Writing Program in C
Expressions and Control Structures (Selection Statements and Loops)

Jan Faigl

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Lecture 02

BE5B99CPL – C Programming Language

Overview of the Lecture

- Part 1 – Expressions
  - Operators – Arithmetic, Relational, Logical, Bitwise, and Other
  - Associativity and Precedence
  - Assignment *K. N. King: chapter 4 and 20*
- Part 2 – Control Structures: Selection Statements and Loops
  - Statements and Coding Styles
  - Selection Statements
  - Loops
  - Conditional Expression *K. N. King: chapters 5 and 6*

Part I

Part 1 – Expressions

Expressions

- **Expression** – prescribes calculation using *operands*, *operators*, and *brackets*
- Expression consists of
  - literals
  - variables
  - constants
  - unary and binary operators
  - function call
  - brackets
- The order of operation evaluation is prescribed by the operator **precedence** and **associativity**.

```
10 + x * y      // order of the evaluation 10 + (x * y)
10 + x + y      // order of the evaluation (10 + x) + y
```

** has higher priority than +*
+ is associative from the left-to-right

- A particular order of evaluation can be precisely prescribed by **fully parenthesized expression**

Simply: If you are not sure, use brackets.

Operators

- Operators are selected characters (or a sequences of characters) dedicated for writting expressions
- Five types of binary operators can be distinguished
  - **Arithmetic** operators – additive (addition/subtraction) and multiplicative (multiplication/division)
  - **Relational** operators – comparison of values (less than, ...)
  - **Logical** operators – logical `AND` and `OR`
  - **Bitwise** operators – bitwise `AND`, `OR`, `XOR`, bitwise shift (left, right)
  - Assignment operator `=` – a variable (l-value) is on its left side
- Unary operators
  - Indicating positive/negative value: + and −

    Operator − modifies the sign of the expression
  - Modifying a variable : ++ and −−
  - Logical negation: !
  - Bitwise negation: ~
- Ternary operator – conditional expression ? :

Reminder

Arithmetic Operators

- Operands of arithmetic operators can be of any arithmetic type

*The only exception is the operator for the integer reminder % defined for the **int** type*

*	Multiplication	`x * y`	Multiplication of x and y
/	Division	`x / y`	Division of x and y
%	Reminder	`x % y`	Reminder from the x / y
+	Addition	`x + y`	Sum of x and y
-	Subtraction	`x - y`	Subtraction x and y
+	Unary plus	`+x`	Value of x
-	Unary minus	`-x`	Value of −x
++	Increment	`++x`/`x++`	Incrementation before/after the evaluation of the expression x
--	Decrement	`--x`/`x--`	Decrementation before/after the evaluation of the expression x

Integer Division

- The results of the division of the operands of the **int** type is the integer part of the division

E.g.. 7/3 is 2 and −7/3 is −2
- For the integer reminder, it holds $x\%y = x - (x/y) * y$

E.g., 7 % 3 is 1   -7 % 3 is -1   7 % -3 is 1   -7 % -3 is -1
- C99: The result of the integer division of negative values is the value closer to 0.
  - It holds that (a/b)*b + a%b = a.

For older versions of C, the results depends on the compiler.

Implementation-Defined Behaviour

- The C standard deliberately leaves parts of the language unspecified
- Thus, some parts depend on the implementation, i.e., compiler, environment, computer architecture

E.g., Reminder behavior for negative values and version of the C prior C99.
- The reason for that is to the focus of C on efficiency, i.e., match the hardware behavior
- Having this in mind, it is best to rather avoid writing programs that depends on implementation-defined behavior.

K.N.King: Page 55

Unary Arithmetic Operators

- Unary operator (++ and −−) change the value of its operand

The operand must be the `l-value`, i.e., an expression that has memory space, where the value of the expression is stored, e.g., a variable.
  - It can be used as **prefix** operator, e.g., ++**x** and −−**x**
  - or as **postfix** operator, e.g., **x**++ and **x**−−
  - In each case, the final value of the expression is different!

int i; int a;	**value of i**	**value of a**
i = 1; a = 9;	**1**	**9**
a = i++;	**2**	**1**
a = ++i;	**3**	**3**
a = ++(i++);	**Not allowed!, value of i++ is not the l-value**	

Notice, for the unary operator `i++` it is necessary to store the previous value of `i` and then the variable `i` is incremented. The expression `++i` only increments the value of `i`. Therefore, `++i` can be more efficient.

Relational Operators

- Operands of relational operators can be of arithmetic type, pointers (of the same type) or one operand can be `NULL` or pointer of the `void` type

<	Less than	`x < y`	1 if x is less than y; otherwise 0
<=	Less than or equal	`x <= y`	1 if x is less then or equal to y; otherwise 0
>	Greater than	`x > y`	1 if x is greater than y; otherwise 0
>=	Greater than or equal	`x >= y`	1 if x is greater than or equal to y; otherwise 0
==	Equal	`x == y`	1 if x is equal to y; otherwise 0
!=	Not equal	`x != y`	1 if x is not equal to y; otherwise 0

Logical operators

- Operands can be of arithmetic type or pointers
- Resulting value `1` means `true`, `0` means `false`
- In the expressions `&&` (Logical AND) and `||` (Logical OR), the left operand is evaluated first
- If the results is defined by the left operand, the right operand is not evaluated

  Short-circuiting behavior – it may speed evaluation of complex expressions in runtime.

&&	Logical AND	`x && y`	1 if x and y is not 0; otherwise 0
||	Logical OR	`x || y`	1 if at least one of x, y is not 0; otherwise 0
!	Logical NOT	`!x`	1 if x is 0; otherwise 0

- Operands && a || have the short-circuiting behavior, i.e., the second operand is not evaluated if the result can be determined from the value of the first operand.

Bitwise Operators

- Bitwise operators treat operands as a series of bits

  Low-Level Programming – A programming language is low level when its programs require attention of the irrelevant. K.N.King: Chapter 20.

&	Bitwise AND	`x & y`	1 if x and y is equal to 1 (bit-by-bit)
|	Bitwise inclusive OR	`x | y`	1 if x or y is equal to 1 (bit-by-bit)
ˆ	Bitwise exclusive or (XOR)	`x ^ y`	1 if only x or only y is 1 (bit-by-bit)
∼	Bitwise complement (NOT)	`~x`	1 if x is 0 (bit-by-bit)
<<	Bitwise left shift	`x << y`	Shift of x about y bits to the left
>>	Bitwise right shift	`x >> y`	Shift of x about y bits to the right

Bitwise Shift Operators

- Bitwise shift operators shift the binary representation by a given number of bits to the left or right
  - Left shift – Each bit shifted off a zero bit enters at the right
  - Right shift – Each bit shift off
    - a zero bit enters at the left – for positive values or unsigned types
    - for negative values, the entered bit it can be either 0 (logical shift) or 1 (arithmetic shift right). Depends on the compiler.
- Bitwise shift operators have lower precedence than the arithmetic operators!
  - $i << 2 + 1$ means $i << (2 + 1)$

  Do not be surprise – parenthesized the expression!

Example – Bitwise Expressions

```
uint8_t a = 4;
uint8_t b = 5;

a     dec: 4 bin: 0100
b     dec: 5 bin: 0101
a & b dec: 4 bin: 0100
a | b dec: 5 bin: 0101
a ^ b dec: 1 bin: 0001

a >> 1 dec: 2 bin: 0010
a << 1 dec: 8 bin: 1000
```

lec02/bits.c

Operators for Accessing Memory

Here, for completeness, details in the further lectures.

- In C, we can directly access the memory address of the variable
- The access is realized through a pointer

  It allows great options, but it also needs responsibility.

Operator	Name	Example	Result
&	Address	`&x`	Pointer to x
*	Indirection	`*p`	Variable (or function) addressed by the pointer p
[]	Array subscripting	`x[i]`	`*(x+i)` – item of the array x at the position i
.	Structure/union member	`s.x`	Member x of the struct/union s
->	Structure/union member	`p->x`	Member x of the struct/union addressed by the pointer p

It is not allowed an operand of the & operator is a bit field or variable of the register class.

*Operator of the indirect address * allows to access to the memory using pointers.*

Other Operators

Operator	Name	Example	Result
()	Function call	`f(x)`	Call the function f with the argument x
(type)	Cast	`(int)x`	Change the type of x to int
sizeof	Size of the item	`sizeof(x)`	Size of x in bytes
? :	Conditional	`x ? y : z`	Do y if x != 0; otherwise z
,	Comma	`x, y`	Evaluate x and then y, the result is the result of the last expression

- The operand of `sizeof()` can be a type name or expression

```
int a = 10;
printf("%lu %lu\n", sizeof(a), sizeof(a + 1.0));
```

lec02/sizeof.c

- Example of the comma operator

```
for (c = 1, i = 0; i < 3; ++i, c += 2) {
   printf("i: %d c: %d\n", i, c);
}
```


Cast Operator

- Changing the variable type in runtime is called type case
- Explicit cast is written by the name of the type in `()`, e.g.,

```
int i;
float f = (float)i;
```

- Implicit cast is made automatically by the compiler during the program compilation
- If the new type can represent the original value, the value is preserved by the cast
- Operands of the `char`, `unsigned char`, `short`, `unsigned short`, and the bit field types can be used everywhere where it is allowed to use `int` or `unsigned int`.

  C expects at least values of the int type.

  - Operands are automatically cast to the `int` or `unsigned int`.

Operators Associativity and Precedence

- Binary operation op is associative on the set $\boldsymbol{S}$ if

  $(x \operatorname{op} y) \operatorname{op} z = x \operatorname{op}(y \operatorname{op} z)$, for each $x, y, z \in \boldsymbol{S}$
- For not associative operators, it is required to specify the order of evaluation
  - Left-associative – operations are grouped from the left

    E.g., $10 - 5 - 3$ *is evaluated as* $(10 - 5) - 3$
  - Right-associative – operations are grouped from the right

    E.g. $3 + 5^2$ *is 28 or* $3 \cdot 5^2$ *is 75 vs* $(3 \cdot 5)^2$ *is 225*
- The assignment is left-associative

  E.g., y=y+8

  First, the whole right side of the operator = is evaluated and then, the results is assigned to the variable on the left.
- The order of the operator evaluation can be defined by the fully parenthesized expression.

Summary of the Operators and Precedence 1/3

Precedence	Operator	Associativity	Name
1	++	L→R	Increment (postfix)
	−−		Decrementation (postfix)
	()		*Function call*
	[]		*Array subscripting*
	. −>		*Structure/union member*
2	++	R→L	Increment (prefix)
	−−		Decrementation (prefix)
	!		Logical negation
	~		Bitwise negation
	− +		Unary plus/minus
	*		*Indirection*
	&		*Address*
	sizeof		*Size*

Summary of the Operators and Precedence 2/3

Precedence	Operator	Associativity	Name
3	()	R→L	*Cast*
4	*, /, %	L→R	Multiplicative
5	+ -		Additive
6	>>, <<		Bitwise shift
7	<, >, <=, >=		Relational
8	==, !=		Equality
9	&		Bitwise AND
10	^		Bitwise exclusive OR (XOR)
11	^		Bitwise inclusive OR (OR)
12	&&		Logical AND
13	||		Logical OR

Summary of the Operators and Precedence 3/3

Precedence	Operator	Associativity	Name
14	? :	R→L	Conditional
15	=		Assignment
	+ =, − =		additive
	∗ =, / =, % =	R→L	multiplicative
	<<=, >>=		bitwise shift
	& =, ^=, | =		Bitwise AND, XOR, OR
15	,	L→R	Comma

K. N. King: Page 735
http://en.cppreference.com/w/c/language/operator_precedence

Simple Assignment

- Set the value to the variable
  Store the value into the memory space referenced by the variable name.
- The form of the assignment operator is
  ⟨**variable**⟩ = ⟨**expression**⟩
  Expression is literal, variable, function call, ...
- C is statically typed programming language
  - A value of an expression can be assigned only to a variable of the same type
    Otherwise type case it necessary
  - Example of implicit type case

```
int i = 320.4; // implicit conversion from 'double' to 'int'
   changes value from 320.4 to 320 [-Wliteral-conversion]

char c = i;    // implicit truncation 320 -> 64
```

- C is type safe only within a limited context of the compilation, e.g., for `printf("%d\n", 10.1);` A compiler reports an error
- In general, C is not type safe
  In runtime, it is possible to write out of the allocated memory space.

Compound Assignment

- A short version of the assignment to compute a new value of the variable from itself:
  ⟨**variable**⟩ = ⟨**variable**⟩ ⟨**operator**⟩ ⟨**expression**⟩
- can be written as
  ⟨**variable**⟩ ⟨**operator**⟩ = ⟨**expression**⟩

Example

```
int i = 10;
double j = 12.6;

i = i + 1;
j = j / 0.2;
```

```
int i = 10;
double j = 12.6;

i += 1;
j /= 0.2;
```

- Notice, assignment is an expression
  The assignment of the value to the variable is a side effect

```
int x, y;

x = 6;
y = x = x + 6;
```


Assignment Expression and Assignment Statement

- The statement performs some action and it is terminated by ;

```
robot_heading = -10.23;
robot_heading = fabs(robot_heading);
printf("Robot heading: %f\n", robot_heading);
```

- Expression has **type and value**
  23 int type, value is 23
  14+16/2 int type, value is 22
  y=8 int type, value is 8
- Assignment is expression and its value is assigned to the left side
- The assignment expression becomes the assignment statement by adding the semicolon

Undefined Behaviour

- There are some statements that can cause **undefined behavior** according to the C standard.
  - `c = (b = a + 2) - (a - 1);`
  - `j = i * i++;`
- The program may behaves differently according to the used compiler, but also may not compile or may not run; or it may even crash and behave erratically, produce meaningless results.
- It may also happened if variables are used without initialization
- Avoid statements that may produce undefined behavior!

Example of Undefined Behaviour

- Standard C does not defined the behaviour for the overflow of the integer value (`signed`)
  - E.g., For the complement representation the expression can be `127 + 1` of the `char` equal to `-128` (see `lec02/demo-loop_byte.c`)
  - Representation of integer values may depend on the architecture and can be different, e.g., binary or inverse codes
- Implementation of the defined behaviour can be computationally expensive, and thus the behaviour is not defined by the standard
- Behaviour is not defined and depends on the compiler, e.g. `clang` and `gcc` without/with the optimization `-O2`
  -
    ```
    for (int i = 2147483640; i >= 0; ++i) {
       printf("%i %x\n", i, i);
    }
    ```
    lec02/int_overflow-1.c
    Without the optimization, the program prints 8 lines, for `-O2`, the program compiled by `clang` prints 9 lines and `gcc` produces infinite loop.
  -
    ```
    for (int i = 2147483640; i >= 0; i += 4) {
       printf("%i %x\n", i, i);
    }
    ```
    lec02/int_overflow-2.c
    A program compiled by `gcc` with `-O2` is crashing

Take a look to the asm code using the compiler parameter -S

Part II

Part 2 – Control Structures: Selection Statements and Loops

Statement and Compound Statement (Block)

- Statement is terminated by `;`
  Statement consisting only of the semicolon is empty statement.
- Block consists of sequences of declarations and statements
- **ANSI C, C89, C90**: Declarations must be placed prior other statements
  It is not necessary for C99
- Start and end of the block is marked by the `{` and `}`
- A block can be inside other block

```
void function(void)
{ /* function block start */
   {/* inner block */
      for (i = 0; i < 10; ++i)
      {
      //inner for-loop block
      }
   }
}
```

```
void function(void) { /* function
    block start */
   { /* inner block */
      for (int i = 0; i < 10; ++i) {
      //inner for-loop block
      }
   }
```

Notice the coding styles.

Coding Style

- It supports clarity and readability of the source code
  https://www.gnu.org/prep/standards/html_node/Writing-C.html
- Formatting of the code is the fundamental step
  Setup automatic formatting in your text editor
- Appropriate identifiers
- Train yourself in coding style even at the cost of slower coding. Readability and clarity is important, especially during debugging.
  Notice, sometimes it can be better to start from scratch
- Recommend coding style (CPL)

```
void function(void)
{ /* function block start */
   for (int i = 0; i < 10; ++i) {
      //inner for-loop block
      if (i == 5) {
         break;
      }
   }
}
```

- Use English, especially for identifiers
- Use nouns for variables
- Use verbs for function names

Lecturer's preference: indent shift 3, space characters rather than tabular.

Coding Styles – Links

- There are many different coding styles
- Inspire yourself by existing recommendations
- Inspire yourself by reading representative source codes

http://users.ece.cmu.edu/~eno/coding/CCodingStandard.html
https://www.doc.ic.ac.uk/lab/cplus/cstyle.html
http://en.wikipedia.org/wiki/Indent_style
https://google.github.io/styleguide/cppguide.html
https://www.kernel.org/doc/Documentation/CodingStyle
https://google.github.io/styleguide/cppguide.html

Control Statements

- Selection Statement
  - Selection Statement: `if ()` or `if () ... else`
  - Switch Statement: `switch () case ...`
- Control Loops
  - `for ()`
  - `while ()`
  - `do ... while ()`
- Jump statements (unconditional program branching)
  - `continue`
  - `break`
  - `return`
  - `goto`

Selection Statement – `if`

- `if (expression) statement`$_1$`; else statement`$_2$
- For `expression != 0` the `statement`$_1$ is executed; otherwise `statement`$_2$
  The statement can be the compound statement
- The `else` part is optional
- Selection statements can be nested and cascaded

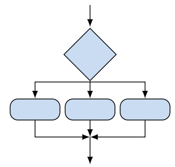
```
int max;
if (a > b) {
   if (a > c) {
      max = a;
   }
}
```

```
int max;
if (a > b) {
   ...
} else if (a < c) {
   ...
} else if (a == b) {
   ...
} else {
   ...
}
```


The `switch` Statement

- Allows to branch the program based on the value of the expression of the enumerate (integer) type, e.g., `int`, `char`, `short`, `enum`
- The form is

  switch (expression) {
     case constant$_1$: statements$_1$; **break;**
     case constant$_2$: statements$_2$; **break;**
     ...
     case constant$_n$: statements$_n$; **break;**
     default: statements$_{def}$; **break;** }



  where *constants* are of the same type as the *expression* and *statements$_i$* is a list of statements
- Switch statements can be nested

Semantics: First the expression value is calculated. Then, the statements under the same value are executed. If none of the branch is selected, statements$_{def}$ under `default` branch as performed (optional)

The `switch` Statement – Example

```
switch (v) {
   case 'A':
      printf("Upper 'A'\n");
      break;
   case 'a':
      printf("Lower 'a'\n");
      break;
   default:
      printf(
      "It is not 'A' nor 'a'\n");
      break;
}
```

```
if (v == 'A') {
   printf("Upper 'A'\n");
} else if (v == 'a') {
   printf("Lower 'a'\n");
} else {
   printf(
   "It is not 'A' nor 'a'\n");
}
```

lec02/switch.c

The Role of the `break` Statement

- The statement `break` terminates the branch. If not presented, the execution continues with the statement of the next `case` label

Example

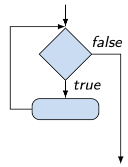
```
int part = ?
switch(part) {
   case 1:
      printf("Branch 1\n");
      break;
   case 2:
      printf("Branch 2\n");
   case 3:
      printf("Branch 3\n");
      break;
   case 4:
      printf("Branch 4\n");
      break;
   default:
      printf("Default branch\n");
      break;
}
```

- part ← 1
  Branch 1
- part ← 2
  Branch 2
  Branch 3
- part ← 3
  Branch 3
- part ← 4
  Branch 4
- part ← 5
  Default branch

lec02/demo-switch_break.c

Loops

- Loop statements **for** and **while** test the controlling expression before the enter to the loop body



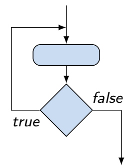
  - **for** – initialization, condition, change of the controlling variable can be a part of the syntax

```
for (int i = 0; i < 5; ++i) {
   ...
}
```

  - **while** – controlling variable out of the syntax

```
int i = 0;
while (i < 5) {
   ...
   i += 1;
}
```

- The **do** loop tests the controlling expression after the first loop




```
int i = -1;
do {
   ...
   i += 1;
} while (i < 5);
```

The for Loop

- The basic form is: `for (expr₁; expr₂; expr₃) statement`
- All $expr_i$ are expression and typically they are used for
  1. $expr_1$ – initialization of the controlling variable (side effect of the assignment expression)
  2. $expr_2$ – Test of the controlling expression
  3. If $expr_2$ `!=0` the `statement` is executed; Otherwise the loop is terminated
  4. $expr_3$ – updated of the controlling variable (performed at the end of the loop
- Any of the expressions $expr_i$ can be omitted
- **break** statement – force termination of the loop
- **continue** – force end of the current iteration of the loop

  The expression $expr_3$ is evaluated and test of the loop is performed.
- An infinity loop can be written by omitting the expressions

```
for (;;) {...}
```


The continue Statement

- It transfers the control to the evaluation of the controlling expressions of the loops
- The `continue` statement can be used inside the body of the loops
  - `for ()`
  - `while ()`
  - `do...while ()`
- Examples

```
int i;
for (i = 0; i < 20; ++i) {
   if (i % 2 == 0) {
      continue;
   }
   printf("%d\n", i);
}
```

lec02/continue.c

```
for (int i = 0; i < 10; ++i) {
   printf("i: %i ", i);
   if (i % 3 != 0) {
      continue;
   }
   printf("\n");
}
```

lec02/demo-continue.c

```
clang demo-continue.c
./a.out
i:0
i:1 i:2 i:3
i:4 i:5 i:6
i:7 i:8 i:9
```


The break Statement – Force Termination of the Loop

- The program continue with the next statement after the loop
- Example in the `while` loop

```
int i = 10;
while (i > 0) {
   if (i == 5) {
      printf("i reaches 5, leave the loop\n");
      break;
   }
   i--;
   printf("End of the while loop i: %d\n", i);
}
```

lec02/break.c

- Example in the `for` loop

```
for (int i = 0; i < 10; ++i) {
   printf("i: %i ", i);
   if (i % 3 != 0) {
      continue;
   }
   printf("\n");
   if (i > 5) {
      break;
   }
}
```

```
clang demo-break.c
./a.out
i:0
i:1 i:2 i:3
i:4 i:5 i:6
```

lec02/demo-break.c

The goto Statement

- Allows to transfers the control to the defined label

  It can be used only within a function body
- Syntax `goto label;`
- The jump `goto` can jump only outside of the particular block
- Can be used only within a function block

```
int test = 3;
for (int i = 0; i < 3; ++i) {
   for (int j = 0; j < 5; ++j) {
      if (j == test) {
         goto loop_out;
      }
      fprintf(stdout, "Loop i: %d j: %d\n", i, j);
   }
}
return 0;
loop_out:
fprintf(stdout, "After loop\n");
return -1;
```

lec02/goto.c

Nested Loops

- The **break** statement terminates the inner loop

```
for (int i = 0; i < 3; ++i) {
   for (int j = 0; j < 3; ++j) {
      printf("i-j: %i-%i\n", i, j);
      if (j == 1) {
         break;
      }
   }
}
```

```
i-j: 0-0
i-j: 0-1
i-j: 1-0
i-j: 1-1
i-j: 2-0
i-j: 2-1
```

- The outer loop can be terminated by the `goto` statement

```
for (int i = 0; i < 5; ++i) {
   for (int j = 0; j < 3; ++i) {
      printf("i-j: %i-%i\n", i, j);
      if (j == 2) {
         goto outer;
      }
   }
}
outer:
```

```
i-j: 0-0
i-j: 0-1
i-j: 0-2
```

lec02/demo-goto.c

Example – isPrimeNumber() 1/2

```
#include <stdbool.h>
#include <math.h>

_Bool isPrimeNumber(int n)
{
   _Bool ret = true;
   for (int i = 2; i <= (int)sqrt((double)n); ++i) {
      if (n % i == 0) {
         ret = false;
         break;
      }
   }
   return ret;
}
```

lec02/demo-prime.c

- Once the first factor is found, call `break` to terminate the loop

  It is not necessary to test other numbers

Example – isPrimeNumber() 2/2

- The value of `(int)sqrt((double)n)` is not changing in the loop

```
for (int i = 2; i <= (int)sqrt((double)n); ++i) {
   ...
}
```

- We can use the comma operator to initialize the `maxBound` variable

```
for (int i = 2, maxBound = (int)sqrt((double)n);
     i <= maxBound; ++i) {
   ...
```

- Or, we can declare `maxBound` as constant

```
_Bool ret = true;
const int maxBound = (int)sqrt((double)n);
for (int i = 2; i <= maxBound ; ++i) {
   ...
}
```

E.g., Compile and run `demo-prime.c`: `clang demo-prime.c -lm; ./a.out 13`

Conditional Expression – Example Greatest Common Divisor

```
int getGreatestCommonDivisor(int x, int y)
{
   int d;
   if (x < y) {
      d = x;
   } else {
      d = y;
   }
   while ( (x % d != 0) || (y % d ! = 0)) {
      d = d - 1;
   }
   return d;
}
```

- The same with the conditional expression: $expr_1$? $expr_2$: $expr_3$

```
int getGreatestCommonDivisor(int x, int y)
{
   int d = x < y ? x : y;
   while ( (x % d != 0) || (y % d ! = 0)) {
      d = d - 1;
   }
   return d;
}
```

lec02/demo-gcd.c

Summary of the Lecture

Topics Discussed

- Expressions
  - Operators – Arithmetic, Relational, Logical, Bitwise, and others
  - Operator Associativity and Precedence
  - Assignment and Compound Assignment
  - Implementation-Defined Behaviour
  - Undefined Behaviour
- Coding Styles
- Select Statements
- Loops
- Conditional Expression

- Next: Data types, memory storage classes, function call