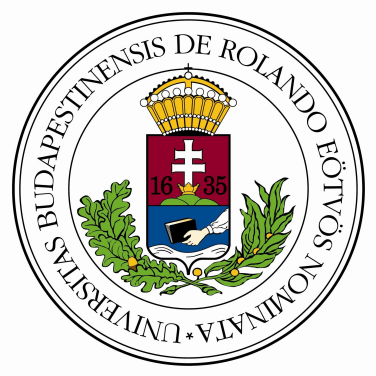
Imperative programming

Second Lecture



Kozsik Tamás, Porkoláb Zoltán

ELTE Eötvös Loránd Tudományegyetem

Outline

Repetition

- Source Code
  - Interpreting (Python)
  - Compiling, Running (C)
- Source File
- Interpreter: Pros and Cons
- Executable Shell-script

Source File in C

factorial.c

```c
#include <stdio.h>

int factorial( int n ){
    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

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

tenfactorial.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

```
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
```

Static and Dynamic 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

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

Sharing Declarations

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

Conditional Compilation

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

Programs in C

Compilation Time

- Source Files (`.c` and `.h`)
- Preprocessing
- Compilation Units
- Compilation
- Object Files
- Static Linking
- Executable File

Run Time

- Executable File, Object Files (in a dynamically linkable library)
- Dynamic Linking
- Running Program

Outline

Rules of a Programming Language

- Lexical
- Syntactical
- Semantic

Lexical Rules

What building blocks make up the language?

- Keywords: `while`, `for`, `if`, `else` stb.
- Operators: `+`, `*`, `++`, `?:` etc.
- Grouping signs (parentheses) and Separators
- Literals: `42`, `123.4`, `44.44e4`, `"Hello World!"` etc.
- Identifiers
- Comments

Case-(in)sensitive?

Literals

Whole Number:

- decimal form: `42`, `1_000_000` (Python)
- octal (C) and hexadecimal form: `0123`, `0xCAFE`
- unsigned representation (C): `34u`
- represented on more bits than int (C): `99L`
- and combined (C): `0xFEEL`

Literals

Floating Point Numbers:

- trivial: 3.141593
- with exponent: 31415.93E-4
- represented on more bits (C): 3.14159265358979L
- and combined (C): 31415.9265358979E-4L

Literals

Characters and Strings in C

- characters: `'a'`, `'9'`, `'$'`
- strings: `"a"`, `"almafa"`, `"1984"`
- escape-sequences: `'\n'`, `'\t'`, `'\r'`, `"\n"`, `"\r\n"`
- string of multiple parts: `"alma" "fa"`
- strings written in multiple lines:

```
"alma\
fa"
```

Identifiers

- Alphanumeric
- Do not start with a number!
- Can contain _ sign?

Good

- `factorial`, `i`
- `computePi`, `open_file`, `worth2see`, `Z00`
- `__main__`

Bad

- `2cents`
- `fifty%`
- nőnemű and $A\theta\acute{\eta}\nu\alpha$ (they work in Java)

Syntax Rules

How can we build?

- How is a loop or branch built up?
- How does a subprogram look like? etc.

Backus-Naur Form (Backus Normal Form) – BNF

```
<statement> ::= <expression-statement>
              | <while-statement>
              | <if-statement>
              | ...

<while-statement> ::=  while (<expression>) <statement>

<if-statement> ::=  if (<expression>) <statement>
                    <optional-else-part>

<optional-else-part> ::= ""
                       | else <statement>
```

Semantic Rules

Does it make sense what we built?

- Were the used variables declared? (C)
- Was the operation called with the right type of arguments?

etc.

Role of a Type

- Protects from programmer errors
- Expresses the thoughts of the programmer
- Helps forming abstractions
- Helps generating efficient code

Type-checking

- Did we use variables and functions according to their types?
- Programs which are not type-correct do not make sense.

Static and Dynamic Type-system

- C compiler checks type-correctness in *compile time*
- Some languages check this at *run time* using its interpreter

Strongly and Weakly Typed Languages

- In a weakly typed language, values are automatically converted if its needed
  - Convenient at first
  - Easy to write something different than expected
- C and Python has rather strict rules (they are strongly typed)

Static and Dynamic Semantic Rules

- Static: checkable by the compiler
- Dynamic: checkable only at runtime

Decision problem...

Summary

- Lexical: What are the building blocks?
- Syntactic: How do we build structures?
- Semantic: Does the built up structure make sense?
  - Static Semantic Rules
  - Dynamic Semantic Rules

Parts of a Compiler

- Lexer: Sequence of Tokens
- Parser: Syntax Tree, Symbol Table
- Semantic (e.g. type-) Checking

(or compilation errors from different levels)

Formal Languages

- Lexical rules: Regular Grammar
- Syntax rules: Context-free Grammar
- Semantic rules: Context-sensitive Grammar or Unrestricted Grammar

Semantics of a Program

The meaning of a program according to the rules of the language

Definition of a Language

- Lexica
- Syntax
- Semantics
- Pragmatics

Pragmatics

How can we express ourselves efficiently?

- Conventions / Standards
- Idioms
- Good and Bad practices

etc.

Conventions / Standards

General or Specific to a certain group (company)

- placement of braces
- naming (e.g. setter/getter)
- identifier names, language
- lower- and uppercase letters