# Selection

**CSC100 Introduction to programming in C/C++**

## Table of Contents



## 1 README

- Algorithms are the core of programming
- Example for an algorithm: "When you come to a STOP sign, stop."
- The human form of algorithm is heuristics
- Example for a heuristic: "To get to the college, go straight."
- For programming, you need both algorithms and heuristics
- Useful tools to master = design algorithms (and heuristics)
  - Pseudocode (task flow description)
  - Flow charts (task flow visualization)
- In this chapter, we go beyond simple statements and include logical conditions (like in the algorithm example above)
- This section follows chapter 3 in Davenport/Vine (2015) and chapters 4 and 5 in King (2008).

## 2 Operators in C

- Mathematically, operators are really functions: `f(i,j)=i+j`
- C has many operators, both unary (`-1`) and binary (`1+1`)
- Different programming languages differ greatly rgd. operators

- Types of operators in C:

| OPERATOR | WHY | EXAMPLES | EXPRESSION |
|---|---|---|---|
| Arithmetic | To compute | * + - / % | i * j + k |
| Relational | To compare | < > <= >= | i > j |
| Equality | To compare (in/equality) | = ! | i == j |
| Logical | To confirm (truth) | && ! | i && j |
| Assignment | To change | = | i = j |
| Increment/decrement | To change stepwise | ++, +- | ++i |

- Conditional operators used in C:

| OPERATOR | DESCRIPTION | EXPRESSION | VALUE |
|---|---|---|---|
| == | Equal | 5 == 5 | true |
| != | Not equal | 5 != 5 | false |
| > | Greater than | 5 > 5 | false |
| < | Less than | 5 < 5 | false |
| >= | Greater than or equal to | 5 >= 5 | true |
| <= | Less than or equal to | 5 <= 5 | true |

- The value of an evaluated conditional operator is **Boolean** (logical)

# 3 Pseudocode

## 3.1 Overview

- Pseudocode is a method to quickly write down/analyze an algorithm or a heuristic
- The prefix pseudo- comes from Ancient Greek ψευδής, meaning "lying", "false" or "untrue", as in "pseudoscience" or "pseudonym"

## 3.2 Example

- Example: player problem statement in 1 below:

> "Drink a health potion when a character's health is 100 or less. If health reaches 100 or more, resume battle."

- Given the problem 1, this is the pseudocode 1[1]:

```
if health <= 100
  Drink health potion
else
  Resume battle
end if
```

- The code in 1 would not compile as a C program (you can test yourself: which mistakes would the compiler find?[2])
- The conceptual "trick" with generating pseudocode from a prose description is to identify the condition so that you can perform a comparison (= apply a conditional operator) and act accordingly

## 3.3 Practice

- `[X]`

  Turn the following problem 1 into pseudocode!

  > "Allow a player to deposit or withdraw gold from a game bank account."

- `[X]`

  Given the problem 1, the pseudocode could be 1:

```
if action == deposit
   Deposit gold into account
else
   Withdraw gold from account
end if
```

- `[ ]`

  The problem from 1 is refined in 1:

  > "Allow a player to deposit or withdraw gold from a game bank account. If a player elects to withdraw gold, ensure that sufficient funds exist."

- `[ ]`

  Given the refined problem 1, the pseudocode could be 1:

```
if action == deposit
   Deposit gold into account
else
   if balance < amount requested
      Insufficient funds
   else
      Withdraw gold
   end if
end if
```

  Or it could be 1:

```
if action == deposit
   Deposit gold into account
```

```
else
   if balance >= amount requested
      Withdraw gold
   else
      Insufficient funds
   end if
end if
```

- Pseudocode does not need to compile or run so it is closer to a heuristic than to an exact algorithm.
- Code however needs to be exact and is always algorithmic

# 4 Flowcharts

## 4.1 Overview

- Flowcharts are popular among computing analysts and programmers
- Flowcharts are a special case of process models
- Process modeling is a key 21st century skill

- Rather than use flowcharts, we use BPMN - Business Process Model and Notation - see figure 1 below for an overview of the whole language (you can get this poster at SAP Signavio).

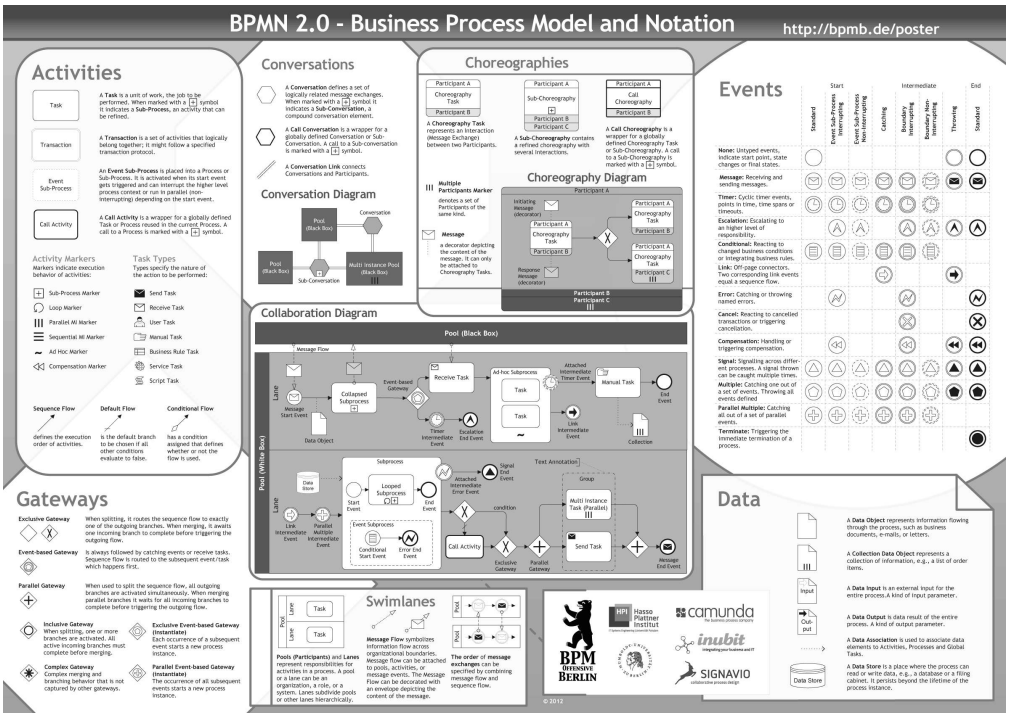


Figure 1: (Free) BPMN 2.0 poster from SAP Signavio

## 4.2 SAP Signavio

- SAP Signavio is a state-of-the-art process modeling environment

- It includes process mining and workflow management tools. The figure 2 below shows the app dashboard.

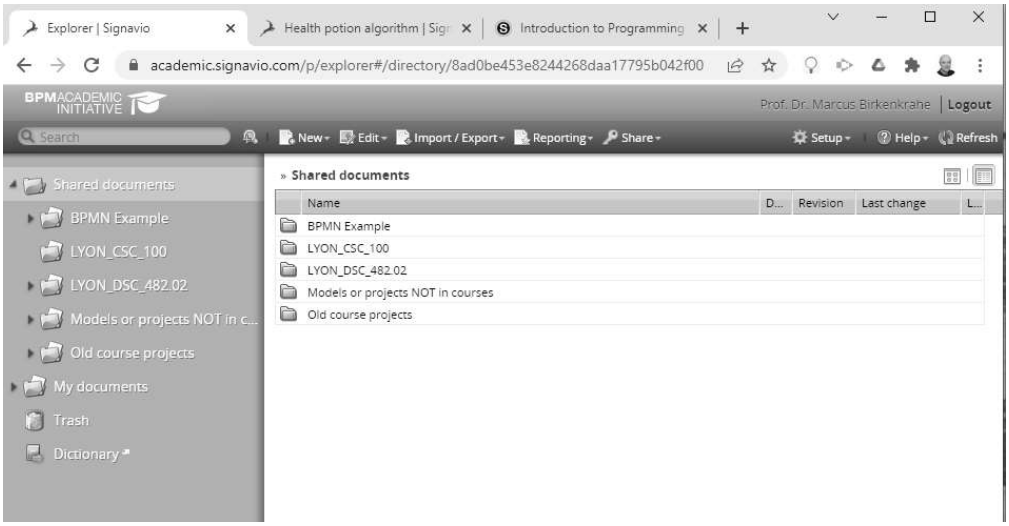


Figure 2: SAP Signavio dashboard / explorer

- Register in my Signavio workspace if you haven't done it yet
- Use the link in Schoology to register

## 4.3 Practice

- [X] Open Signavio Process Manager
- [X]

  Create a simple BPMN based on the battle 1.

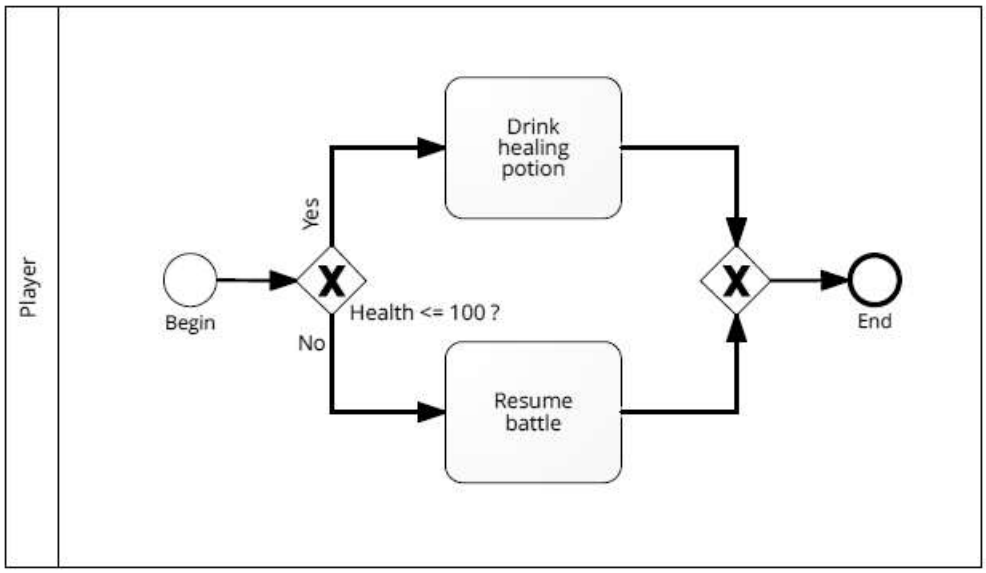


Figure 3: Health potion algorithm (battle problem) as BPMN diagram

- [X] In the next models, change "Insufficient funds" to "Do not withdraw gold" (tasks need to be articulated as active).

- [X]

  Create a model based on 1.

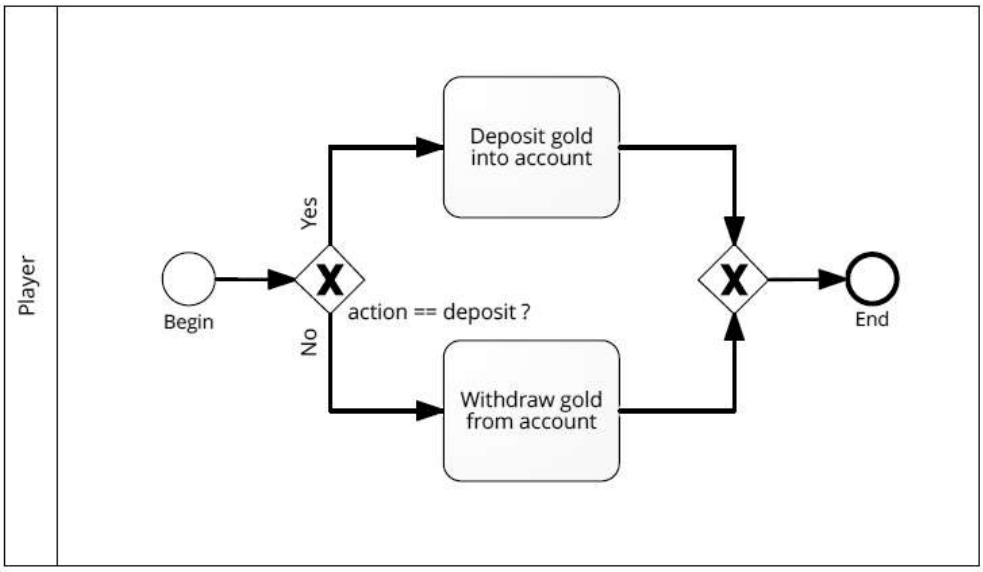


Figure 4: Gold deposit algorithm as BPMN diagram - version 1

- [X]

  Create a model based on 1

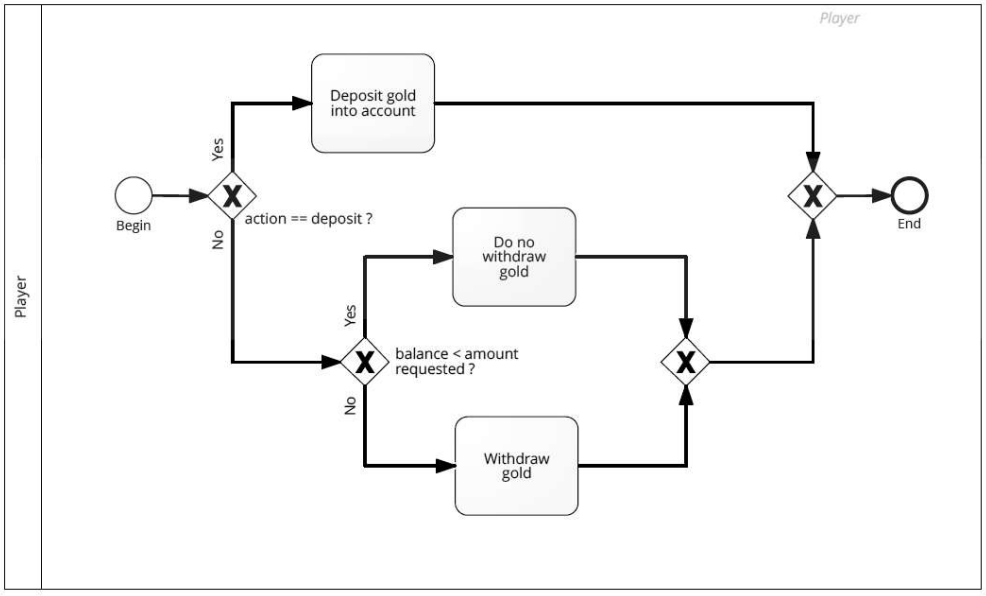


Figure 5: Gold deposit algorithm as BPMN diagram - version 2

- What changes if you use 1 instead?

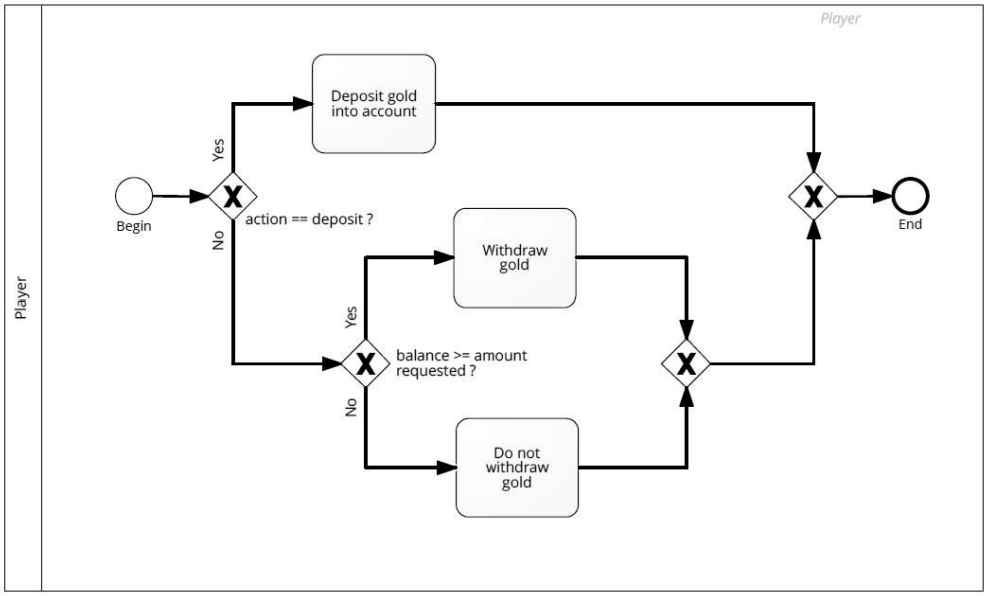


Figure 6: Gold deposit algorithm as BPMN diagram - version 3

### 4.4 Points to remember

- Every model needs a pool = process owner
- Conditions become gateways
- Use active sentences for tasks
- When the flow is split, it must be rejoined
- All elements must be named
- Do not change the size of elements
- All elements can be "overloaded"

### 4.5 BPMN elements

- Roles (pools, lanes, participants)
- Tasks (things to do)
- Events (status)
- Flow (between tasks or events)
- Gateways (decision points, condition check)

## 5 Simple and nested 'if' structures

### 5.1 Overview and example

- If structure in C is very similar to pseudocode
- 1 is the C version of the pseudocode 1 from earlier.

```
if (health <= 100)
  // drink health potion
```

```
else
  //resume battle
```

- Differences: condition needs parentheses; no "end if" statement
- The health check results in a Boolean answer: `true` or `false`
- To run, the program needs a declaration of the `health` variable
- Multiple statements need to be included in braces

- Here is a version that will run. The variable has been initialized

```
int health = 99;

if (health <= 100) {
  // drink health potion
  printf("\nDrinking health potion!\n");
 }
 else {
   // resume battle
   printf("\nResuming battle!\n");
 }
```

### 5.2 Battle by numbers

- The last version 1 below reads input from a file.

```
echo "1" > drink
```

- We only have two simple if structures, and no nesting.

```
int response = 0; // initialize response

puts("\n\tIn-Battle Healing\n\n1:\tDrink health potion\n\n2:\tResume battle\n");

printf("\nEnter your selection: ");
scanf("%d", &response);

if (response == 1)
  printf("\nDrinking health potion!\n");

if (response == 2)
  printf("\nResuming battle!\n");
```

- You can also tangle the file and run `battle.c` from the commandline:

```
C-c C-v t
gcc -o battle battle.c
battle
```

# 6 Boolean algebra

- [ ] What is algebra about?[3]

- Algebra allows you to form small worlds with fixed laws so that you know exactly what's going on - what the output must be for a given input. This certainty is what is responsible for much of the magic of mathematics.
- Boole's (or Boolean) algebra, or the algebra of logic, uses the values of TRUE (or 1) and FALSE (or 0) and the operators AND (or "conjunction"), OR (or "disjunction"), and NOT (or "negation").

- Truth tables are the traditional way of showing Boolean scenarios:

| p | q | p AND q |
|---|---|---|
| TRUE | TRUE | TRUE |
| TRUE | FALSE | FALSE |
| FALSE | TRUE | FALSE |
| FALSE | FALSE | FALSE |

| p | q | p OR q |
|---|---|---|
| TRUE | TRUE | TRUE |
| TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE |
| FALSE | FALSE | FALSE |

| p | NOT p |
|---|---|
| TRUE | FALSE |
| FALSE | TRUE |

- Using the three basic operators, other operators can be built. In electronics, and modeling, the "exclusive OR" operator or "XOR", is e.g. equivalent to (p AND NOT q) OR (NOT p AND q)

| p | q | p XOR q | P = p AND (NOT q) | Q = (NOT p) AND q | P OR Q |
|---|---|---|---|---|---|
| TRUE | TRUE | TRUE | FALSE | FALSE | FALSE |
| TRUE | FALSE | FALSE | TRUE | FALSE | TRUE |
| FALSE | TRUE | FALSE | FALSE | TRUE | TRUE |
| FALSE | FALSE | TRUE | FALSE | FALSE | FALSE |

- XOR is the operator that we've used in our BPMN models for pseudocode as a gateway operator - only one of its outcomes can be true but never both of them
- [ ]

  What is the result of each of the following expressions?

| Expression | Result |
|---|---|
| `3 = 3 AND 4 = 4` | |
| `3 = 4 AND 4 = 4` | |

| Expression | Result |
|---|---|
| 4 = 3 OR 4 = 4 | |
| NOT ( 4 == 4 ) | |
| NOT ( 3 == 4 ) | |

- Algebraic operations are more elegant and insightful than truth tables. Watch "Proving Logical Equivalences without Truth Tables" (2012) as an example.

# 7 Order of operator operations

- In compound operations (multiple operators), you need to know the order of operator precedence
- C has almost 50 operators. The most unusual are compound increment/decrement operators[4]:

| STATEMENT | COMPOUND | PREFIX | POSTFIX |
|---|---|---|---|
| i = i + 1; | i += 1; | ++i; | i++; |
| j = j - 1; | j -= 1; | –i; | i–; |

- ++ and – have side effects: they modify the values of their operands: ++i yields i+1 and increments i:

```
int i = 1;
printf("i is %d\n", ++i);  // prints "i is 2"
printf("i is %d\n", i);  // prints "i is 2"
```

```
i is 2
i is 2
```

```
int j = 1;
printf("j is %d\n", j++);  // prints "j is 1"
printf("j is %d\n", j);  // prints "j is 2"
```

```
j is 1
j is 2
```

- Here is another illustration with an assignment of post and prefix increment operators:

```
int num1 = 10, num2 = 0;
puts("start: num1 = 10, num2 =0");

num2 = num1++;
printf("num2 = num1++, so num2 = %d, num1 = %d\n", num2, num1);

num1 = 10;
num2 = ++num1;
printf("num2 = ++num1, so num2 = %d, num1 = %d\n", num2, num1);
```

```
start: num1 = 10, num2 =0
num2 = num1++, so num2 = 10, num1 = 11
num2 = ++num1, so num2 = 11, num1 = 11
```

- The table 1 shows a partial list of operators and their order of precedence from 1 (highest precedence, i.e. evaluated first) to 5 (lowest precedence, i.e. evaluated last)

| ORDER | OPERATOR | SYMBOL | ASSOCIATIVITY |
|---|---|---|---|
| 1 | increment (postfix) | ++ | left |
| | decrement (postfix) | -- | |
| 2 | increment (prefix) | ++ | right |
| | decrement (prefix) | -- | |
| | unary plus | + | |
| | unary minus | - | |
| 3 | multiplicative | * / % | left |
| 4 | additive | + - | left |
| 5 | assignment | = *= /= %= += -= | right |

- Left/right associativity means that the operator groups from left/right. Examples:

| EXPRESSION | EQUIVALENCE | ASSOCIATIVITY |
|---|---|---|
| i - j - k | (i - j) - k | left |
| i * j / k | (i * j) / k | left |
| -+j | - (+j) | right |
| i %=j | i = (i % j) | right |
| i +=j | i = (j + 1) | right |

- [ ]

  Make sure to write some of these out yourself and run examples. I found `%=` quite challenging: a modulus and assignment operator. `i %= j` computes `i%j` (i modulus j) and assigns it to `i`.

  What is the value of `i` after running the code below?

```
int i = 10, j = 5;
i%=j;
printf("i was 10 and is now %d = 10 %% 5\n", i);
```

```
i was 10 and is now 0 = 10 % 5
```

- [ ]

Let's write a program 1 to check the equivalence claimed in the table 1.

| OPERATOR | EXAMPLE | EQUIVALENT TO |
|---|---|---|
| += | a += b | a = a + b |
| -= | a -= b | a = a - b |
| *= | a *= b | a = a * b |
| /= | a /= b | a = a / b |
| %= | a %= b | a = a % b |

```
int a = 10, b = 2;

printf("10 += 2 is %d\n", a += b); a = 10; b = 2;
printf("10 -= 2 is %d\n", a -= b); a = 10; b = 2;
printf("10 *= 2 is %d\n", a *= b); a = 10; b = 2;
printf("10 /= 2 is %d\n", a /= b); a = 10; b = 2;
printf("10 %%= 2 is %d\n", a %= b);
```

```
10 += 2 is 12
10 -= 2 is 8
10 *= 2 is 20
10 /= 2 is 5
10 %= 2 is 0
```

- [ ]

Using the table 1, parenthesize the following expression:

```
a = b += c++ -d + --e / -f
```

Group 1:

```
a = b += (c++) -d + --e / -f
```

Group 2:

```
a = b += (c++) -d + (--e) / (-f)
```

Group 3:

```
a = b += ((c++) -d) + ((--e) / (-f))  // -,+ group from left to right
```

Group 4:

```
(a = (b += ((c++) -d) + ((--e) / (-f)))) // =, += group from right to left
```

Much better to write this like as a series of commands:

```
c++;  // c = c + 1
e--;  // e = e - 1
-f;   // f = -f
c = c - d;
e = e / f;
b = b + c + e;
a = b;
```

# 8 Compound if structures and input validation

## 8.1 Booleans in C

- C evaluates all non-zero values as TRUE, and all zero values as FALSE:

```
if (3) puts("3 is TRUE"); // non-zero expression
if (!0) puts("0 is FALSE"); // !0 is literally non-zero
```

```
3 is TRUE
0 is FALSE
```

- The Boolean operators AND, OR and NOT are represented in C by the logical operators `&&`, `||` and `!`, respectively

## 8.2 ! operator

- The ! operator is a unary operator that is evaluated from the left. It is TRUE when its argument is FALSE, and vice versa.
- [ ] If `i = 100`, what is `!i`?
- [ ] If `j = 0.5e-15`, what is `!i`?

- Let's check!

```
int i = 100;
double j = 1.e-15;
printf("!%d is %d because %d is non-zero!\n", i, !i, i);
printf("!(%.1e) is %d because %.1e is non-zero!\n", j, !j, j);
```

```
!100 is 0 because 100 is non-zero!
!(1.0e-015) is 0 because 1.0e-015 is non-zero!
```

## 8.3 && operator

- Evaluates a Boolean expression from left to right
- Value is TRUE if and only if both sides of the operator are TRUE

- Examples:

```
if ( 3 > 1 && 5 < 10 ) // = TRUE AND TRUE = TRUE
  printf("The entire expression is true.\n");
```

```
The entire expression is true.
```

```
if (!(3 > 5 && 5 < 5) ) // = NOT (FALSE AND FALSE) = TRUE
  printf("The entire expression is false.\n");
```

```
The entire expression is false.
```

## 8.4 || operator

- Evaluates a Boolean expression from left to right
- It is FALSE if and only if both sides of the operator are FALSE
- It is TRUE if either side of the operator is TRUE

- Examples:

```
if ( 3 > 5 || 5 <= 5 ) // TRUE OR TRUE = TRUE
  printf("The entire expression is true.\n");
```

```
The entire expression is true.
```

```
if ( 3 > 5 || 6 < 5 ) { }// = FALSE OR FALSE = FALSE
 else printf("The entire expression is false.\n");
```

```
The entire expression is false.
```

- Practice that using the GDrive notebook!

## 8.5 Checking for upper and lower case

- Characters are represented by ASCII[5] character sets
- E.g. `a` and `A` are represented by the ASCII codes 97 and 65, resp.
- `[ ]`

  Let's check that.

```
echo "a A" > ascii
```

```
char a, A;
scanf("%c %c", &a, &A);
printf("The ASCII value of %c is %d\n", a, a);
printf("The ASCII value of %c is %d\n", A, A);
```

- User-friendly programs should use compound conditions to check for both lower and upper case letters:

```
if (response == 'A' || response == 'a')
```

- Practice that using the GDrive notebook!

### 8.6 Checking for a range of values

- To validate input, you often need to check a range of values
- This is a common use of compound conditions, logical and relational operators

- We first create an input file `num`.

```
echo 5 > num
```

- `[ ]`

  What does the code in 1 do? Will it run? What will the output be for our choice of input?

```
int response = 0;

scanf("%d", &response);

if ( response < 1 || response > 10 ) {
  puts("Number not in range.");
 } else {
  puts("Number in range.");
 }
```

```
Number in range.
```

- Head to the practice notebook for some challenging exercises!

## 9 `IN PROGRESS` The switch structure

- The `switch` statement is fairly complex: it combines conditional expressions, constant expressions, default and break statements.

```
switch ( expression ) {
case constant expression : statements
...
case constant expression : statements
default : statements
}
```

1. **Controlling expression:** an integer expression in parentheses. Characters are treated as integers in C and cannot be tested.
2. **Case labels:** each case begins with a constant expression - this is like any other expression except that it cannot contain variables or function calls.

3. **Statements:** any number of statements. No braces required around the statements. The last statement is usually `break` to close the `case`.

## 9.1 Simple example

- In the example code 1, the `grade` is set in the variable declaration. Depending on the value, a `case` is triggered and the corresponding statements are executed.
- [ ]

  What is the output of the code in 1 for different values: 5,3,0,-1,0.5?

  What does the program implement?

```
int grade = 3;

switch (grade) {
 case 4:
 case 3:
 case 2:
 case 1:
   printf("Passing");
   break;
 case 5:
 case 6:
   printf("Failing");
   break;
 default:
   printf("Illegal grade");
   break;
 }
```

```
Passing
```

- [ ]

  Q: What does this program implement?

  > Answer: The program reflects "passing" grades 4,3,2,1, and "failing" grade 0. Any other grade value is not allowed. (This happens to be the European grade scale, which is A = 1 to D = 4, and F = 5 or 6.)

- You can also put several case labels on the same line as in 1, which is otherwise identical to 1.

```
int grade = 3;

switch (grade) {
 case 4: case 3: case 2: case 1:
   printf("Passing");
   break;
 case 5:
 case 6:
   printf("Failing");
   break;
 default:
   printf("Illegal grade");
```

```
  break;
 }
```

```
Passing
```

- It is not possible to write a case label for a range of values.
- The default case (when none of the case expressions apply) is optional, and it does not have to come last.

### 9.2 The role of the `break` statement

- The `switch` statement is a controlled jump. The case label is but a marker indicating a position within the switch.
- `[ ]`

  Let's run the previous program again, without the `break` statements. What do you think the output will be?

```
int grade = 3;

switch (grade) {
 case 4:
 case 3:
 case 2:
 case 1:
   printf("Passing");
 case 5:
 case 6:
   printf("Failing");
 default:
   printf("Illegal grade");
 }
```

```
PassingFailingIllegal grade
```

- `[ ]`

  What happens without the `break` statements?

  > Answer: When the last statement in a case has been executed, control falls through to the first statement in the following case; its case label is ignored. Without `break` (or some other jump statement, like `return` or `goto`, control flows from one case to the next.

- Deliberate falling through (omission of `break`) should be indicated with an explicit comment.

# 10 References


- Davenport/Vine (2015) C Programming for the Absolute Beginner (3ed). Cengage Learning.
- GVSUmath (Aug 10, 2012). Proving Logical Equivalences without Truth Tables [video]. URL: youtu.be/iPbLzl2kMHA.
- Kernighan/Ritchie (1978). The C Programming Language (1st). Prentice Hall.
- King (2008). C Programming - A modern approach (2e). W A Norton.
- Orgmode.org (n.d.). 16 Working with Source Code [website]. URL: orgmode.org

# Footnotes:

[1] In Org mode, you can use the language as an example header argument to enable syntax highlighting. For pseudocode, this will of course not work perfectly, since most syntax elements are not in C.

[2] Undeclared variable `health`, missing closure semi-colons after the statements, functions `Drink` and `Resume` not known, and more.

[3] Algebra is the branch of mathematics that allows you to represent problems in the form of abstract, or formal, expressions. The abstraction is encapsulated in the notion of a variable (an expression of changing value), and of an operator acting on one or more variables (a function having the variable as an argument, and using it to compute something).

[4] These operators were inherited from Ken Thompson's earlier B language. They are not faster just shorter and more convenient.

[5] ASCII stands for the American Standard Code for Information Interchange.

Author: Marcus Birkenkrahe
Created: 2022-03-27 Sun 21:59
Validate