P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #6

Introduction to C

History, Syntax, Types, Variables,
Arithmetic Operators, Control Structures

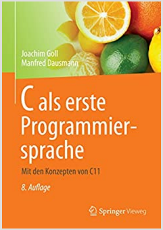
History

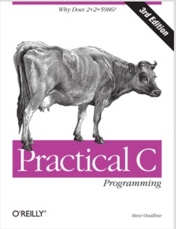
- First C version developped 1971–1973 by Dennis Ritchie at Bell Labs for reimplementing the UNIX operating system.
- In 1978 The C Programming Language by Kernighan and Ritchie was published and became a de facto standard (known today as K&R C).
- First official standard published in 1989 by ANSI (C89 or ANSI C).
- Minor revisions followed
  - ISO standard in 1990 (with small changes compared to C89)
  - Some additions in 1995
- Further revisions, C99 and C11, introduced e.g. some successful features from C++, multithreading or preprocessor enhancements.
- Most recent standard is C18 (mostly corrections to C11).

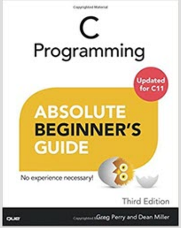
Properties

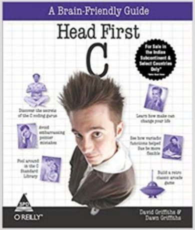
- Classical imperative/procedural language.
- Small set of keywords supplemented by a large set of standard library routines.
- Was intended as a high-level abstraction of assembler allowing
  - to write code close to the hardware
  - while still being modular, structured and as portable as possible
- C is most often used in systems programming (OS and embedded stuff) but also for application programming and signal processing.
- Still one of the most popular languages; had strong influence on C++, Objective-C, C#, Java, PHP and Perl.

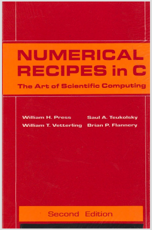
Book Suggestions











First C program

- C source code is in free-format
  - no fixed line length
  - no column rules (← FORTRAN77)
  - no indentation rules (← Python)
  - no line continuation character
- Commands terminated by semicolon ;
- Language is case–sensitive
- Code can be grouped in blocks using curly braces { . . . } (← scoping)

Classic example

Contents of source code file `helloWorld.c`

```
#include <stdio.h>

int main( void ) {

  /* this is a comment */
  printf( "Hello World!\n" );

  return 0;
}
```

First C program

- Comments
  - are ignored by the compiler
  - enhance readability of code, e.g. document function arguments
- Syntax
  - classic style:
    `/* this is a comment */`
    everything between opening /* and closing */ is ignored
  - C++-style also allowed:
    `// this is a comment!`
    ignore rest of line after //

Classic example

Contents of source code file `helloWorld.c`

```c
#include <stdio.h>

int main( void ) {

  /* this is a comment */
  printf( "Hello World!\n" );

  return 0;
}
```

First C program

- Functions:
  - Functions are subprograms (remember: small set of keywords plus large set of library functions)
  - C programs consist of one or more functions
  - `main` is the function called by the OS at program start (must always be there)
- `include` directive:
  - used to add contents of header file `stdio.h` to our program
  - informs compiler on details of function `printf` (prototyping)

Classic example

Contents of source code file `helloWorld.c`

```c
#include <stdio.h>

int main( void ) {

  /* this is a comment */
  printf( "Hello World!\n" );

  return 0;
}
```

First C program

- `return` statement can be used to end a function.
- This will hand control back to the caller
- If done in `main` this is the calling process.
- `return 0` in demo ends program and returns integer value 0 to the calling process.
- `int main` sets the return value for main to be integer (required by standard: OS progamming)

Classic example

Contents of source code file `helloWorld.c`

```
#include <stdio.h>

int main( void ) {

  /* this is a comment */
  printf( "Hello World!\n" );

  return 0;
}
```

Variables

Variable declaration

As an imperative language C uses variables. These are containers in which we store (intermediate) results of our computations.

Each variable must be declared before we can use it. Optionally we can initialise it with a value at definition, too.

Defining a variable: `<data type> <variable name>[= value];`

Names are case sensitive. May contain numbers, but not start with one. Length is compiler dependent.

Note that a variable is **always** associated with a data type. This association is fixed and cannot be changed.

Variables – Example

- definition of variables of same type can be separated by comma
- variables already defined can be used in intialisation of others
- `printf()`: need different format specifiers for different types
- constant values can be given directly (as literals)

program output:

```
a = 1
b = 3
c = 1.000000e-01
d = 1.000000e-01
e = e or
e = 101
```

```
#include<stdio.h>
int main( int argc, char** argv ) {

  /* define some variables */
  int a = 1, b = a + 2;
  double c = 0.1;
  double d = 1e-1;
  char e = 'e';

  /* Just write some output */
  printf( "a = %d\n", a );
  printf( "b = %d\n", b );
  printf( "c = %e\n", c );
  printf( "d = %e\n", d );
  printf( "e = %c or\ne = %d\n", e, e );
}
```

example by Christian Feichtinger, FAU

Elementary Numerical Data

The basic data numerical simulations deal with are numbers.
Mathematicians know quite different sets of data, e.g.

$\mathbb{N}$	natural numbers	$\{1, 2, 3, 4, \ldots\}$
$\mathbb{Z}$	whole numbers	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathbb{Q}$	rational numbers	$\frac{n}{d}$ with $n \in \mathbb{Z}$, $d \in \mathbb{N}$
$\mathbb{R}$	real numbers	e.g. 0, $\frac{1}{2}$, $\sqrt{2}$, π
$\mathbb{C}$	complex numbers	$u + iv$ with $u, v \in \mathbb{R}$, $i = \sqrt{-1}$

Numerical algorithms are executed by computers, so how are numbers represented there?

Computer Systems & Numbers

Computer systems in hardware typically provide support for two different types of data/numbers:

- integral numbers (integers) are a subset of $\mathbb{Z}$
- machine numbers are a subset of $\mathbb{R}$ in so called floating point representation (we will treat this in detail later on)

This is extended by software, e.g.

- by data-types for complex numbers
- or by mapping integers to letters in a characters set (e.g. ASCII or UTF-8)

Basic Types

- In C one distinguishes basic types (PODs :-) and derived types, which can be built from basic types.
- C offers 22 basic types (20 in C99, 12 in C89)
- Large number mostly due to 11 different integral types
- Major differences between ANSI-C and FORTRAN77
  - no boolean/logical data type
  - no complex data type
  - a special `void` type
  - C does not distinguish functions and subroutines
- C99 introduced `_Bool` and three `_Complex` types (among others)

Integral Types

- C offers the following signed integral types
  - signed char
  - short
  - int
  - long
  - long long
- All of these also have an unsigned counterpart
- Standard does not specify sizes/ranges, but minimally requires

  char (8-bit), short (16-bit), int (16-bit), long (32-bit), long long (64-bit)

  (but see `int64_t` etc. in C99)
- (un)signed char & char (typically) are 1–byte data types

Example: 64-bit Data Models

In C/C++ programming one distinguishes different data models for 64-bit systems. These differ in the size of the intrinsic integral data types.

data model	short	int	long	long long	pointers
LP64	16	32	64	64	64
ILP64	16	64	64	64	64
SILP64	64	64	64	64	64
LLP64	16	32	32	64	64

The available data model in general is compiler dependent.

Ranges for LP64

data-type	size in bytes	largest value
unsigned char	1	255
unsigned short int	2	65,535
unsigned int	4	4,294,967,295
unsigned long	8	18,446,744,073,709,551,615
unsigned long long	8	18,446,744,073,709,551,615
size_t / (void *)	8	18,446,744,073,709,551,615

(note: long long only supported since C99)

Floating Point Types

- C supports three floating point types for real-valued numbers
- In an IEEE conforming setting we have

C type	IEEE precision	F77 equivalent
`float`	single	`real`
`double`	double	`double precision`
`long double`	(depends)	—

- `long double` might be double precision (fall-back to `double`) or double extended (80-bit on x86/86-64) or quadruple/binary128.
- C99 added corresponding complex types
  (shows migration system programming $\longrightarrow$ application language)

Constants (1/2)

- (Literal) Constants

  are data values that are known at compile time and appear in explicit form in the source code, e.g. `A = B + 2.0`

- (Named) Constants

  are also known at compile time and associated with an identifier

```
const double pi = 3.14159265358979;
...
val = sin( x * pi )
```

- Macros

  are meta-constant or symbolic constants; replaced in the code during pre-processing

```
#define PI = 3.14159265358979
```

Constants (1/2)

- (Literal) Constants

  are data values that are known at compile time and appear in explicit form in the source code, e.g. `A = B + 2.0`

- (Named) Constants ← true in C++, not 100% true in C!

  are also known at compile time and associated with an identifier

```
const double pi = 3.14159265358979;
...
val = sin( x * pi )
```

- Macros

  are meta-constant or symbolic constants; replaced in the code during pre-processing

```
#define PI = 3.14159265358979
```

Constants (2/2)

data-type	example literal constant
int	`42`, `-12`
long int	`42L`, `-3L`, `-12l`
long long int	`42LL`, `-3LL`, `-12ll`
unsigned int	`42u`
unsigned long int	`42ul`
unsigned long long int	`42ull`, `123ULL`
double	`-15.0`, `1.0e-2`, `123.456e10`
float	`3.25f`, `1.0e-2f`, `123.456e10f`
char	`'b'`, `'\n'` (newline), `'\t'` (tab),
bool^2	true, false†

†requires inclusion of `stdbool.h`

Derived Types (1/3)

- C offers four different ways to built new types from the basic ones
- `typedef` command allows to declare new typenames and aliases
- arrays
  - `int list[10];` generates an array of 10 `int`s
  - index range always starts at 0
  - multi-dimensional arrays possible
  - dynamic memory allocation requires pointers
    (variable length arrays (VLA) did not really kick off)
- enums
  - an integral data type for a precisely defined set of values
  - values represented by symbolic names
  - `typedef enum {GREEN, YELLOW, RED, YELLOW_RED} trafficLight;`

Derived Types (2/3)

- unions
  - a union makes several variables overlap in main memory

```
union endian {
  int i;
  char c;
};
```

  - not commonly used in scientific programming
  - some similarity with `EQUIVALENCE` in Fortran

Derived Types (3/3)

- structs
  - a struct allows to combine several variables into a single unit
  - invididual members can be accessed with the selection operator .
- example:

```
/* declare new type to store
   information on a student */
typedef struct {
   char firstName[100];
   char familyName[100];
   int number;
   int semester;
} studentInfo;
```

```
/* define variable of
   new type */
studentInfo johnDoe;

/* John is a freshman */
johnDoe.semester = 1;

/* an array of students */
studentInfo course[20];
```

Arithmetic Operations

The four basic arithmetic operations are directly available via the following four operators

symbol	binary operation	unary operation
+	addition	positive sign
-	subtraction	negative sign
*	multiplication	—
/	division	—

Opposed to e.g. Fortran or Python, C does not offer an operator for exponentiation.

Hierarchy of Evaluation

The order of evaluation of algebraic expressions follows the standard mathematical rules, i.e.

(1) expressions in braces
(2) exponentiation
(3) multiplication & division
(4) addition, subtraction & algebraic sign

Competing arithmetic expressions are evaluated left to right:

```
A / B + C   ⟷   ( A / B ) + C
A + B + C   ⟷   ( A + B ) + C
A / B * C   ⟷   ( A / B ) * C
```

Some Terminology

Expression vs. Statement

- Expressions have a return type, statements do not.
- Statements have to be terminated by a ';'
- Most of the time expression are used inside statements
- Examples:
  - `x + y` (expression)
  - `x * y` (expression)
  - `x = x + y/2;` (statement)
  - `a = (b = 2*c);` (statement)

Type of Result? (1/2)

- The result of an arithmetic expression depends on the types of the operands involved.
- If both operands of a binary operation are of the same type, then this is the type of the result

```
int a = 1, b = 2, c;
c = a + b;   // (a+b) is of type int; can be stored in c
```

- But what, if they are different?

```
double val = 1.0;
int factor = 2;
float res;
res = factor * val;
```

Type of Result? (2/2)

- If different operands are involved, they are first converted ('cast') to the 'highest' type involved, e.g.

int ⟶ float ⟶ double

- Potentially also the result needs to be cast for the assignment:

```
double val = 3.0;
int factor = 2;
float res;
res = factor * val;
```

1. 2 (int) promoted to 2.0 (double)
2. result 6.0 (double) down-cast to 6.0f (float)

```
GNU compiler might warn:
conversion to ‘float’ from ‘double’ may alter its value [-Wfloat-conversion]
   res = factor * val;
         ^ ~~~~~~
```

Integer Division (1/2)

- With Integer Division the result of the operation is not rounded to the nearest neighbour, but truncated.

```
int a = 3;
int b = 2;
int c = a/b;
printf( "c stores %d\n", c );
```

- This is especially tricky in mixed expressions:

```
double a = 3.0;
double b = 2.0;
double c = 1/5*(a+b);
double d = (a+b)/5;
double e = (a+b)*0.2;
printf( "c=%2f, d=%f, e=%f\n", c, d, e );
```

Integer Division (1/2)

- With Integer Division the result of the operation is not rounded to the nearest neighbour, but truncated.

```
int a = 3;
int b = 2;
int c = a/b;
printf( "c stores %d\n", c );
```

→ c stores 1

- This is especially tricky in mixed expressions:

```
double a = 3.0;
double b = 2.0;
double c = 1/5*(a+b);
double d = (a+b)/5;
double e = (a+b)*0.2;
printf( "c=%2f, d=%f, e=%f\n", c, d, e );
```

Integer Division (1/2)

- With Integer Division the result of the operation is not rounded to the nearest neighbour, but truncated.

```
int a = 3;
int b = 2;
int c = a/b;
printf( "c stores %d\n", c );
```

$\rightarrow$ c stores 1

- This is especially tricky in mixed expressions:

```
double a = 3.0;
double b = 2.0;
double c = 1/5*(a+b);
double d = (a+b)/5;
double e = (a+b)*0.2;
printf( "c=%2f, d=%f, e=%f\n", c, d, e );
```

$\rightarrow$ c=0.000000, d=1.000000, e=1.000000

Integer Division (2/2)

Modulo Operator

The modulo operator % can be used to obtain the remainder after division.

C11 ensures that the following identity holds:

```
(a/b) * b + (a%b) = a
```

code	result	
7 % 2	1	*number is odd*
8 % 3	2	
−7 % 2	-1	for C99 and C11
−7 % 2	1 or -1	for C90

Assignment Operators

- simple assignment operator
  `A = B` (value of B is stored in/copied to A)

- compound assignment operators combine an operation with the assignment of the result; for the four basic arithmetic operations these are:

compound operator	corresponds to
`A += B`	`A = A + (B)`
`A -= B`	`A = A - (B)`
`A *= B`	`A = A * (B)`
`A /= B`	`A = A / (B)`

In/Decrementation Operators (1/2)

Let a be a variable (more precisely an L-value):

a++	post-fix increment	value of a is returned, afterwards value of a is increased by 1
++a	pre-fix increment	value of a is increased by 1, then it is returned
a--	post-fix decrement	value of a is returned, afterwards value of a is decreased by 1
--a	pre-fix decrement	value of a is decreased by 1, then it is returned

In/Decrementation Operators (2/2)

What will this codelet print?

```
int a = 1;
int b = 2;
++b;
int c = b--;

printf( "%d %d %d", a, b++, c );
```

It prints:

and this codelet?

```
int a = 3;
int b = 2;
int c = (a+b);
int d = --c;

printf( "%d %d", ++c, d++ );
```

It prints:

In/Decrementation Operators (2/2)

What will this codelet print?

```
int a = 1;
int b = 2;
++b;
int c = b--;

printf( "%d %d %d", a, b++, c );
```

It prints:

1 2 3

and this codelet?

```
int a = 3;
int b = 2;
int c = (a+b);
int d = --c;

printf( "%d %d", ++c, d++ );
```

It prints:

In/Decrementation Operators (2/2)

What will this codelet print?

```
int a = 1;
int b = 2;
++b;
int c = b--;

printf( "%d %d %d", a, b++, c );
```

It prints:

```
1 2 3
```

and this codelet?

```
int a = 3;
int b = 2;
int c = (a+b);
int d = --c;

printf( "%d %d", ++c, d++ );
```

It prints:

```
5 4
```

Terminology

Scope

The term scope refers to the spatial (in which parts of the code) and temporal (from when up to when) lifetime of an entity (such as a variable).

- The question is e.g. when we define a variable, where and when do we have read/write access to it?
- Can we have multiple variables with the same name in our code?

Blocks

Blocks

- In C the curly braces can be used to group a collection of statements. This is refered to as a block.
- Syntactically a block can be used anywhere a single statement could.
- A block constitutes a (local) scope of its own.

Example: Block & Scope

- Lines [5–7] define three variables a, b, c.

  Their spatial scope is the main function. They exist during the complete execution of the program.

- Lines [9] and [14] mark lines inbetween as block.

- Line [11] defines another variable named a

  Its scope is the surrounding block. It is different from the outer a, which is 'shadowed'.

- What will the program print?

```
1  #include <stdio.h>
2
3  int main() {
4
5    int a = 0;
6    int b = 2;
7    int c;
8
9    {
10     printf( "a = %d, ", a );
11     int a = 3;
12     printf( "a = %d, ", a );
13     c = a + b;
14   }
15
16   printf( "a = %d, ", a );
17   printf( "c = %d\n", c );
18 }
```

Example: Block & Scope

- Lines [5–7] define three variables a, b, c.

  Their spatial scope is the main function. They exist during the complete execution of the program.
- Lines [9] and [14] mark lines inbetween as block.
- Line [11] defines another variable named a

  Its scope is the surrounding block. It is different from the outer a, which is 'shadowed'.
- What will the program print?

  `a = 0, a = 3, a = 0, c = 5`

```
1  #include <stdio.h>
2
3  int main() {
4
5    int a = 0;
6    int b = 2;
7    int c;
8
9    {
10     printf( "a = %d, ", a );
11     int a = 3;
12     printf( "a = %d, ", a );
13     c = a + b;
14   }
15
16   printf( "a = %d, ", a );
17   printf( "c = %d\n", c );
18 }
```

Booleans (1/2)

- 'True' and 'False' are the two truth values of logic and Boolean algebra.
- Many languages sport special boolean datatypes for storing and literals to represent these.
- Example: FORTRAN77 has a `logical` datatype and the two literals `.TRUE.` and `.FALSE.`
- C originally offered no special boolean type. Instead truth values are represented by integers with

'False' $\equiv 0$, 'True' $\equiv 1$ (or any non-zero value)

Booleans (2/2)

- C99 introduced `_Bool` as datatype (internally an integer).
- When we include `stdbool.h` we can use
  - `bool` (just a typedef)
  - `true` and `false` (aliases for 1 and 0)
- But there are no special format descriptors.
- Code on the right will just print `0 is not 1`

```c
#include <stdio.h>
#include <stdbool.h>

int main() {

  // always works
  _Bool bOne = 0;

  // requires header
  bool bTwo = true;

  printf( "%d is not %d\n",
          bOne, bTwo );
}
```

Comparison Operators

Boolean values result from applying comparison operators:

C	F77	math symbol	meaning
`>`	`.GT.`	$>$	greater than
`>=`	`.GE.`	$\geqslant$	greater than or equal to
`<`	`.LT.`	$<$	less than
`<=`	`.LE.`	$\leqslant$	less than or equal to
`==`	`.EQ.`	$=$	equal to
`!=`	`.NE.`	$\neq$	not equal to

`4*9 == 42` $\longrightarrow$ `false`

`0.5 >= 0.01` $\longrightarrow$ `true`

Logical Operators

Five logical operators exist for manipulating boolean variables and boolean expressions:

C	F77	math symbol	meaning
`!`	`.NOT.`	$\neg$	negation
`&&`	`.AND.`	$\wedge$	logical and
`||`	`.OR.`	$\vee$	logical or
`==`	`.EQV.`	$\equiv$	equivalence
`!=`	`.NEQV.`	$\not\equiv$	antivalence

Boolean Tables

unary operation

a	$\neg a$
T	F
F	T

`.NOT.` / `!`

binary operations

	T	F
T	T	F
F	F	F

`.AND.` / `&&`

	T	F
T	T	T
F	T	F

`.OR.` / `||`

	T	F
T	T	F
F	F	T

`.EQV.` / `==`

	T	F
T	F	T
F	T	F

`.NEQV.` / `!=`

Control Flow

In imperative programming control structures allow to steer the flow of control, i.e. (the order in) which individual statements, instructions or function calls are executed or evaluated.

explicit jumps

allow the unconditional transfer of control to another part of the program

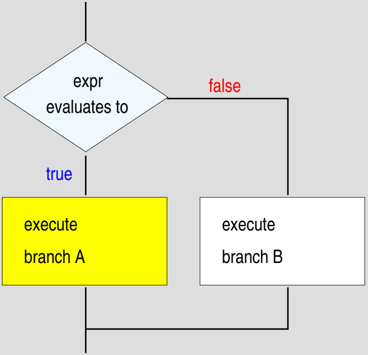
loops

can be used to repeatedly execute the same (group of) statements

branches

allow to execute different groups of statements depending on the current state of the program

Standard IF-THEN-ELSE (1/3)



flow chart representation

Assume that expr is an expression that evaluates to a boolean value (e.g. `norm <= tol`), depending on this test different program code may be executed:

```
if ( expr ) {
  branch A: group of statements
}
else {
  branch B: group of statements
}
```

Standard IF-THEN-ELSE (2/3)

Assume that we want to print the specialisation of a geosciences bachelor student and that the specialisation is encoded in the `int` variable `stype`.

We can do this by writing three separate IF statements.
Introduces unnecessary comparisons!
(think of stype $= 1$).

```
if( stype == 1 ) printf( "Geowiss (Geophysik)" );
if( stype == 2 ) printf( "Geowiss (Geologie)" );
if( stype == 3 ) printf( "Geowiss (Mineralogie)" );
```

Standard IF-THEN-ELSE (3/3)

Alternatively we can solve the problem by nesting if statements using `else`.

Note that there is no special EL(SE)IF keyword in C. An `else` always belongs to the last previous `if`.

Advantages:

- fewer comparisons for stype $= 1$
- easy to add error handling for invalid value

```
if( stype == 1 ) printf( "Geowiss (Geophysik)" );
else if( stype == 2 ) printf( "Geowiss (Geologie)" );
else if( stype == 3 ) printf( "Geowiss (Mineralogie)" );
else printf( "Error" );
```

Summary

- Conditional branches allow to make decisions at runtime.
- Code on the right will execute one of the branches depending on the result of evaluating logical <expression1> and <expression2>.
- Since there initially was no boolean data type, C uses a result of
  - 0 for false
  - $\neq 0$ for true

```
if ( <expression1> ) {
... branch A ...
}
else if ( <expression2> ) {
... branch B ...
}
else {
... branch C ...
}
```

Switch–Statement

- Often branch to take depends on parameter value from a certain finite list of choices.
- Many languages support a special construct for this.
- In C it's `switch/case`.
- Special keyword `default` in case nothing else fits.
- Concept works well together with `enum` types.

```
switch( <variable> ) {
case <constant 1>:
   ... branch 1 ...
   break;
case <constant 2>:
   ... branch 2 ...
   break;
...
case <constant n>:
   ... branch n ...
   break;
default:
   ... error? ...
}
```

Example: Switch–Statement

- In our computational fluid code we must distinguish certain types of boundary conditions
  - NOSLIP
  - FREESLIP
  - INFLOW
  - OUTFLOW
- We construct an enumeration data–type `typedef enum {NOSLIP, FREESLIP, INFLOW, OUTFLOW} BC_TYPE;`
- And use switch/case for checking current simulation type

```
BC_TYPE bcType;
bcType = INFLOW;
switch( bcType ) {
case NOSLIP:
  ...
  break;
case FREESLIP:
  ...
  break;
...
default:
  printf( "ERROR!\n" );
}
```

Loops

- C supports three different types of loop variants
  1. `for` – loops
  2. `while` – loops
  3. `do while` – loops
- The most commonly used and most flexible one is the `for`–loop.
- Infinite loops are allowed.

For Loops (1/4)

The `for` construct has three parts:

1. code in front of first `;` executed before loop
2. code between `;`'s is evaluated before each loop; only if it evaluates to true is the loop executed (again)
   - can be check on counter value
   - or any logical expression
3. code after last `;` is executed after end of each loop

```
#include <stdio.h>

int main() {

  int k;

  for( k = 0; k < 10; k++ ) {
    printf( "%d ", k );
  }
}

// outputs
// 0 1 2 3 4 5 6 7 8 9
```

For Loops (2/4)

- Any of the three parts can be empty.
- If the middle one is empty, then it is non-zero $\equiv$ true.
- Also $7 \neq 0$.
- All three loops are infinite.
- What will the program print?

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

int main() {

  for( ; ; ) {
    printf( "." );
  }

  for( ; 7 ; ) {
    printf( "?" );
  }

  for( ; true ; ) {
    printf( "-" );
  }
}
```

For Loops (3/4)

- We can define the loop counter variable as part of the loop. Then this is its scope!
- We are allowed to change the counter inside the loop body.

#	k on entry	2*k	k++
1	0	0	1
2	1	2	3
3	3	6	7
4	7	14	15
5	15	30	31

```
#include <stdio.h>

int main() {

  for( int k = 0; k < 20; k++ ) {
    k *= 2;
    printf( "%d .. ", k );
  }

}

// outputs
// 0 .. 2 .. 6 .. 14 .. 30 ..
```

For Loops (4/4)

- If loop counter exists outside loop we can access its value after the loop terminates.
- This will be the value for which the test failed!
- So the code on the right prints

```
9 is odd
7 is odd
5 is odd
3 is odd
1 is odd
```

```
#include <stdio.h>

int main() {

  int k;

  for( k = 10; k >= 2; k-- ) {
    if( k%2 == 1 ) {
      printf( "%d is odd\n", k );
    }
  }

  if( k%2 == 1 ) {
    printf( "%d is odd\n", k );
  }
}
```

Other Loops (1/3)

- infinite loops and general stopping checks are typically used with
  - while loop, here condition is checked before loop is started
    `while( <expression> ) { ... }`
  - do loop, here condition is checked after end of loop
    `do { ... } while( <expression> );`
- `break` statement allows to terminate a (infinite) loop completely
- while the `continue` statement allows to prematurely terminate a single loop execution

Other Loops (2/3)

Example: while loop

```
#include <stdio.h>

int main() {

  int k = 3;

  while( k > 0 ) {
    printf( "%d .. ", --k );
  }
}
```

left code prints: 2 .. 1 .. 0 ..

Example: do-while loop

```
#include <stdio.h>

int main() {

  int k = 2;

  do {
    printf( "%d .. ", k-- );
  }
  while( k >= 0 );
  }
}
```

right code prints: 2 .. 1 .. 0 ..

Other Loops (3/3)

Example: while loop

```c
#include <stdio.h>

int main() {

  int k = 5;

  while( k < 5 ) {
    printf( "%d .. ", k++ );
  }
}
```

left code prints: nothing

Example: do-while loop

```c
#include <stdio.h>

int main() {

  int k = 5;

  do {
    printf( "%d .. ", k++ );
  }
  while( k < 5 );
}
```

right code prints: 5 ..

Break vs. Continue

- The first loop will print
  `1 .. 2 .. 3 ..`
  - Once `k` reaches 4 the `break` get's executed and the loop is terminated completely.
- The second loop will print
  `2 .. 4 .. 6 .. 8 ..`
  - Whenever the `if`-condition is fulfilled, `continue` is executed and the current iteration terminated.

```
#include <stdio.h>

int main() {
  int k = 1;
  while( k > 0 ) {
    printf( "%d .. ", k );
    k++;
    if( k == 4 ) break;
  }

  for( k = 1; k <= 8; k++ ) {
    if( k%2 == 1 ) continue;
    printf( "%d .. ", k );
  }
}
```

Explicit Jumps

explicit jumps

allow the unconditional transfer of control to another part of the program

Examples of explicit jumps are:

1. function calls: `sin(x)`
2. the `return` statement
3. use of `goto`
   (yes, also C offers goto :-)

```
#include <stdio.h>

int main() {

  int err = 1;
  printf( "three\n" );
  printf( "two\n" );
  printf( "one\n" );
  if( err > 0 ) goto abort;
  printf( "zero\n" );
  printf( "ignition\n" );
  abort:
  printf( "mission aborted\n" );
}
```

generally use of goto is “considered harmful” and should be avoided
($\rightarrow$ spaghetti code)

only few cases were it is warranted