Programming II

Martin Pergel, perm@kam.mff.cuni.cz

9. března 2014

Overhead information

- Programming II extends Programming I,
- Language-change – C# instead of Pascal,
- C# gets educated further by Pavel Ježek (2nd year),
- Technical programming, algorithms, data structures.

Obligations Credit + Exam

- Credit-requirements:
  1. Active participants at the assignments (as in winter-term),
  2. Practical test (write and debug specified program within 90 minutes not in Pascal),
  3. Semester project (as in winter term).
- Exam – normal exam without computers, Within 2 or 3 hours design a program that cannot be written within 3 hours – focus on principal problems.

Towards C#

- Bad news:
  - Learning a new language,
  - Language essentially uses object,
  - C# is relatively complicated language,
  - Error-messages not always make a good sense to you (complicated).
- Good news:
  - Leaving 20 years old environment,
  - There exist multiplatform implementation for C#,
  - You may use it for fulfilling pending obligations from winter term.

Step aside – object programming

- World consists of objects (pen, black-board, lecturer).
- Objects are mutually interacting (lecture lectures the students),
- objects are living their own life,
- there may be many objects of a particular type (say, of type student).

Objects and classes

- Objects are individual items around us.
- Each item has an underlying data-type $\Rightarrow$ class.
- Class reflects a collection of objects of the same type [e.g., student].
- We may view it as the underlying data-type [`student`] .
- Also possible: Platon had a world of ideals (where ideal instance of each object can be found). Class refers to this world of ideals (e.g., ideal student).

Technical point of view

- An object (e.g., car) has some properties (attributes), say, color, brand, weight, sizes,...
- Thus we may observe it as something similar to structures (records from Pascal).
- But also objects have some abilities called **methods.** E.g., car $\Rightarrow$ start up, stop the motor,...
- ... so objects can be imagines as piles of attributes (variables) and methods (functions).
- Object programming gives us much more (inheritance, encapsulation, polymorphism), but we have to go step by step (these three paradigms take place in a few weeks).

Towards C# I/III
general ideas of syntax

- C# is a language from the family of C-programming language,
- C should be an obfuscated Pascal, thus some ideas hold, some were changed.
- Both language are structures, so program consist of blocks (where we are using control-structures, define and call functions, functions are returning values and yet we may work with variables).
- Languages from family of C-language are case sensitive!
- Blocks are not delimited by `begin` and `end` but by braces ({ and }).

Towards C# II/III

general ideas of syntax

- Variable-definition differs, instead of `var a:integer;` we write `int a;`, i.e., type-name preceeds the variable name, keyword `var` is omitted. Types are slightly renamed (`char`, `int`, `long`, `double...`)
- Function-definition differs: Same modification as for vars, omit keyword `procedure/function`, put return-type instead: `function f(a:integer):longint;` ⇒ `long f(int a)` **note the missing semicolon!**

Towards C# III/III

general ideas of syntax

- Procedures are functions that return no value ⇒ data-type `void`:
  `procedure f(a:integer);` ⇒ `void f(int a)`
- When defining/calling a function, brackets are compulsory:
  `function f:integer;` ⇒ `int f()`
  otherwise we cannot simply distinguish variable- and function-definition!
  `x:=f;` ⇒ `x= f();`
  not always we want to call the function, maybe we want to create a reference on it.

Basic operators

- $+, -, *$ – similar to Pascal,
- $/$ – also similar, but adaptive. For ints integral division, for at least one non-integer, non-integral division.
- % – modulus (`mod`),
- logical: &&, ||, ^, ! – `and, or, xor, not`,
- relational: $<, >, <=, >=$ (the same as in Pascal), but...
- $==, =$ – comparison for equality (Pascal =) and assignment (Pascal `:=`).
- Brackets are the same (priority-forcing). Beware of priorities!

Basic control-structures

- `if(condition) expression_or_block`,
- `while(condition) expression_or_block`,
- `if` and `while` statements are very similar to Pascal,
- difference: Statements are not separated but ended with semicolon, thus:
- `if(condition) statement; else other_statement`
- Beware: `if(condition) block else statement_or_block` – block begins with { and ends with }. After } and before `else` the semicolon must not be placed!

For-cycle

- For-cycle is generalized:
  `for(init; condition; increment) body`
- `init` is the initial expression (can be used, e.g., as in Pascal),
- `condition` is a condition as in `while`-cycle, `for`-cycle is being iterated while this condition holds.
- `increment` is an incremental expression – it gets performed after each iteration (after we perform the body).
- **There is no automatic incrementation for the cycling variable!** Conversely, there are no specific restrictions on modifying this variable.
- Any part of the for-cycle can be empty: `for(;;);` (empty init, cond, inc and body – this cycle never ends).

Functions return a value...

... in a different way than in Pascal

- There is no special variable (with the same name as the function as in Pascal).
- The aim of the function is to return the value, thus when the value is clear, there is not reason to continue interpretting the function.
- Return-value is operated using keyword `return` followed by the expression we want to return (in procedure, i.e., function returning `void` this expression is empty).
- Example: `return 1`; or `return "something"`;.
- Note that strings are not inside apostrophes but inside quotation-marks!
- Strings shall be educated further in semester due to particular issues...

Remarkable operators $++$ and $--$

pars prima

- Often we want to increment or decrement a value of a variable.
- In Pascal ... functions `inc` and `dec`.
- Since C ... unary operators $++$ and $--$.
- They either preceede or succeed a variable:
- Example $++a$ – means increment value of a by one,
- $--a$ similarly decrement. These operators return a value. Their value is the new value of the given variable, i.e., example:
- `int a,b; a=1;b=++a;` – now, both `b` and `a` have value 2.

Remarkable operators ++ and −−

pars secunda

- Usually we want first use the (original) value and then modify it.
- In such a situation we use postfix versions:
  `a=1; b=a++;` – now, `a` is two but `b` is 1.
- These operators incur a side-effect. A variable with a side-effect-operator must not appear more than once in an expression and it is prohibited to perform more than one side-effect (for each variable) in one expression:
- `b=a++++;` is not well-defined (two side-effects on `a`),
- `b= a++ + ++c;` is well-defined,
- `b= a++ + a;` not well-defined (`a` with side-effect appears twice).

Assignment expression VS statement

- In Pascal – assignment statement: `a:=b;`
- In languages of C-family – assignment expression, the value of the assignment expression is the assigned value, thus we may perform: `a=b=c=d=1;`
- Moreover, since C we may initialize variables when defining them:
- `int a=1,b=2,c=3;`

Example

function computing factorial

```
long factorial(int a)
{    int b=1,c;
     for(c=1;c<=a;c++)
         b=b*c;
     return b;
}
```

Another example

again a function computing factorial

```
long factorial(int a)
{    int b=1,c;
     for(c=1;c<=a;c++)
         b*=c; //operator multiply-by
     return b;
}
```

Yet another example

again a function computing factorial

```
long factorial(int a)
{    int b,c;
     for(b=c=1;c<=a;b*=c++);
     //for-cycle with empty body
     return b;
}
```

And yet another example

again a function computing factorial

```
long factorial(int a)
{    int b,c;
     for(b=c=1;c<=a;)b*=c++;
     //for-cycle with empty incremental expr.
     return b;
}
```

Comments

in the program – not those ones to the lecture

- We cannot use symbols { and } as in Pascal for comments,
- we use the sequences /∗ and ∗/, respectively, instead.
- Example: `/* Here is a comment... */`
- Many comments are one-line only, thus:
- yet another possibility of comments – starting with //
- This comment ends with the end of line:
- Example:

```
int f() // a function named f returning int
```

Impression

Languages from family of C-programming language are designed in an efficient way (unnecessary keywords are removed) but the program looks more impersonal (than in Pascal).

Specifics

that made us avoid C# in the winter term...

- There are yet several specifics:
- Instead of stray code between `begin` and `end` (as in Pascal), the main program is represented by function `Main`.
- And that's not all, we have to perform the ritual whose meaning we'll try to explain before the end of the lecture as much as possible (and we'll continue next lecture).

Ritual

that must be performed before we start writing C# program

```
namespace x{
     class y{
          public static void Main()
          {    // here the main program takes place
          }
     }
}
```

Today we skip modifiers `public` and `static`.

Analyzing the ritual I/II

- Probably you recognized the definition of function `Main` which is the entry-point of the program.
- Keyword `class` define a class the sense we defined it while talking about object programming, i.e.,
- in object-oriented languages, everything is either a class or an object, thus even a program must be defined as a class (as the objects are instances of the class [i.e., variables of this data-type]).
- Keyword `class` is followed by the name of the class we are currently defining; the definition of the class is inside a block (surrounded by braces).
- Thus even the function `Main` must be surrounded by some object.

Analyzing the ritual II/II

- The name of the "surrounding" object is not prescribed, we may name it almost as we want.
- Keyword `namespace` introduces so called namespace. It is a pile of classes - at the moment we may imagine it as something similar to class (just it has a bit different abilities, as we'll see later).
- In C#, all classes must be defined in some namespace, thus we have to surround the name of the class by some namespace.
- Syntactically, we define namespaces similarly to classes, so the keyword `namespace` is followed by the name of the namespace and the definition is inside a block (inside braces).
- As it was for the classes, even the name of the namespace is not prescribed, thus we may name it arbitrarily.

First program

doing something...

```
namespace x{
     class y{
          public static void Main()
          {    System.Console.WriteLine("This is a
program doing something...");
               for(;;);
          }
     }
}
```

Remarks

to the program

- Dot (binary operator) works in a similar way as in Pascal, now it is the operator of accessing the class, object or a namespace.
- There is a namespace `System` containing static class `Console`.
- Class `Console` is equipped by a method `WriteLine` doing something similar to what you know from Pascal.

Yet one ritual
related to the program

- When defining functions, we are defining them in the same class (as method `Main`) and we have to define them as `static`, too (until we start understanding what `static` really means.
- If we want to define something like global variables, we define them also in the class and also as `static`:
  `static int a,b,c;`.
- We do not have to define them asi `public` (but we do not lose anything when doing so).
- **Beware:** Local variables are not defined as static (or public).

Short practical presentation

of Visual Studio .NET 2013 Express

... and that will be all for today.

Thank you for your attention.