

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

Part I

Part 1 – Expressions

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*

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 writing 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

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 remainder, 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.

Arithmetic Operators

- Operands of arithmetic operators can be of any arithmetic type

*The only exception is the operator for the integer remainder % 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 |

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
<code>i = 1; a = 9;</code>	1	9
<code>a = i++;</code>	2	1
<code>a = ++i;</code>	3	3
<code>a = ++(i++);</code>	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.

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.

<code>&&</code>	Logical AND	<code>x && y</code>	1 if x and y is not 0; otherwise 0
<code> </code>	Logical OR	<code>x y</code>	1 if at least one of x, y is not 0; otherwise 0
<code>!</code>	Logical NOT	<code>!x</code>	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.

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

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

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.

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

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

*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.*

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`

Other Operators

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

Summary of the Operators and Precedence 1/3

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

Operators Associativity and Precedence

- Binary operation `op` is **associative** on the set **S** if

$$(x \text{ op } y) \text{ op } z = x \text{ op } (y \text{ op } z), \text{ for each } x, y, z \in \mathbf{S}$$
- 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 2/3

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

Compound Assignment

- A short version of the assignment to compute a new value of the variable from itself:

$$\langle \text{variable} \rangle = \langle \text{variable} \rangle \langle \text{operator} \rangle \langle \text{expression} \rangle$$

- can be written as

$$\langle \text{variable} \rangle \langle \text{operator} \rangle = \langle \text{expression} \rangle$$

Example

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

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

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
$$\langle \text{variable} \rangle = \langle \text{expression} \rangle$$

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.

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 behave 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 happen if variables are used without initialization
- Avoid statements that may produce undefined behavior!**

Part II

Part 2 – Control Structures: Selection Statements and Loops

Example of Undefined Behaviour

- Standard C does not define 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

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)

```

1 void function(void)
2 { /* function block start */
3   for (int i = 0; i < 10; ++i) {
4     //inner for-loop block
5     if (i == 5) {
6       break;
7     }
8   }
9 }
```

- 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.

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`

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>

Selection Statement – if

- `if (expression) statement1; else statement2`
- For `expression != 0` the `statement1` is executed; otherwise `statement2`
The statement can be the compound statement
- The `else` part is optional
- Selection statements can be nested and cascaded

```

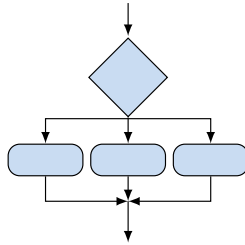
int max;
if (a > b) {
    if (a > c) {
        max = a;
    }
}

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 constant1: statements1; break;
  case constant2: statements2; break;
  ...
  case constantn: statementsn; break;
  default: statementsdef; 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 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

```
1 int part = ?
2 switch(part) {
3   case 1:
4     printf("Branch 1\n");
5     break;
6   case 2:
7     printf("Branch 2\n");
8   case 3:
9     printf("Branch 3\n");
10    break;
11   case 4:
12    printf("Branch 4\n");
13    break;
14   default:
15    printf("Default branch\n");
16    break;
17 }
```

- 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

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

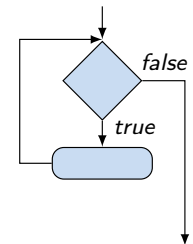
Loops

- 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

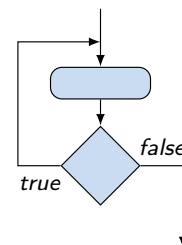

```
for (int i = 0; i < 5; ++i) {
  ...
}
```
- `while` – controlling variable out of the syntax


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

The expression `expr3` is evaluated and test of the loop is performed.
- An infinity loop can be written by omitting the expressions


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

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 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);
}
```

```
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
```

lec02/continue.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

```
1 int test = 3;
2 for (int i = 0; i < 3; ++i) {
3     for (int j = 0; j < 5; ++j) {
4         if (j == test) {
5             goto loop_out;
6         }
7         fprintf(stdout, "Loop i: %d j: %d\n", i, j);
8     }
9 }
10 return 0;
11 loop_out:
12 fprintf(stdout, "After loop\n");
13 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; ++j) {
        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() 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`

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

Conditional Expression – Example Greatest Common Divisor

```
1 int getGreatestCommonDivisor(int x, int y)
2 {
3     int d;
4     if (x < y) {
5         d = x;
6     } else {
7         d = y;
8     }
9     while ( (x % d != 0) || (y % d != 0) ) {
10        d = d - 1;
11    }
12    return d;
13 }
```

- The same with the conditional expression: **expr₁ ? expr₂ : expr₃**

```
1 int getGreatestCommonDivisor(int x, int y)
2 {
3     int d = x < y ? x : y;
4     while ( (x % d != 0) || (y % d != 0) ) {
5         d = d - 1;
6     }
7     return d;
8 }
```

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