Case and switch vs. if

- But now a few more important remarks to note:
  - the value after the case **must not be an expression** containing variables or any other entities whose values aren't known at compilation time;
  - the case branches are **scanned in the same order** in which they are specified in the program; this means that the most common selections should be placed first (in fact, this could make your program a little faster in some cases).

Outline

1. Flow control (continued), more data types and computer logic
   1. Loops
   2. Computer logic
2. **Aggregating data into arrays**
   1. switch - the different face of 'if'
   2. **Vectors: why do you need them?**
3. Quiz

Arrays – why?

- We know how declare variables that can store exactly one given value at a time.

```
int var1, var2, var3, var4, var5, var5, var7, var8, var9;
```

- Think of how convenient it would be if we could declare a variable that can store **more than one value**.

Arrays – why?

```
int numbers[5];
```

- We read this record as follows: we create a variable called *numbers*; it's intended to store five values (note the number enclosed inside brackets) of type *int* (which we know from the keyword *int* at the beginning of the declaration).
- The "C" language adopted the convention that the elements in an array are numbered **start** **from 0**.

Arrays – why?

- How do we assign a value to the chosen element of the array?

```
numbers[0] = 111;
```

Arrays – why?

- We need a value stored in the **third** element of the array and we want to assign it to the variable *i*.

```
i = numbers[2];
```

Arrays – why?

- And now we want the value of the **fifth** element to be copied to the **second** element

```
numbers[1] = numbers[4];
```

- The value inside the brackets, which selects one element of the vector, is called an **index**

Arrays – why?

- We want to calculate the sum of all the values stored in the *numbers* array.

```
int numbers[5], sum = 0, i;

for(i = 0; i < 5; i++)
   sum += numbers[i];
```

Arrays – why?

- The next task is to assign the same value (e.g. 2012) to all the elements of the array.

```
int i, numbers[5];

for(i = 0; i < 5; i++)
  numbers[i] = 2012;
```

Arrays – why?

- Now let's try to rearrange the elements of the array i.e. reverse the order of the elements: let's swap around the first and the fifth as well as the second and fourth elements. The third one we'll leave untouched.
- Question: how can we **swap the values** of two variables?

```
int variable1 = 1, variable2 = 2;

variable2 = variable1;
variable1 = variable2;
```

Arrays – why?

- Question: how can we **swap the values** of two variables?

```
int variable1 = 1, variable2 = 2, auxiliary;

auxiliary = variable1;
variable1 = variable2;
variable2 = auxiliary;
```

Arrays – why?

- It's acceptable with an array of 5 elements, but with 99 elements it certainly wouldn't work.

```
/* swap elements #1 and #5 */
auxiliary  = numbers[0];
numbers[0] = numbers[4];
numbers[4] = auxiliary;

/* swap elements #2 and #4 */
auxiliary  = numbers[1];
numbers[1] = numbers[3];
numbers[3] = auxiliary;
```

Arrays – why?

- Let’s employ the services of a *for* loop. Look carefully at how we **manipulate** the values of the indices.

```
for(i = 0; i < 2; i++) {
  auxiliary = numbers[i];
  numbers[i] = numbers[4 – i];
  numbers[4 – i] = auxiliary;
}
```

Outline

1. Flow control (continued), more data types and computer logic
   1. Loops
   2. Computer logic
2. Aggregating data into arrays
   1. switch - the different face of ‘if’
   2. Vectors: why do you need them?
3. **Quiz**

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
    int i = 1;
    int j = 0;
    while(i < 111) {
        j++;
        i *= 2;
    }
    printf("%d",j);
    return 0;
}
```

- the program outputs 7
- the program outputs 9
- the program outputs 5

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int i = 0;
   int j = 100;

   for(i = i; j; i++)
      j /= 3;
   printf("%d",i);
   return 0;
}
```

- the program outputs 3
- the program outputs 7
- the program outputs 5

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int i,j;
   for(i = 100; i <= 100; i++)
      j = i;
   printf("%d",j);
   return 0;
}
```

- the program outputs 100
- the program outputs 98
- the program outputs 99

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int i = -100, j = 200;
   if(i > 0 && j < 0)
      i++;
   else if(i < 0 && j < 0)
      i--;
   else if(i < 0 && j > 0)
      j--;
   else
      j--;
   printf("%d",i + j);
   return 0;
}
```

- the program outputs 99
- the program outputs 100
- the program outputs 101

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int i,j = 3;

   i = --j;
   j += !(i % 2);
   printf("%d",j);
   return 0;
}
```

- the program outputs 1
- the program outputs 3
- the program outputs 2

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
    int i =  1,j =  i;
    int w1,w2;
    w1 = (i > 0) && (j < 0) || (i < 0) && (j > 0);
    w2 = (i <= 0) || (j >= 0) && (i >= 0) || (j <= 0);
    printf("%d",w1 == w2);
    return 0;
}
```

- the program outputs -1
- the program outputs 1
- the program outputs 0

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int i,j=0;
   for(i = 0; i < 10; i+=2)
      switch(i) {
         case 0: j++; break;
         case 2: j++;
         case 4: j++; break;
         default: j--;
      }
   printf("%d",j);
   return 0;
}
```

- the program outputs 2
- the program outputs 1
- the program outputs 0

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int i,t[5];
   for(i = 0; i < 5; i++)
      t[i] = 2 * i;
   i = 0;
   for(i = 0; i < 5; i++)
      i += t[i];
   printf("%d",i);
   return 0;
}
```

- the program outputs 13
- the program outputs 14
- the program outputs 12