Imperative programming

Introduction

Kozsik Tamás, Porkoláb Zoltán

ELTE Eötvös Loránd Tudományegyetem

Outline

Course Goals

- Concepts
- Terminology
- Knowledgeable language usage
    - Imperative Programming
    - Procedural Programming
    - Modular Programming
- Only partly: programming skills
- Usage of Linux and CLI tools
    - see Jurassic Park (link)
    - and TadeusTaD note: `$su root -c "killall raptors"`

Programming Languages Used

- C (Why do we learn C? link!)

Course Format

- Lecture
- Practice
- Consultation

Passing the Course

- Continuous: small tests on practice
- End of Semester: exam

Expected Workload

Total 150 work hours

- Contact hours: 13x5
- At home practice and learning: 12x5
- Preparation for exam: 20
- Exam: 5

More information

Course homepage (Hungarian yet):

http://kto.web.elte.hu/hu/oktatas/

In canvas

http://canvas.elte.hu/

Outline

Programming Languages

- Human-Machine Communication
- Human-Human Communication

Programming Paradigms

Cognitive schemes, required language tools

For example:

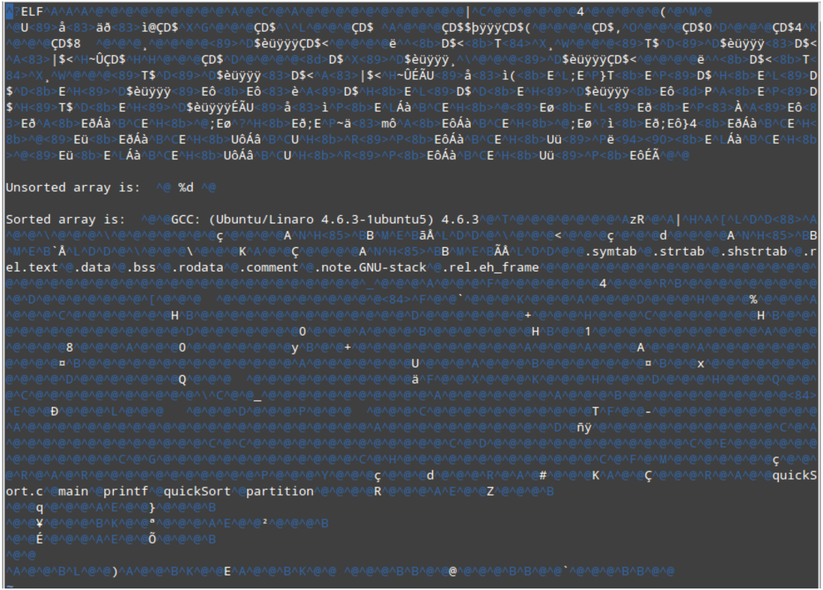
- Imperative Programming
- Functional Programming
- Logic Programming

- Procedural Programming
- Modular Programming
- Object-oriented Programming

- Sequential Programming
- Concurrent Programming
- Parallel Programming
- Distributed Programming

- Aspect-oriented Programming
- Component-based Programming
- Service-oriented Programming
- Contract-based Programming

Machine code



Assembly

```
quickSort:
.LFB1:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $40, %esp
        movl    12(%ebp), %eax
        cmpl    16(%ebp), %eax
        jge     .L6
        movl    16(%ebp), %eax
        movl    %eax, 8(%esp)
        movl    12(%ebp), %eax
        movl    %eax, 4(%esp)
```

„High-level' ' Programming Languages

- Fortran
- LISP
- Algol
- COBOL
- BASIC
- **C**

etc.

Modern, Comfortable Languages

- **Python**
- **Haskell**
- C++
- Java
- Ada

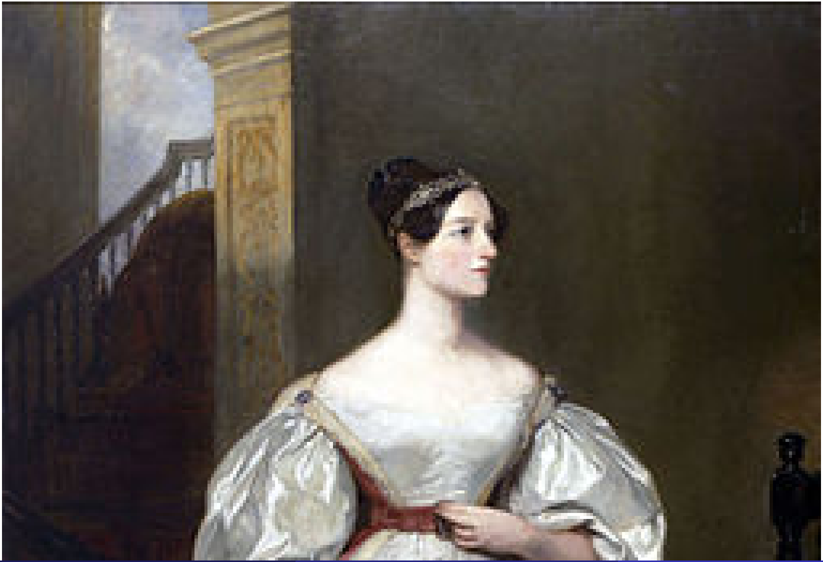
etc.

Ada Lovelace (Analytical Engine, Charles Babbage)

Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data. $^{1}V_1$ (1)	$^{1}V_2$ (2)	$^{1}V_3$ (n)	Working Variables. $^{0}V_4$	$^{0}V_5$	$^{0}V_6$	$^{0}V_7$	$^{0}V_8$	$^{0}V_9$	$^{0}V_{10}$	$^{0}V_{11}$	$^{0}V_{12}$	$^{0}V_{13}$	Result Variables. $^{1}V_{21}$ B_1 in a decimal fraction. (B_1)	$^{1}V_{22}$ B_3 in a decimal fraction. (B_3)	$^{1}V_{23}$ B_5 in a decimal fraction. (B_5)	$^{0}V_{24}$ (B_7)
1	×	$^{1}V_2 \times {}^{1}V_3$	$^{1}V_4, {}^{1}V_5, {}^{1}V_6$	$\left\{ {}^{1}V_2 = {}^{1}V_2 \atop {}^{1}V_3 = {}^{1}V_3 \right\}$	$= 2n$	...	2	n	$2n$	$2n$	$2n$											
2	−	$^{1}V_4 - {}^{1}V_1$	$^{2}V_4$	$\left\{ {}^{1}V_4 = {}^{2}V_4 \atop {}^{1}V_1 = {}^{1}V_1 \right\}$	$= 2n - 1$	1	...	...	$2n-1$													
3	+	$^{1}V_5 + {}^{1}V_1$	$^{2}V_5$	$\left\{ {}^{1}V_5 = {}^{2}V_5 \atop {}^{1}V_1 = {}^{1}V_1 \right\}$	$= 2n + 1$	1	...	...	...	$2n+1$												
4	÷	$^{2}V_5 \div {}^{2}V_4$	$^{1}V_{11}$	$\left\{ {}^{2}V_5 = {}^{0}V_5 \atop {}^{2}V_4 = {}^{0}V_4 \right\}$	$= \frac{2n-1}{2n+1}$	...	...	...	0	0	...	...	...	...	...	$\frac{2n-1}{2n+1}$						
5	÷	$^{1}V_{11} \div {}^{1}V_2$	$^{2}V_{11}$	$\left\{ {}^{1}V_{11} = {}^{2}V_{11} \atop {}^{1}V_2 = {}^{1}V_2 \right\}$	$= \frac{1}{2} \cdot \frac{2n-1}{2n+1}$	...	2	...	...	...	...	...	...	...	...	$\frac{1}{2} \cdot \frac{2n-1}{2n+1}$						
6	−	$^{0}V_{13} - {}^{2}V_{11}$	$^{1}V_{13}$	$\left\{ {}^{2}V_{11} = {}^{0}V_{11} \atop {}^{0}V_{13} = {}^{1}V_{13} \right\}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$	...	...	...	...	...	...	...	...	...	...	0	....	$-\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$				
7	−	$^{1}V_3 - {}^{1}V_1$	$^{1}V_{10}$	$\left\{ {}^{1}V_3 = {}^{1}V_3 \atop {}^{1}V_1 = {}^{1}V_1 \right\}$	$= n - 1\ (= 3)$	1	...	n	...	...	...	...	...	...	$n-1$							
8	+	$^{1}V_2 + {}^{0}V_7$	$^{1}V_7$	$\left\{ {}^{1}V_2 = {}^{1}V_2 \atop {}^{0}V_7 = {}^{1}V_7 \right\}$	$= 2 + 0 = 2$	...	2	...	...	...	...	2										
9	÷	$^{1}V_6 \div {}^{1}V_7$	$^{3}V_{11}$	$\left\{ {}^{1}V_6 = {}^{1}V_6 \atop {}^{0}V_{11} = {}^{3}V_{11} \right\}$	$= \frac{2n}{2} = A_1$	...	...	...	...	...	$2n$	2	...	...	...	$\frac{2n}{2} = A_1$						
10	×	$^{1}V_{21} \times {}^{3}V_{11}$	$^{1}V_{12}$	$\left\{ {}^{1}V_{21} = {}^{1}V_{21} \atop {}^{3}V_{11} = {}^{3}V_{11} \right\}$	$= B_1 \cdot \frac{2n}{2} = B_1 A_1$	...	...	...	...	...	...	...	...	...	...	$\frac{2n}{2} = A_1$	$B_1 \cdot \frac{2n}{2} = B_1 A_1$	....	B_1			
11	+	$^{1}V_{12} + {}^{1}V_{13}$	$^{2}V_{13}$	$\left\{ {}^{1}V_{12} = {}^{0}V_{12} \atop {}^{1}V_{13} = {}^{2}V_{13} \right\}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \cdot \frac{2n}{2}$	...	...	...	...	...	...	...	...	...	...	....	0	$\left\{-\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \cdot \frac{2n}{2}\right\}$				
12	−	$^{1}V_{10} - {}^{1}V_1$	$^{2}V_{10}$	$\left\{ {}^{1}V_{10} = {}^{2}V_{10} \atop {}^{1}V_1 = {}^{1}V_1 \right\}$	$= n - 2\ (= 2)$	1	...	...	...	...	...	...	...	...	$n-2$							
13	−	$^{1}V_6 - {}^{1}V_1$	$^{2}V_6$	$\left\{ {}^{1}V_6 = {}^{2}V_6 \atop {}^{1}V_1 = {}^{1}V_1 \right\}$	$= 2n - 1$	1	...	...	...	...	$2n-1$											
14	+	$^{1}V_1 + {}^{1}V_7$	$^{2}V_7$	$\left\{ {}^{1}V_1 = {}^{1}V_1 \atop {}^{1}V_7 = {}^{2}V_7 \right\}$	$= 2 + 1 = 3$	1	...	...	...	...	...	3										
15	÷	$^{2}V_6 \div {}^{2}V_7$	$^{1}V_8$	$\left\{ {}^{2}V_6 = {}^{2}V_6 \atop {}^{2}V_7 = {}^{2}V_7 \right\}$	$= \frac{2n-1}{3}$	...	...	...	...	...	$2n-1$	3	$\frac{2n-1}{3}$									
16	×	$^{1}V_8 \times {}^{3}V_{11}$	$^{4}V_{11}$	$\left\{ {}^{1}V_8 = {}^{0}V_8 \atop {}^{3}V_{11} = {}^{4}V_{11} \right\}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3}$	...	...	...	...	...	...	...	0	...	...	$\frac{2n}{2} \cdot \frac{2n-1}{3}$						
17	−	$^{2}V_6 - {}^{1}V_1$	$^{3}V_6$	$\left\{ {}^{2}V_6 = {}^{3}V_6 \atop {}^{1}V_1 = {}^{1}V_1 \right\}$	$= 2n - 2$	1	...	...	...	...	$2n-2$											
18	+	$^{1}V_1 + {}^{2}V_7$	$^{3}V_7$	$\left\{ {}^{2}V_7 = {}^{3}V_7 \atop {}^{1}V_1 = {}^{1}V_1 \right\}$	$= 3 + 1 = 4$	1	...	...	...	...	...	4										
19	÷	$^{3}V_6 \div {}^{3}V_7$	$^{1}V_9$	$\left\{ {}^{3}V_6 = {}^{3}V_6 \atop {}^{3}V_7 = {}^{3}V_7 \right\}$	$= \frac{2n-2}{4}$	...	...	...	...	...	$2n-2$	4	...	$\frac{2n-2}{4}$	...	$\left\{\frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{3} = A_3\right\}$						
20	×	$^{1}V_9 \times {}^{4}V_{11}$	$^{5}V_{11}$	$\left\{ {}^{1}V_9 = {}^{0}V_9 \atop {}^{4}V_{11} = {}^{5}V_{11} \right\}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{4} = A_3$	...	...	...	...	...	...	...	...	0	...							
21	×	$^{1}V_{22} \times {}^{5}V_{11}$	$^{0}V_{12}$	$\left\{ {}^{1}V_{22} = {}^{1}V_{22} \atop {}^{0}V_{12} = {}^{2}V_{12} \right\}$	$= B_3 \cdot \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{3} = B_3 A_3$	...	...	...	...	...	...	...	...	...	...	0	$B_3 A_3$	....	......	B_3		
22	+	$^{2}V_{12} + {}^{2}V_{13}$	$^{3}V_{13}$	$\left\{ {}^{2}V_{12} = {}^{0}V_{12} \atop {}^{2}V_{13} = {}^{3}V_{13} \right\}$	$= A_0 + B_1 A_1 + B_3 A_3$	...	...	...	...	...	...	...	...	...	...	....	0	$\left\{A_3 + B_1 A_1 + B_3 A_3\right\}$				
23	−	$^{2}V_{10} - {}^{1}V_1$	$^{3}V_{10}$	$\left\{ {}^{2}V_{10} = {}^{3}V_{10} \atop {}^{1}V_1 = {}^{1}V_1 \right\}$	$= n - 3\ (= 1)$	1	...	...	...	...	...	...	...	...	$n-3$							
					Here follows a repetition of Operations thirteen to twenty-three.																	
24	+	$^{4}V_{13} + {}^{0}V_{24}$	$^{1}V_{24}$	$\left\{ {}^{4}V_{13} = {}^{0}V_{13} \atop {}^{0}V_{24} = {}^{1}V_{24} \right\}$	$= B_7$	...	...	...	...	...	...	...	...	...	...	....	....	....	......	......	......	B_7
25	+	$^{1}V_1 + {}^{1}V_3$	$^{1}V_3$	$\left\{ {}^{1}V_1 = {}^{1}V_1 \atop {}^{1}V_3 = {}^{1}V_3 \right\}$; $^{5}V_6 = {}^{0}V_6$; $^{5}V_7 = {}^{0}V_7$	$= n + 1 = 4 + 1 = 5$; by a Variable-card.; by a Variable card.	1	...	$n+1$	...	...	0	0										

Augusta Ada King, Countess of Lovelace (née Byron, 1815–1852)



Ancient Programming

- Physical wiring (e.g. ENIAC, 1945)
- Machine code (von Neumann-architecture, 1945)
- Assembly (1949–)
- High-level programming languages
    - Plankalkül (Konrad Zuse, 1942–1945)
    - Fortran (John Backus et al., 1954)
    - LISP (John McCarthy, 1958)
    - Algol (1958, 1960, 1968)
    - COBOL (1959)
    - BASIC (Kemény–Kurtz, 1964)

Important Languages

- Simula-67 (Dahl–Nygaard, 1967)
- Pascal (Niklaus Wirth, 1970)
- C (Dennis Ritchie, 1972)
- Ada (1980)
- SQL (Chamberlin–Boyce, 1974)
- C++ (Bjarne Stroustrup, 1985)
- Eiffel (Bertrand Meyer, 1986)
- Erlang (Armstrong–Virding–Williams, 1986)
- Haskell (1990)
- Python (Guido van Rossum, 1990)
- Java (James Gosling, 1995)
- JavaScript (Brendan Eich, 1995)
- PHP (Rasmus Lerdorf, 1995)
- C# (2000)
- Scala (Martin Odersky, 2004)

Most Popular Languages (September 2018., TIOBE-index)

Sep 2018	Sep 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.436%	+4.75%
2	2		C	15.447%	+8.06%
3	5	↑	Python	7.653%	+4.67%
4	3	↓	C++	7.394%	+1.83%
5	8	↑	Visual Basic .NET	5.308%	+3.33%
6	4	↓	C#	3.295%	-1.48%
7	6	↓	PHP	2.775%	+0.57%
8	7	↓	JavaScript	2.131%	+0.11%
9	-	⇈	SQL	2.062%	+2.06%
10	18	⇈	Objective-C	1.509%	+0.00%
11	12	↑	Delphi/Object Pascal	1.292%	-0.49%
12	10	↓	Ruby	1.291%	-0.64%
13	16	↑	MATLAB	1.276%	-0.35%
14	15	↑	Assembly language	1.232%	-0.41%
15	13	↓	Swift	1.223%	-0.54%
16	17	↑	Go	1.081%	-0.49%
17	9	⇊	Perl	1.073%	-0.88%
18	11	⇊	R	1.016%	-0.80%

Origins of C

```
                                     +----> Objective-C
                                     |
        Fortran --------------+      |           +---> D
                              |      |           |
Assembly ----> BCPL ----> B ----> C -----> C++ ----> Java
                              |      |           |
       Algol60 ---------------|      |           +---> C#
                |             |      |
                +--> Algol68--+      +-------------------> Go
```

Evolution of C

- 1969 Ken Thompson develops B language (simplified BCPL)
- 1969 Ken Thompson, Dennis Ritchie and others start working on UNIX
- 1972 Dennis Ritchie develops the C language
- 1972-73 UNIX kernel is rewritten in C
- 1977 Johnson's Portable C Compiler
- 1978 Brian Kernighan and Dennis Ritchie: The C Programming Language book
- 1989 ANSI C standard (C90) (32 keywords)
- 1999 ANSI C99 standard (+5 keywords)
- 2011 ANSI C11 standard (+7 keywords)
- 2018 C18 ISO/IEC standard

We will use mostly ANSI C (C90).

Outline

Structure of Programs

- Expressions
- Statements
- Subprograms (functions/procedures, routines, methods)
- Modules (libraries, classes, packages)

C Example

```c
int factorial( int n )
{
    int result = 1;
    int i;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

Expressions

```
n
"Hello world!"
100
n+1
++i
range(2,n+1)
employees[factorial(3)].salary * 100
```

Statements

```
result = 1;
```

```
result *= i;
```

```
return result;
```

```
for( i=2; i<=n; ++i ){ result *= i; }
```

```
while(1) printf("Gyurrrika szép!\n");
```

Simple Statements

- Assignment
- Empty Statement
- Subprogram Call
- Return from Function

Control Structures

- Branches
- Loops etc.

```
int gcd( int n, int m )
{
    while( n != m )
        if( n > m )
            n -= m;
        else
            m -= n;
    return n;
}
```

Braces in Control Structures

Braces left out

```
int gcd( int n, int m )
{
    while( n != m )
        if( n > m )
            n -= m;
        else
            m -= n;
    return n;
}
```

Foolproof solution

```
int gcd( int n, int m )
{
    while( n != m ){
        if( n > m ){
            n -= m;
        } else {
            m -= n;
        }
    }
    return n;
}
```

Dangling Else

I wrote this

```
if( x > 0 )
   if( y != 0 )
      y = 0;
else
   x = 0;
```

It means this

```
if( x > 0 )
   if( y != 0 )
      y = 0;
   else
      x = 0;
```

I wanted this

```
if( x > 0 ){
   if( y != 0 )
      y = 0;
} else
   x = 0;
```

See...

goto-fail (Apple) link!

Writing to Standard Output

We write an integer and a newline.

```
printf("%d\n",factorial(10));
```

More Complex Output

```
printf("10! = %d, ln(10) = %f\n", factorial(10), log(10));
```

Types

- Express how to interpret a sequence of bits
- Determine what valid values a variable can have
- Tie operations to certain, valid values

In C

- `int` – interval of integers (whole numbers), e.g. $[(-1) * 2^{63} \; .. \; 2^{63} - 1]$
- `float` – subset of rational numbers
- `char` – characters in the extended ASCII character set
- `char[]` – array of characters, a text
- `int[]`– array of integers
- `int*` – pointer to an integer

etc.

The Role of Types

- Protection from programmer errors
- Express thoughts of the programmer
- Helps forming abstractions
- Helps compiling efficient code

Type checking

- Are variables, functions used according to their types?
- Programs with type incorrectness are invalid.

Static and Dynamic Typing

The C compiler checks type-correctness in *compile time*.

Strongly and Weakly Typed Languages

- In a weakly typed language, values are automatically converted to other types if necessary
  - Comfortable at the beginning
  - Easy to write something completely different unintentionally
- In C and Python, type conversion rules are rather strict (they are strongly typed)

Subprograms

- Describing a computation of multiple steps
- General, can get Parameters, Reusable
- Structuring programs – handling complexity
  - No longer than a page
- Has multiple names
  - routine or subroutine
  - function: computes a value and returns it
  - procedure: can change the program state
  - method: OOP terminology

Main Program

Where program execution starts.

C

Subprogram with the name: `main`

```
int main()
{
    int half = 21;
    printf("%d\n",2*half);
    return 0;                 /* successful execution */
}
```

Comments

```c
int main()
{
    int half = 21;
    printf("%d\n",2*half);
    return 0;                /* this is a comment */
}
```

Modules

Modularity: encapsulation, independence, narrow interfaces

- Reusable program libraries
    - e.g. standard library
- Bigger units of the program
- Making abstractions

Dividing into Modules

Reusable `factorial`

- `factorial.c` – factorial function
- `tenfactorial.c` – main program

tenfactorial.c

```
#include <stdio.h>

int factorial( int n );  /* declaring the function */

int main()
{
    printf("%d\n", factorial(10));
    return 0;
}
```

Outline

Source Code

- Code written in a programming language
- Computer: machine code
- Execution
    - interpretation (Python)
    - compilation, execution (C)
- Source file
    - `factorial.c`

Interpreter

- Executing the source code step-by-step, statement-after-statement
  - Invalid statement triggers an error
  - Valid statement is executed
- Execution of the statement: according to built-in machine code

Cons

- Runtime error on bad statement (rarely executed statement???)
- Slower running of the program

Pros

- Integration of program writing and execution
  - REPL = Read-Evaluate-Print-Loop
  - Making a prototype fast
- Easier for beginners

Source File in C

factorial.c

```c
#include <stdio.h>

int factorial( int n ){   /* you can also put { here */
    int result = 1;
    int i;
    for(i=2; i<=n; ++i){
        result *= i;
    }
    return result;
}

int main(){
    printf("%d\n", factorial(10));
    return 0;
}
```

Separation of Compilation and Execution

- Lots of programming errors can be detected without running the program
- Checking the program prior to everything else
- Only has to be done once (during *compilation*)
- Less errors during execution
- Goal: efficient and reliable machine code!

“Compilation time” and “Execution time”

Compilation

- source code in source file
  - `factorial.c`
- compiler
  - `gcc -c factorial.c`
- target code, object code
  - `factorial.o`

Compilation Unit

- Part of the source code (e.g. a module)
- Compiler gets one at a time
- Object code is produced from it

One program usually consists multiple compilation units.

In C

Content of a source file

Linking, Executable Code

- Objects (target code, object code)
  - `factorial.o` etc.
- Linker
  - `gcc -o factorial factorial.o`
- Executable code
  - `factorial`
  - default name: `a.out`

Multiple objects (linked together) make one executable.

Execution

```
./factorial
```

Multiple Compilation Units

factorial.c

```c
int factorial( int n )
{
    int result = 1, i;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

tenfactorial.c

```c
#include <stdio.h>

int factorial( int n );

int main()
{
    printf("%d\n", factorial(10));
    return 0;
}
```

Compilation, Linking, Execution

```
gcc -c factorial.c tenfactorial.c
gcc -o factorial factorial.o tenfactorial.o
./factorial
```

Two steps can be joint into one command

- source code in source files
  - `factorial.c` and `tenfactorial.c`
- compilation and linking in one step
  - `gcc -o factorial factorial.c tenfactorial.c`
- execution the binary
  - `factorial`

Compilation Errors

- Breach of language rules
- Detected by the compiler

factorial.c

```c
int factorial( int n )
{
    int result = 1;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

gcc -c factorial.c

```
factorial.c: In function ‘factorial’:
factorial.c:6:9: error: i undeclared (first use in this function)
     for(i=2; i<=n; ++i)
         ^
```

Linking Errors

factorial.c

```
int factorial( int n )
{
    int result = 1, i;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

tenfactorial.c

```
#include <stdio.h>

int faktorial( int n );

int main()
{
    printf("%d\n", faktorial(10));
    return 0;
}
```

Compilation, Linking, Error

```
$ gcc -c factorial.c tenfactorial.c
$ gcc -o factorial factorial.o tenfactorial.o
tenfactorial.o: In function `main':
tenfactorial.c:(.text+0xa): undefined reference to `faktorial'
collect2: error: ld returned 1 exit status
```

Compilation and Run Time Linkage

Static Linkage

- Before execution of the program
- Right after creating object code
- Pros: Linkage problems at compile time

Dynamic Linkage

- During execution of the program
- Dynamically linkable object code
  - Linux *shared object*: `.so`
  - Windows *dynamic-link library*: `.dll`
- Pros
  - Smaller executable size
  - Smaller memory usage (memory footprint)

Preprocessing

C preprocessor: produces source code from source code

Macros

```c
#define WIDTH 80
...
char line[WIDTH];
```

Sharing Declarations

```c
#include <stdio.h>
...
printf("Hello world!\n");
```

Conditional Compilation

```c
#ifdef FRENCH
printf("Salut!\n");
#else
printf("Hello!\n");
#endif
```