C, Basics, Types, Literals, Expressions, Statements

Tomasz Żok

Literature

- “C in a nutshell,” Peter Prinz, Tony Crawford
- “Computer architecture,” John Hennessy, David Patterson
- “Zrozumieć programowanie,” Gynvael Coldwind

Language basics

- C is a general-purpose, procedural programming language devised in 1970s by Dennis Ritchie
- Design ideas:
  - Source code portability
  - Ability to operate “close to hardware”
  - Efficiency
- C language is relatively compact and easy to port to new systems
- Its standard library is written itself in portable C

Structure of C programs

- A basic building blocks are **functions** which can invoke one another
- Functions contains **statements** to be executed sequentially
- Statements can be grouped into **block statements** or just **blocks**
- Every C program must define at least one function with the special name `main()` – this is the entrypoint when the program starts
- In C each function must be *declared* before its usage. If the function *definition* is located later in the code, you need to provide a function **prototype**
- Functions definitions cannot be nested

Source files

- C source files usually follow such internal structure:
  - Preprocessor directives
  - Global declarations
  - Function definitions
- You can write whole program in a single C source file, but the language allows to organize the program in a modular way
- When code is divided into several files, you need to declare the same functions and global variables. These declarations are typically located in header files
- Usually, C source files have suffix `.c` and header files have suffix `.h`

Source files

- The compiler treats each C source file together with its included headers as a single **translation unit**
- The translation unit is parsed into **tokens** i.e. smallest semantic units such as variable names or operators
- C language allows to use all kinds of whitespaces and breaks between tokens to format a *human-readable* code as you wish
- One exception: preprocessor directives require to be the only statement in the line
- A most-general style description: use one declaration or statement per line and use whitespaces to reflect nested blocks of code

Comments

- Two ways to define comments in C:

```
/*
 Block comment
 */

// line comment
```

- The preprocessor replaces each comment with a whitespace, so the following is possible:

```
int c = /* first */ 7 + /* second */ 19;
```

- Comments' tokens (`//` and `/* */`) embedded in a string literal are not treated as comments:

```
printf("// This is not a comment");
```


Character sets

- C supports the basic character set consisting of:
  - Letters of Latin alphabet
  - Decimal Digits
  - The following graphic characters: `! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ _ { | } ~`
  - Five whitespace characters: space, horizontal tab, vertical tab, newline, form feed
  - Four non-printable characters: *null* character (`\0`), alert (`\a`), backspace (`\b`) and carriage return (`\r`)
- Extended character sets are available through either *wide characters* and *multibyte characters*

Identifiers

- **Identifier** refers to the names of variables, functions, macros, structures and other objects in C program
- Identifiers can contain the following characters:
  - Letters from a basic character set `a-z` and `A-Z`
  - Underscore character _
  - Decimal digits `0-9` (but the first character of an identifier must not be a digit)
- There are 44 reserved words, which cannot be used as identifiers e.g. `while`
- Many identifiers are already taken by the standard library e.g. `printf`

```
// valid identifiers
int x, _1toN, Before;
// invalid identifiers
int x-ray, 100errors, break;
```


Identifier namespaces

- All identifiers belong to exactly one category:
  - Labels
  - Tags (names of structures, unions or enumeration types)
  - Names of structure/union members
  - Other also called *ordinary identifiers*
- Identifiers from different categories can be the same without conflict, but it is advised to not overuse this mechanism

```
struct x { int x; };
void f(struct x *x) {
    x->x = 0;
}
```


Identifier scopes

- Each identifier belongs to a scope, which determines that part of translation unit in which the identifier is meaningful:
  - **File scope**: defined outside all blocks and parameter lists; the identifier is available everywhere below its declaration
  - **Block scope**: defined in a block; the identifier is valid until the end of the smallest block containing its declaration
  - **Function prototype scope**: is valid only in the prototype; can be omitted
  - **Function scope**: the identifier is valid in the whole function block; this concerns labels which allow to use `goto` statement to jump to any block within function

Identifier scopes

```
int x;                  // file scope

int f(int a, int b);    // function prototype scope

int g() {
    int t = 0;          // function scope
    while (t < 10) {
        int s = t / 2;  // block scope
        t++;
    }
                        // variable `s` is not visible here
}
```

Identifier scopes

- Generally the scope begins after identifier declaration
- Exception: definitions of structure, union and enumeration types can reference itself
- It is possible to declare identifier in the same namespace if it is embedded in a block or function. Then the *inner* declaration hides the *outer* one

```
int x = 1;          // x is an integer
while (condition) {
    float x = 1.0f; // x is a real number
}
x++;                // x is again an integer
```


How the C compiler works

- Each C source file together with its included headers is a single **translation unit**
- C compiler translates it into machine code and writes it into an **object file** (often with suffix `.o` or `.obj`):
  1. Characters are read and converted if necessary
  2. A combination of backslash and newline is deleted (it allows to write macros in multiple lines)
  3. The source is parsed into tokens
  4. Preprocessor directives and macros are expanded
  5. Escape sequences are converted
  6. Adjacent string literals are concatenated
  7. The actual compiling
  8. Linker generates executable file
- Linker combines object files and library functions as an **executable file**
- The executable file has information needed to load and start it

Tokens

- A token is a keyword, identifier, constant, string literal or a symbol (operator, semicolon, parenthesis, brace)
- When generating tokens, the compiler follows a principle: each successive non-whitespace character must be appended to the token unless it makes it invalid

```
printf("Hello world\n");

printf
(
"Hello world\n"
)
;
```

```
a+++b

a
++
+
b
```


Types

- In C the term **object** refers to a location in memory whose content can represent values
- Named object == variable
- Objects have **types** to determine the size of the object in memory and how to interpret the bit pattern e.g. *signed* vs *unsigned*
- Types in C:
  - Basic (integer and real numbers)
  - Enumerated
  - The `void` type
  - Derived (pointers, arrays, structures)
  - Unions
  - Functions

Integer types

Type	Synonyms
`signed char`	
`int`	`signed, signed int`
`short`	`short int, signed short, signed short int`
`long`	`long int, signed long, signed long int`
`long long`	`long long int, signed long long, signed long long int`
`_Bool`	`bool`
`unsigned char`	
`unsigned int`	`unsigned`
`unsigned short`	`unsigned short int`
`unsigned long`	`unsigned long int`
`unsigned long long`	`unsigned long long int`

Character type

- Type `char` can be signed or unsigned depending on the compiler
- This type can be used in arithmetic expressions or as a character code

```
char c = 'A';
printf("c = %c which has code %d\n", c, c);
c++;
printf("next is %c which has code %d\n", c, c);
```

```
c = A which has code 65
next is B which has code 66
```

Integer types

- C standard defines that:
  - `char` occupies exactly 1 byte
  - `short` occupies **at least** 2 bytes
  - `long` occupies **at least** 4 bytes
  - `long long` occupies **at least** 8 bytes
- The actual sizes of `short`, `long` and `long long` may be larger than the minimum required, but the following must be true: `sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)`
- The size of `int` is best adapted to the target system architecture

Signed integers

The most common representation of signed integers is **two's complement**:

Binary	Unsigned	Signed
00000000	0	0
00000001	1	1
...	...	...
01111111	127	127
10000000	128	-128
10000001	129	-127
...	...	...
11111110	254	-2
11111111	255	-1

Limits

- Header file `limits.h` contains several constants and macros to determine variables' limits in runtime

```
#include <limits.h>
#include <stdio.h>

int main() {
    printf("'int' limits [%d, %d]\n", INT_MIN, INT_MAX);
    printf("on this system 'char' is %s\n", CHAR_MIN == 0 ?
           "unsigned" : "signed");
    return 0;
}
```

```
'int' limits [-2147483648, 2147483647]
on this system 'char' is signed
```


Overflow

- An overflow occurs when the result of arithmetic operation exceeds the maximum or minimum values of the variable's type
- With unsigned integers, the overflow is ignored which essentially means that the value stored in a variable is equal to the remainder of a division by `UTYPE_MAX + 1`:

```
unsigned int i = UINT_MAX;     // i == UINT_MAX
i++;                           // i == 0
```

- With signed integers, overflow is an **undefined behaviour**. Some compilers may ignore it, other might abort the program

Standard integer types

- Several integer types are defined by the standard library:
  - `wchar_t`, `char16_t`, `char32_t`
  - `size_t`, `ptrdiff_t`
- There are also integer types with exact width:

Type	Meaning	Implementation
`intN_t`, `uintN_t`	Exactly *N* bits	Optional
`int_leastN_t`, `uint_leastN_t`	At least *N* bits	Required for *N* = 8, 16, 32, 64
`int_fastN_t`, `uint_fastN_t`	The fastest type with at least *N* bits	Required for *N* = 8, 16, 32, 64
`intmax_t`, `uintmax_t`	The widest integer type	Required

Floating-point types

- C defines a few types to represent non-integer numbers:
  - `float`: single precision
  - `double`: double precision
  - `long double`: extended precision
- The most common standard IEC 605559 (IEEE 754) defines:

Type	Size	Range	Smallest positive value	Precision
`float`	4 bytes	±3.4e+38	1.2e-38	6 digits
`double`	8 bytes	±1.7e+308	2.3e-308	15 digits
`long double`	10 bytes	±1.1e+4932	3.4e-4932	19 digits

Complex numbers

- C language defines accompanying types: *float _Complex*, *double _Complex* and *long double _Complex*
- In *complex.h* header, there are additional definitions of types (*float complex*, etc.) and macros (e.g. *CMPLX(real, imag)*)

Enumerated types

- The definition follows this general scheme:
  enum **identifier** *{ enumerator-list }*
- The enumeration constants represent integer values
- By default, the first name in the list gets value 0 if not assigned explicitly and all other get the preceding value increased by 1 if not assigned explicitly
- When using explicit assignment, some names can have the same value

Enumerated types

```
#include <stdio.h>

enum color { red, green=4, blue, black=0 };

int main() {
    enum color c = blue;
    printf("blue times 2 = %d\n", c * 2);
    printf("black = %d\n", black);
    return 0;
}
```

Result:

```
blue times 2 = 10
black = 0
```


The void type

- *void* means that there is no value
- There are three usage scenarios with *void*:
  - Function return types:
    `void subroutine() { ... }`
  - Expressions with no value:
    `(void)printf("No need for return value!\n");`
  - "Multipurpose" pointers to any type:
    `void *malloc(size_t size);`

The void type

```
void f(void) {
    return;
}

void g() {
    return;
}

int main() {
    f(1, 2, 3);
    g(1, 2, 3);
    return 0;
}
```


Literals

- Literals are tokens that represent fixed values: integers, floating-point nubmers, a character or a string
- Integer literals can be expressed as decimal, octal or hexadecimal number depending on the prefix
- Decimal literals start with a non-zero digit e.g. *123*, *820*
- Octal literals start with a digit zero e.g. *017* equals $1 \times 8 + 7 = 15$ in decimal notation
- Hexadecimal literals start with either *0x* or *0X* and can use letters from *A* to *F* in upper- or lowercase e.g. *0xAA*, *0Xaa*
- The type of literal is either *int*, *long* or *long long* depending on the value, but you can also force a different type by using a suffix *L*, *LL*, *U*, *UL* or *ULL*

Floating-point constants

- Floating-point numbers can be expressed in decimal or hexadecimal notation
- A decimal notation consists of digits with decimal point with the possibility of using scientific notation of *e* or *E* followed by a number treated as exponent of the power of 10:

Constant	Value
1.0	1
1.23e5	1.23×10^5
6e-17	6×10^{-7}

- The default type is `double`, but you can use suffix *F* (float) or *L* (long double) to change this

Floating-point constants

- Hexadecimal floating-point constants begin with either `0x` or `0X`, a hexadecimal number, letter *p* or *P* and a **decimal** number treated as an exponent of the power of 2
- You must always use *p* or *P* and an exponent even if it is zero
- Otherwise the last letter *F* would be ambiguous – *F* as in float type of literal or *F* as a hexadecimal digit equal to 15

Constant	Value
0x1p-2	$1 \times 2^{-2} = \frac{1}{4}$
0x1Fp1F	$(1 \times 16 + 15) \times 2^1 = 62$ (float type)

Character constants

- A character constant is one or more characters enclosed in single quotation marks:

```
'a'   'XY'   '*'
```

- Character constants have the type `int` equal to the encoded value of characters (warning: multi-character constants can be stored differently by compilers)
- You can also prefix the literal with *L*, *u* or *U* to enforce its type to be `wchar_t`, `char16_t` or `char32_t`

Escape sequences

Escape	Value
`\'` `\"`	Single or double quotation mark
`\?`	Question mark
`\\`	Backslash
`\a`	Alert
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t` `\v`	Horizontal or vertical tab
`\o` (octal digits)	Character with given octal code
`\xh` (hexadecimal digits)	Character with given hexadecimal code
`\uh` `\Uh`	Character with given universal character name

String literals

- String literal consists of a sequence of characters in double quotation marks
- Strings are arrays of characters terminated by the null character (i.e. `""` occupies one byte in memory)
- You can initialize an array with string literal or assign address of the first character to a pointer:

```
char s[100] = "Array initialization";
char *p = "Pointer initialization";
```


String literals

- Adjacent string literals (separated only by whitespace) are concatenated by the compiler:

```
char *p = "This "
          "is" " a "
          "test";
```

- Alternatively you can use backslash followed by newline: (but beware of whitespace)

```
char *p = "This \
           is a \
           test";
```

String literals

- String literals content should not be modified unless when array is initialized:

```
char s[10] = "Test";
*s = 'a';              // OK

char *p = "Test";
*p = 'a';              // ERROR!
```


Type conversions

- In C you can combine expressions of values with different types
- Compiler performs implicit conversions where possible, otherwise it raises a compilation error
- An **explicit cast operator** has the following form:

```
(type_name) expression
```

- For example:

```
int x = 10, y = 3;
float z1 = x / y;          // 3.000000
float z2 = (float)x / y;   // 3.333333
```


Type conversions

- C automatically converts values of arithmetic types to the wider of two types (e.g. from `float` to `double`)
- To compute expressions values, C performs integer promotion to wider type and then the result is written according to the original type:

```
char c = '?';
short var = 100;
if (c < 'A') {      // 'A' is an int, so c gets promoted
                    // to int
    var = var + 1;  // 1 is an int, so var is promoted
                    // to int, then addition is computed
                    // and the result is fitted to short
                    // variable
}
```


Type conversions

- Type conversions are also performed for function call arguments and with `return` statement to match function's return type
- Any scalar value is converted to `_Bool` with the rule that value zero equals `false` and any non-zero value equals `true`
- Conversion from real to integer number is performed by discarding the fractional part
- When converting to a type that may not exactly store all values (e.g. from `long` to `float`) the actual value is the next available:

```
long l = 123456789L;
float f = l;
printf("%lu %f %f\n", l, f, (double)f - l);
  // 123456789 123456792.000000 3.000000
```


Type conversions

- You can explicitly convert pointers to different types:

```
float f = 1.0;
float *pf = &f;
uint32_t *pi = (uint32_t *) &f;
printf("float=%f uint32=0x%X\n", *pf, *pi);
  // float=1.000000 uint32=0x3F800000
```

- You can convert anything to a `void *` and vice versa:

```
void qsort(void *array, size_t n, size_t element_size,
          int (*compare)(const void *, const void *));
```


Expressions

- Expression consists of a sequence of constants, identifiers and operators
- Expression's value implies the type of the expression
- Expressions can consist of other expressions:

```
s * f(x, y/2);
```

Lvalues and rvalues

- `lvalue` (from *left* or rather *location*) is a term to describe any object which can be assigned to i.e. a variable name, an array element
- `rvalue` (from *right*) is what may appear on the right side of the assignment operator e.g. constants or expressions

Expression	Lvalue?
`array[1]`	Yes
`&array[1]`	No
`ptr`	Yes
`*ptr`	Yes
`ptr+1`	No
`*ptr+1`	No

Lvalues and rvalues

- When an object has `constant` qualifier, it cannot be used as `lvalue`:

```
const int a = 10;
a = 20;                // ERROR!

int b = 20;
const int *pb = &b;
b = 30;                // OK
*pb = 40;              // ERROR!
```


Side effects

- Expression evaluation not only produces a result, but may also have side effects

```
i = i++;               // undefined behaviour
...
array[i] = array[i++]; // undefined behaviour
...
f(&i) + g(&i);         // undefined behaviour
```


Precedence

- Operators used in expressions have predefined precedence rules to automatically derive order of computation:

```
a - b * c;    // equivalent with a - (b * c)
```


Operators

Operator	Meaning
`=`	Assignment
`* *=`	Multiplication
`/ /=`	Division
`% %=`	Modulo
`+ +=`	Addition
`- -=`	Subtraction
`+` (unary)	Positive
`-` (unary)	Negative

Operators

- `+` and `-` operators can be used on pointers to refer to object a certain number of object sizes away
- `%` operator requires integer operands
- The result of assignment is itself an expression so it can be chained, but beware of implicit type conversions:

```
char c;
while ((c = getchar()) != EOF) { /* ... */ }
```

- Operators `x op= y` (like `x += y`) are equal to `x = x op (y)`, with this exception in mind:

```
x[++i] *= 2;             // i is incremented once
x[++i] = x[++i] * (2);   // i is incremented twice
```

Increment and decrement

Operator	Side effect	Value
x++	*x* increased by 1	Value before *x* was incremented
++x		Value after *x* was incremented
x--	*x* decreased by 1	Value before *x* was decremented
--x		Value after *x* was decremented

```
char s[10] = "ABC";
char *p = s;
printf("%c\n", *(p++)); // A
printf("%c\n", *(++p)); // C
```


Comparative operators

- There are several comparative operators:
  `< <= > >= == !=`
- For <, <=, >, >= one must be true:
  - Both operands are of arithmetic types
  - Both operands are pointers to object of the same type
- For == and != additionally these are permitted:
  - Both operands are pointers to functions of the same type
  - One operand is an object pointer, while the other is `void *` or null pointer
- Comparison of pointers depends on the relative positions of objects in memory
- Comparative operators have low precedence, so `a < b + 1` is equivalent to `a < (b + 1)`

Logical operators

- Logical expressions can be joined with && or || or negated with !
- Operators && and || are evaluated from left to right
- If the value of left operand is sufficient to determine the results, then the right operand is not evaluated at all

Bitwise operators

Operator	Meaning
`& &=`	AND
`| |=`	OR
`^ ^=`	XOR
`<< <<=`	Shift left
`>> >>=`	Shift right
`~`	Bitwise negation

Bitwise operators

- You can clear certain bits in an integer by performing bitwise AND with another integer containing 0s only in required bits:

```
char a = 0x7A; // 0x7A == 0111 1010
a &= 0xF8      // 0xF8 == 1111 1000
               // -----------------
               //         0111 1000
```

- You can set certain bits by similar operation using bitwise OR:

```
char a = ~0x2F //  0x2F == 0010 1111
               // ~0x2F == 1101 0000
a |= 0x06;     //  0x06 == 0000 0110
               // ------------------
               //          1101 0110
```


Bitwise operators

- For both operators >> and << the second operand must be non-negative and smaller then the bit width of the first operand
- With left shift, the vacated right positions are filled with zeros, and all bits shifted beyond the leftmost position are lost
- Generally, left shift by n bits corresponds to multiplication by 2^n, but beware of two's complement:

```
int i = INT_MAX;
printf("%x %x\n", i, i << 1);
  // 7fffffff fffffffe
printf("%d %d\n", i, i << 1);
  // 2147483647 -2
```

Bitwise operators

- With right shift, the bits shifted beyond position 0 are lost, and vacated positions are filled with:
  - Zeros if left operand is unsigned or signed but non-negative
  - Ones if left operand is negative

```
int j = -1;
unsigned int k = ~0;
printf("j=0x%x k=0x%x\n", j, k);
  // j=0xffffffff k=0xffffffff
j >>= 1;
k >>= 1;
printf("j=0x%x k=0x%x\n", j, k);
  // j=0xffffffff k=0x7fffffff
```


Memory addressing operators

Operator	Meaning
&	Address of
*	Object pointed by
[]	Object in array under index
.	Structure or union member
->	Structure or union member dereferenced from pointer

The & and * operators

- If *x* is of type *T*, then *&x* is of type *pointer to T*
- The operand of & operator must be an addressable `lvalue`:

```
float f = 0.0f, *pf;
pf = &f;             // OK
pf = &(f+1);         // ERROR!
```

- If *p* is a pointer, then **p* represents a function or object that *p* points to
- If *p* is an object pointer, then **p* is an `lvalue` (i.e. can assign value to it):

```
float f = 0.0f, *pf;
pf = &f;
*pf = 1.0f;          // f == 1.0f
```


The [] operator

- The `[]` operator allows to address specific elements in an array
- The expression `x[y]` is translated to `(*((x)+(y)))`, which has the following implications:

```
int t[] = { 1, 2, 3 };
printf("%d %d %d\n",
       t[1],         // 2
       1[t],         // 2
       *(t + 1));    // 2
```


The . and -> operators

- The left operand of `.` must be of structure or union type and right operand must be a valid name of that type's member
- The result of `.` is an `lvalue` except when the structure or union is returned by a function:

```
struct point {
    int x;
    int y;
}
struct point get_point();

struct point p1 = { 1, 2 };
p1.x = 20;                  // OK
int var = get_point().x;    // OK
get_point().x = 20;         // ERROR!
```


The . and -> operators

- The operator `->` works similarly to `.`, but its left operand is a pointer
- Expressions with `->` can be rewritten with `.`, because `p->m` equals `(*p).m`
- This works also the other way round, because `x.m` equals `(&x)->m` as long as `x` is an addressable `lvalue`
- Operators `.` and `->` can be combined with array subscripts:

```
array[i]->member;
```

Other operators

- `()` operator is used to specify arguments in a function call e.g. `log(x)`
- `,` represents sequential evaluation
- `sizeof` returns the number of bytes an object or type occupies in memory
- `? :` is a ternary conditional evaluation:

```
int a = b ? c : d;

if (b) {
    a = c;
} else {
    a = d;
}
```


`sizeof` operator

- The argument of `sizeof` can be a variable name, a type name or an expression. When the latter is used, the expression is not actually evaluated, but only its type is determined:

```
int x, *px;

sizeof x;     // number of bytes int variable occupies
sizeof(x)     // the same
sizeof(int)   // the same

sizeof(px);   // number of bytes a pointer to int occupie
sizeof(int*); // the same

sizeof(*px);  // number of bytes int occupies
              // but the expression is not evaluated
```


`sizeof` operator

```
int t[10];

sizeof(*t);   // number of bytes int occupies
sizeof(t[0]); // the same

sizeof(t);    // array size is known at compile-time
              // so this returns number of bytes
              // the entire array occupies

sizeof(t) / sizeof(*t);
              // number of elements in an array
```


Statements

- A **statement** specifies one or more actions to be performed e.g. assign value to a variable
- Statements are executed sequentially (in the order of appearance), taking into account jump and loops which might change the flow of the program
- Each statement ends with a semicolon `;`
- There is also a special *null statement* consisting of just the semicolon
- **Block statements** are enclosed between braces `{ }` and might contain multiple statements

`while` statement

- A **while** loop executes a statement as long as its controlling expression is true:

```
while (expression)
    statement
```

- With while loops, the condition is evaluated first and the statement is executed only if it holds true at least once
- Syntactically, the loop body consists of a single statement, but you can use block statement to perform multiple actions in a loop

```
int x = 10;
while (x > 0) {
    printf("%d\n", x);
    x--;
}
```


`for` statement

- A `for` loop is a statement with additional logic contained:

```
for (expression1; expression2; expression3)
    statement
```

- `expression1` is an initialization step (it is executed once at the beginning)
- `expression2` is a controlling term (loop is executed as long as it is true)
- `expression3` is an adjustment term (it is executed after loop body finishes and before `expression2` is evaluated)
- Any of the three expressions is optional, which makes this a valid loop:

```
for (;;)
```

for statement

- A missing controlling expression is considered to be always true
- The following two loops are equal:

```
while (expression)
    statement
for(; expression; )
    statement
```

- In fact, any *for* loop can be rewritten as a *while* loop (with surrounding statements) and vice versa
- If initialization step or adjustment term require more than one action, you can use comma operator:

```
for (i = 0, j = n; i < j; i++, j--)
    t[i] = t[j];
```


do...while statements

- A *do...while* is a loop in which the statement is executed at least once:

```
do statement while (expression);
```

- The *expression* is evaluated after statement is executed and loop continues if the result is true

```
int i;
do {
    t[i] = s[i];
} while (s[i++] != '\0');
```


Nested loops

- The inner *statement* in any loop may be another loop statement:

```
for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
        statement
```

- You can use the following statements to influence loop execution:
  - *continue* to immediately jump to control expression evaluation
  - *break* to immediately jump out of the loop
- Both *continue* and *break* are effective for the most nested loop that contains them

if statement

- An *if* statement has the following form:

```
if (expression) statement1 [ else statement2 ]
```

- The *else* clause is optional
- The *expression* is evaluated and if has non-zero value then *statement1* is executed
- Otherwise *statement2* is executed, if present

if statement

- When *if* statements are nested, then *else* corresponds to the last *if*:

```
if (n > 0)
    if (n % 2 == 0)
        puts("even");
    else
        puts("odd");
```


if statement

- Using braces to enclose a block can be used to influence this behaviour:

```
if (n > 0) {
    if (n % 2 == 0)
        puts("even");
} else
    puts("negative or zero");
```

`if` statement

- To select one out of many criteria, a cascaded *`if`* and *`else if`* statements can be used:

```
if (x < 10)
    statement1
else if (x < 20)
    statement2
else if (x < 30)
    statement3
else
    statement4
```

- The expressions will be executed one after another until one of them is true or *`else`* clause is reached

`switch` statement

- A *`switch`* statement causes the program flow to jump to one of several statements according to an integer expression:

```
switch (expression) statement;
```

- The *`statement`*, or a *switch body*, consists of *`case`* labels and at most one *`default`* label:

```
case constant: statement
default:       statement
```

- The *`expression`* is evaluated once and its value is compared against constant integers in *`case`* labels
- If the *`expression`* value equals one of cases' constants then the program jump to the case location

`switch` statement

- Constant values in *`case`* labels must be unique
- The *`default`* label is optional and can be placed anywhere in the *`switch`* body
- If *`default`* is not present and neither *`case`* constant matches the expression value, then the program continues execution directly after the *`switch`* statement
- After program jumps to one of *`case`* labels, it continues execution sequentially possibly going through other *`case`* labels and *`default`* one
- You can use *`break`* statement to explicitly stop execution at certain place

`switch` statement

- When *`x == 1`*, the following program will print *`1 2`*:

```
switch (x) {
    case 1:
        puts("1");
    case 2:
        puts("2");
}
```


Unconditional jumps

- Unconditional jumps allow the program to continue execution from a different place
- When such jump occur, variables from a left block are automatically destroyed
- *`break`* statement may occur in a loop or *`switch`* body and it jumps to the first statement after the loop or *`switch`*

```
while (expression1) {
    if (expression2)
        break;
    statement;
}
```


Unconditional jumps

- *`continue`* can be used only in a loop and it skips the rest of loop's body execution
- In *`while`* and *`do...while`* loops, *`continue`* jumps to control expression evaluation
- In *`for`* loops, *`continue`* jumps to evaluation of *`expression3`* (adjustment term)

```
for (i = 0; i < n; i++) {
    if (t[i] % 2 == 1) {
        continue;
    }
    // only even numbers are processed
}
```

Unconditional jumps

- *goto* statement result in a jump to a different location in the same function represented with a label:

```
  goto label;

label:
  statement
```

- Labels have their own namespace, so their names do not conflict with other identifiers
- Labels are just informations for *goto* instructions and does not impact code execution in any way if reached in a regular program flow

Unconditional jumps

- You should never use *goto* to jump into a block statement after some of its variables are defined:

```
if (x > 0)
    goto bad_example;
for (i = 0; i < n; i++) {
    int t[n];
  bad_example:
    t[i] = 0;     // ERROR!
}
```

- Also, you should avoid using *goto* at all! (Dijkstra, 1968)

Go to Statement Considered Harmful. E. Dijkstra. Communications of the ACM. 1968. *11*(3):147–148.

Unconditional jumps

- *return* statement ends execution of a current function and returns to the location where the function was invoked
- Functions may contain multiple *return* statements in their bodies:

```
int f(int n) {
    if (n % 2 == 0) return n - 2;
    return n + 2;
}
```

- Functions returning *void* type may use just *return* not followed by any expression
- When *return* is not present, these functions implicitly return when the execution reaches end of function block