Selection

CSC100 Introduction to programming in C/C++ (Summer 2022)

Table of Contents

1. DONE README

- In this section of the course, we go beyond simple statements and turn to program flow and evaluation of logical conditions
- This section follows chapter 3 in Davenport/Vine (2015) and chapters 4 and 5 in King (2008).

2. DONE Preamble

- **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 when designing algorithms:
  - **Pseudocode** (task flow description)
  - **Visual modeling** (task flow visualization)

3. DONE Operators in C

- Mathematically, operators are really functions: `f(i,j)=i+j`
- C has many operators, both unary (`-1`) and binary (`1+1`)

- Types of operators in C:

OPERATOR	WHY	EXAMPLES	EXPRESSION
Arithmetic	compute	* + - / %	i * j + k
Relational	compare	< > <= >=	i > j
Equality	compare (in/equality)	`= !`	i == j

OPERATOR	WHY	EXAMPLES	EXPRESSION
Logical	confirm (truth)	&&	i && j
Assignment	change	=	i = j
Increment/decrement	change stepwise	++, +-	++i

- Conditional operators used in C are important for program flow:

OPERATOR	DESCRIPTION	EXPRESSION	BOOLEAN 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) - e.g. `2==2` evaluates as `TRUE` or `1`.

4. DONE Operators in other languages

- Different programming languages differ greatly rgd. operators. For example, in the language R, the `|>` operator ("pipe") passes a data set to a function[1].

```
## pipe data set into function
mtcars |> head()  ## same as head(mtcars)
```

Output:

```
                  mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

- You met the > operator of the `bash` shell language that redirects standard output to a file:

```
## create empty file called "empty"
> empty
```

5. DONE Pseudocode

5.1. Overview

- **Pseudocode** is a method to write down/analyze an *algorithm* or a *heuristic* without having to bother with *syntax*
- The prefix pseudo- comes from Ancient Greek ψευδής, meaning "lying", "false" or "untrue", as in "pseudoscience" or "pseudonym"
- 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

5.2. 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[2]:

```
if health is less than 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?[3])
- The conceptual "trick" with generating pseudocode from a prose description is to identify the **logical condition** so that you can perform a comparison (= apply a **conditional operator**)

- The pseudocode 1 leads to the condition `health < 100`:

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

- Notice that you could also use another operator: >= This operator would have had the same effect but it is not what you were supposed to code.

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

- **Rule:** when making models (via **abstraction**), always stay as close to the problem description as possible - in terms of language, logic, tone, etc. If you're unsure, ask.

5.3. Let's practice!

Get the practice file from GitHub:

```
wget tinyurl.com/prsmbhvh -O pseudo.org -o log
```

6. DONE Process models

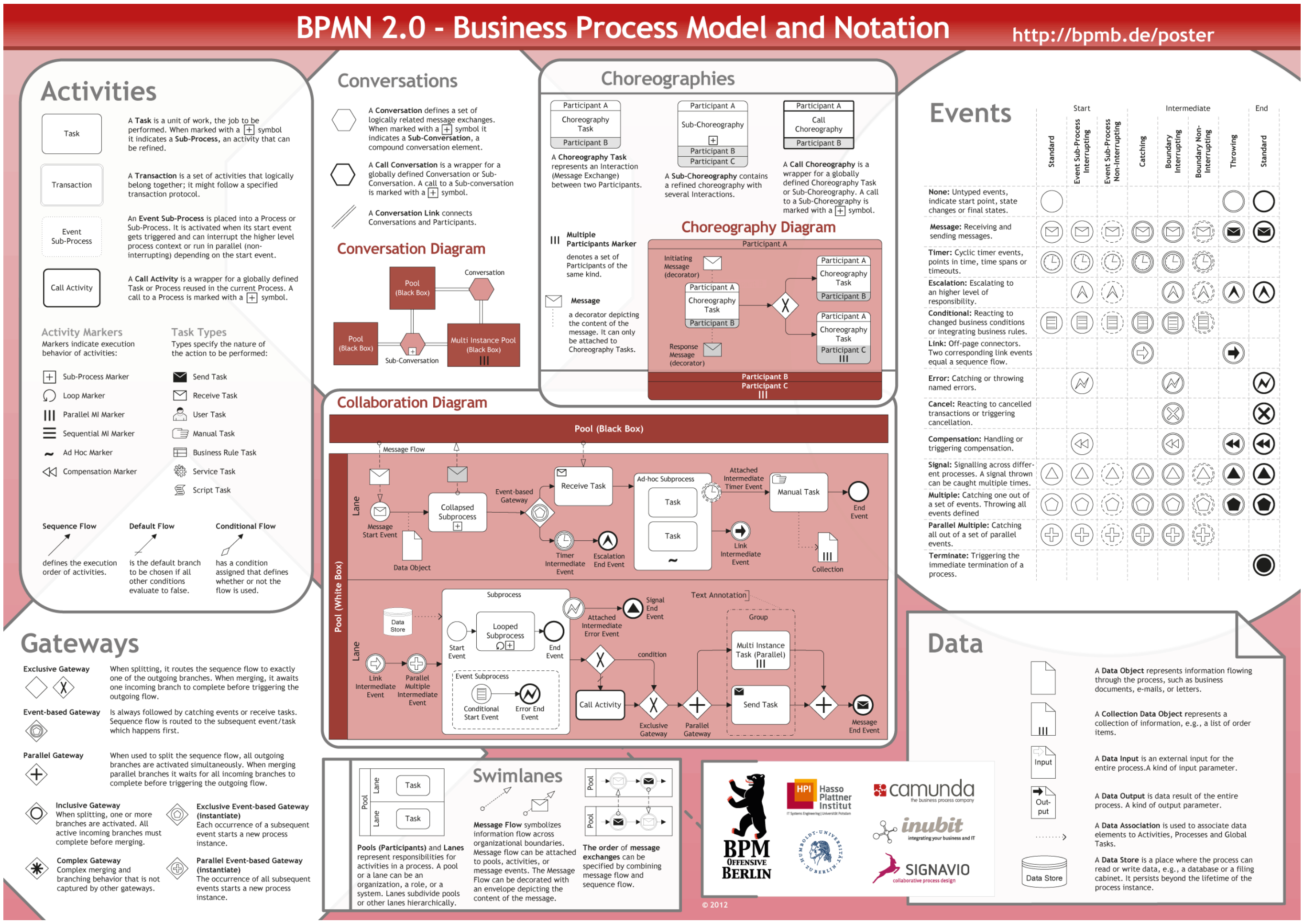
6.1. Objectives

- [X] Understand what a process manager (software) is and does
- [X] Learn the basics of Business Process Model and Notation (BPMN)
- [X] Learn how to create simple models in BPMN
- [X] Apply modeling skills to creating pseudocode models
- [X] Adding Pseudocode/BPMN models to future programming assignments will give you bonus points (5 extra points per assignment)

6.2. 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, because **process** is the central paradigm of modern work organization

- 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).



BPMN 2.0 - Business Process Model and Notation http://bpmb.de/poster

Activities

A **Task** is a unit of work, the job to be performed. When marked with a ⊞ symbol it indicates a **Sub-Process**, an activity that can be refined.

A **Transaction** is a set of activities that logically belong together; it might follow a specified transaction protocol.

An **Event Sub-Process** is placed into a Process or Sub-Process. It is activated when its start event gets triggered and can interrupt the higher level process context or run in parallel (non-interrupting) depending on the start event.

A **Call Activity** is a wrapper for a globally defined Task or Process reused in the current Process. A call to a Process is marked with a ⊞ symbol.

Activity Markers
Markers indicate execution behavior of activities:

Sub-Process Marker
Loop Marker
Parallel MI Marker
Sequential MI Marker
Ad Hoc Marker
Compensation Marker

Task Types
Types specify the nature of the action to be performed:

Send Task
Receive Task
User Task
Manual Task
Business Rule Task
Service Task
Script Task

Sequence Flow defines the execution order of activities.

Default Flow is the default branch to be chosen if all other conditions evaluate to false.

Conditional Flow has a condition assigned that defines whether or not the flow is used.

Gateways

Exclusive Gateway When splitting, it routes the sequence flow to exactly one of the outgoing branches. When merging, it awaits one incoming branch to complete before triggering the outgoing flow.

Event-based Gateway Is always followed by catching events or receive tasks. Sequence flow is routed to the subsequent event/task which happens first.

Parallel Gateway When used to split the sequence flow, all outgoing branches are activated simultaneously. When merging parallel branches it waits for all incoming branches to complete before triggering the outgoing flow.

Inclusive Gateway When splitting, one or more branches are activated. All active incoming branches must complete before merging.

Complex Gateway Complex merging and branching behavior that is not captured by other gateways.

Exclusive Event-based Gateway (instantiate) Each occurrence of a subsequent event starts a new process instance.

Parallel Event-based Gateway (instantiate) The occurrence of all subsequent events starts a new process instance.

Conversations

A **Conversation** defines a set of logically related message exchanges. When marked with a ⊞ symbol it indicates a **Sub-Conversation**, a compound conversation element.

A **Call Conversation** is a wrapper for a globally defined Conversation or Sub-Conversation. A call to a Sub-conversation is marked with a ⊞ symbol.

A **Conversation Link** connects Conversations and Participants.

Conversation Diagram

Choreographies

A **Choreography Task** represents an Interaction (Message Exchange) between two Participants.

A **Sub-Choreography** contains a refined choreography with several Interactions.

A **Call Choreography** is a wrapper for a globally defined Choreography Task or Sub-Choreography. A call to a Sub-Choreography is marked with a ⊞ symbol.

Multiple Participants Marker denotes a set of Participants of the same kind.

Message a decorator depicting the content of the message. It can only be attached to Choreography Tasks.

Choreography Diagram

Collaboration Diagram

Swimlanes

Pools (Participants) and **Lanes** represent responsibilities for activities in a process. A pool or a lane can be an organization, a role, or a system. Lanes subdivide pools or other lanes hierarchically.

Message Flow symbolizes information flow across organizational boundaries. Message flow can be attached to pools, activities, or message events. The Message Flow can be decorated with an envelope depicting the content of the message.

The order of **message exchanges** can be specified by combining message flow and sequence flow.

Events

None: Untyped events, indicate start point, state changes or final states.
Message: Receiving and sending messages.
Timer: Cyclic timer events, points in time, time spans or timeouts.
Escalation: Escalating to an higher level of responsibility.
Conditional: Reacting to changed business conditions or integrating business rules.
Link: Off-page connectors. Two corresponding link events equal a sequence flow.
Error: Catching or throwing named errors.
Cancel: Reacting to cancelled transactions or triggering cancellation.
Compensation: Handling or triggering compensation.
Signal: Signalling across different processes. A signal thrown can be caught multiple times.
Multiple: Catching one out of a set of events. Throwing all events defined
Parallel Multiple: Catching all out of a set of parallel events.
Terminate: Triggering the immediate termination of a process.

Data

A **Data Object** represents information flowing through the process, such as business documents, e-mails, or letters.

A **Collection Data Object** represents a collection of information, e.g., a list of order items.

A **Data Input** is an external input for the entire process.A kind of input parameter.

A **Data Output** is data result of the entire process. A kind of output parameter.

A **Data Association** is used to associate data elements to Activities, Processes and Global Tasks.

A **Data Store** is a place where the process can read or write data, e.g., a database or a filing cabinet. It persists beyond the lifetime of the process instance.

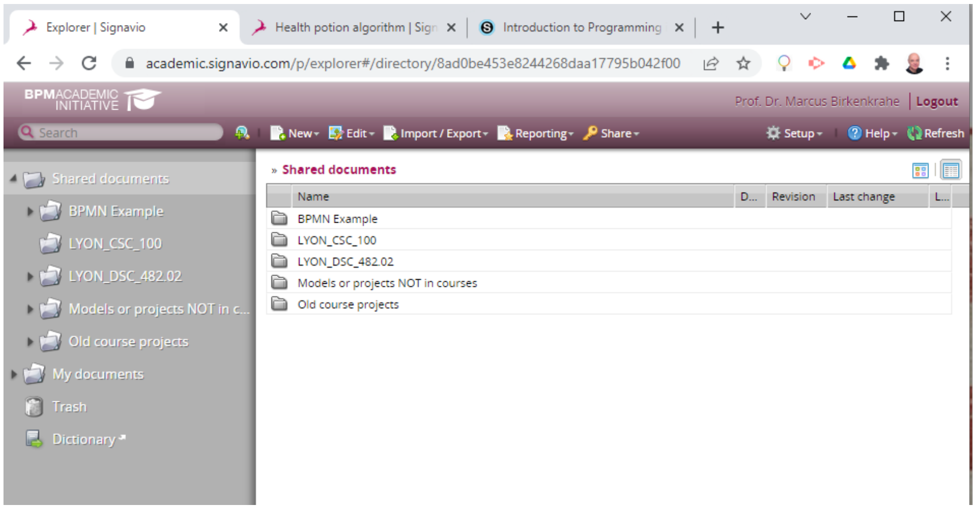
© 2012

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

6.3. 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.



Name	D...	Revision	Last change	L...
BPMN Example				
LYON_CSC_100				
LYON_DSC_482.02				
Models or projects NOT in courses				
Old course projects				

Figure 2: SAP Signavio dashboard / explorer

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

6.4. 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"

6.5. BPMN elements

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

6.6. Let's practice

Download the [raw] Org-mode file `bpmn.org` from GitHub:

```
wget tinyurl.com/58mw8wuc -O bpmn.org -o log
file bpmn.org
```

Download the [raw] image file `battle.png` from GitHub

```
wget tinyurl.com/2s3f3t9c -O battle.png -o log
file battle.png
```

The new `file` command provides file type information. It should tell you that `bpmn.org` is a `text`, and that `battle.png` is a `PNG` file.

Now open the file `bpmn.org` in Emacs.

7. DONE Simple and nested 'if' structures

7.1. Overview and example

- `If` statement structure in C is very similar to pseudocode `If`

- 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 `{...}`

- The source code 1 will run. The variable has been declared and initialized:

```
int health = 101;

if (health <= 100) {
  // drink health potion
  puts("This is what you do:");
  printf("Drinking health potion!\n");
 }
 else {
   // resume battle
   puts("This is what you do:");
   printf("Resuming battle!\n");
 }
```

```
This is what you do:
Resuming battle!
```

7.2. Single vs. nested IF structures

- In the example 1, the IF statements are evaluated independently, case by case. It does not matter if any of them fails. We'll see that there is a C control structure for that.

```
if ( i == 1 )
// do one thing
```

```
if ( i == 2)
// do another thing
```

The figure 3 shows the BPMN model for this program:

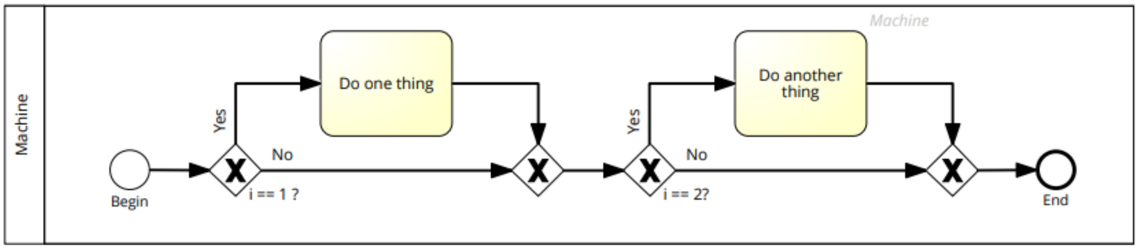


Figure 3: Single IF statements

- In the example 1, the second part of the IF statement is entered only if the first condition fails.

```
if ( i == 1 ) {
   // do one thing
}
else if ( i == 2) {
     // do another thing
}
```

The figure 4 shows the BPMN model for this program:

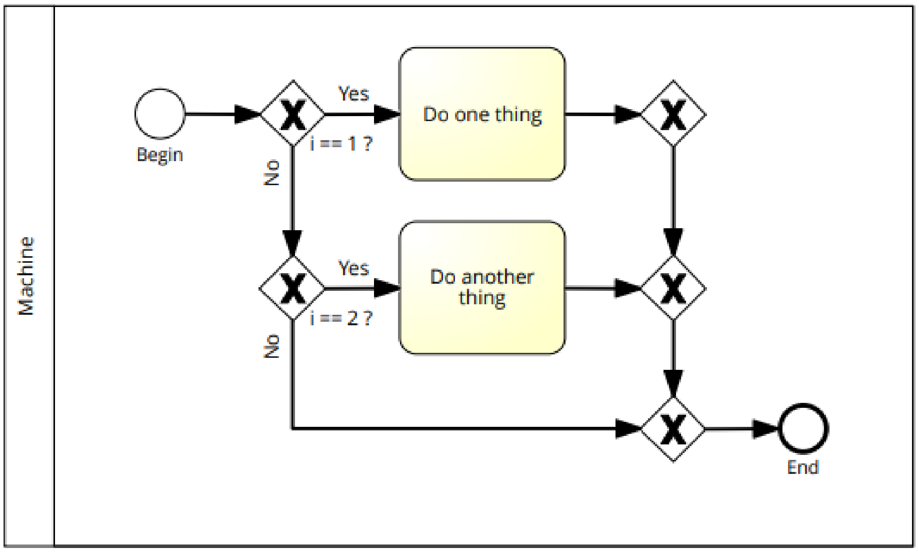


Figure 4: Single IF statements

- Which one of these you implement, depends strongly on the problem and on your performance requirements (they're quite different in speed - which you do you think performs better?)

7.3. Let's practice!

Download the practice file `battle.org` from GitHub and check its file type:

```
wget tinyurl.com/z526bwuh -O battle.org -o log
file battle.org
```

8. DONE Boolean algebra

- [ ] What is algebra about?[4]
- 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	FALSE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	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

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

9. DONE 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[5]:

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: the *prefix* operator `++i` increments `i+1` and then fetches the value `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
```

- The *postfix* operator `++j` also means `j = j + 1` but here, the value of `j` is fetched, and then incremented.

```
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

- 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 = 10` after running the code below?

```
int i = 10, j = 5;
i %= j; // compute modulus of i and j and assigns it to i
printf("i was 10 and is now %d = 10 %% 5\n", i);
```

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

10. DONE Compound if structures and input validation

10.1. Download the practice file

- Get the file with `wget`
- Check the file type with `file`
- Display the first 2 lines with `head`

```
wget tinyurl.com/2y6wut43 -O ops.org -o log
file ops.org
head -n 2 ops.org
```

10.2. Booleans in C

- C evaluates all non-zero values as `TRUE` (`1`), and all zero values as `FALSE` (`0`):

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

  ```
  3 is TRUE
  ```

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

10.3. ! operator (logical NOT)

- The ! operator is a "unary" operator that is evaluated from the left. It is `TRUE` when its argument is `FALSE` (`0`), and it is `FALSE` when its argument is `TRUE` (non-zero).
- `[X]`

  If `i = 100`, what is `!i`?

  The Boolean value of `100` is TRUE. Therefore, `!100` = `!TRUE` = `FALSE`

- `[X]`

  If `j = 1.0e-15`, what is `!j`?

  The Boolean value of `1.0e-15` is TRUE. Therefore, `!1.0e-15` = `!TRUE` = `FALSE`

- `[ ]`

  Let's check!

  ```
  // declare and assign variables
  int i = 100;
  double j = 1.e-15;
  // print output
  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-15) is 0 because 1.0e-15 is non-zero!
  ```

10.4. && operator (logical AND)

- Evaluates a Boolean expression from left to right
- Its value is `TRUE` if and only if **both** sides of the operator are `TRUE`
- `[X]`

  Example: guess the outcome first

```
if ( 3 > 1 && 5 == 10 )
  printf("The expression is TRUE.\n");
 else
   printf("The expression is FALSE.\n");
```

```
The expression is FALSE.
```

- `[ ]`

  Example: guess the outcome first

```
if (3 < 5 && 5 == 5 )
  printf("The expression is TRUE.\n");
 else
   printf
     ("The expression is FALSE.\n");
```

```
The expression is TRUE.
```

10.5. || operator (logical OR)

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

  Example: guess the outcome first

```
if ( 3 > 5 || 5 == 5 )
  printf("The expression is TRUE.\n");
 else
   printf("The expression is FALSE.\n");
```

```
The expression is TRUE.
```

- `[X]`

  Example: guess the outcome first

```
if ( 3 > 5 || 6 < 5 )
  printf("The expression is TRUE.\n");
 else
   printf("The expression is FALSE.\n");
```

```
The expression is FALSE.
```

10.6. Checking for upper and lower case

- Characters are represented by ASCII[6] character sets

- E.g. `a` and `A` are represented by the ASCII codes 97 and 65, resp.

- Let's check that.

  ```
  echo "a A" > ./src/ascii
  cat ./src/ascii
  ```

  In 1, two characters are scanned and then printed as characters and as integers:

  ```
  char c1, c2;
  scanf("%c %c", &c1, &c2);
  printf("The ASCII value of %c is %d\n", c1, c1);
  printf("The ASCII value of %c is %d\n", c2, c2);
  ```

  ```
  The ASCII value of a is 97
  The ASCII value of A is 65
  ```

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

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

10.7. 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` with a number in it.

  ```
  echo 5 > ./src/num
  cat ./src/num
  ```

- `[ ]`

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

  ```
  int response = 0; // declare and initialize integer

  scanf("%d", &response);  // scan integer input

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

  ```
  Number in range.
  ```

- How can you translate a range like `![1,10]` into a conditional expression? It means that we want to test if a number is outside of the closed interval `[1,10]`.
- The numbers that fulfil this condition are smaller than 1 or greater than 10, hence the condition is `x < 1 || x > 10`.

- This is more conveniently written as `x < 1 || 10 < x`.

10.8. Let's practice

Open and complete the `operators.org` practice file.

11. NEXT The switch structure

11.1. Download the practice file

- Get the file with `wget`
- Check the file type with `file`
- Display the first 2 lines with `head`

```
wget tinyurl.com/2p95hxrp -O switch.org -o log
file switch.org
head -n 2 switch.org
```

11.2. Overview

- 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
  }
  ```

- **Controlling expression:** an integer expression in parentheses, like `(5)`. Characters are treated as integers in C and cannot be tested, so `('a')` is not allowed.
- **Case labels:** each case begins with a constant expression, like `Case 5:` - this is like any other expression except that it cannot contain variables or function calls.
- **Statements:** any number of statements. No braces required around the statements. The last statement is usually `break` to close the case.

11.3. 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 1 for different values:

VALUE	OUTPUT
5	Failing
3	Passing
0	Illegal grade
-1	Illegal grade

VALUE	OUTPUT
0.5	Illegal grade

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: Which problem/solution set does the 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
```

- Note: You cannot 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.

11.4. 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.

11.5. Let's practice!

Open and complete the `switch.org` practice file.

12. 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] Only from R version 4.1 - before that, you have to use the magrittr pipe operator `%>%`.

[2] 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.

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

[4] 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).

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

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

Author: Marcus Birkenkrahe
Created: 2022-06-13 Mon 16:55