Wokół języka C
Krokodyl jest zupełnie inny...
część 3 – znów te straszne paradygmaty

Wielu elektroników chce programować w C bez wgłębiania się w szczegóły języka. Chętnie wykorzystują oni wszelkie „gotowce”. Ich działania w rzeczywistości polegają na modyfikacji programów znalezionych gdzieś w Internecie. A gdy coś nie działa, proszą o pomoc na forach. Jeśli i Ty masz takie podejście, NIE CZYTAJ tego artykułu! Jeżeli jednak należysz do tych, którzy chcą jak najlepiej rozumieć to, co robią – podejmij trud zrozumienia przedstawionych dalej, zupełnie obcych Ci, na razie, zagadnień.

Tytuł artykułu pochodzi z końcówki przypomnianego w pierwszej części artykułu starego dowcipu, w którym Władek stara się wytłumaczyć Staśkowi, co to jest i jak wygląda krokodyl:

– A wiesz, jak wygląda koń?

– A jakże.

– No to krokodyl jest zupełnie inny!

Ucząc się, próbujemy dopasować „to nowe, nieznane” do tego, co już wiemy i znamy. Ale w przypadku języka C mamy ten sam problem co Władek i Stasiek: w programie nie dostrzegamy podobieństwa do tego, co już znamy. Wszystko dlatego, że...

krokodyl jest zupełnie inny!

W poprzednich odcinkach omówiliśmy podstawowe „cegiełki” języka C. Na koniec wspomnieliśmy jednak o tym, że elegancki i proste analogie mogą wprowadzić w błąd.

Tajemnicze skrzynki – z korbą czy bez?

Piękno i prostota podstawowej koncepcji języka C polega między innymi na tym, że elementy programu porównaliśmy do skrzynek (niekoniecznie czarnych), do których coś wkładamy (argumenty – liczby) i które *zwracają wartość*. Używamy określenia „zwracają” (od angielskiego *return*), ale nie jest to „zwracanie” na zasadzie oddawania pożyczki. Skrzynka przeprowadza jakieś operacje (akcje) i po prostu **daje** jakiś wynik – jakąś wartość.

Możemy przyjąć, że „największa skrzynka” to cały program, „średnie skrzynki” to *funkcje*, a „małe” skrzynki to *wyrażenia*. Za najmniejsze, ale w pewnym sensie „niekompletne skrzynki” moglibyśmy też uznać nawet pojedyncze *operatory*. Jednak raczej uznajmy, że w języku C „najmniejszymi skrzynkami” są proste *wyrażenia*, które jak wiesz, składają się z operatorów i argumentów.

„Skrzynkowa” idea programu w języku C jest elegancka i prosta, ale nasuwają się tu wątpliwości. Otóż jeśli za *wyrażeniem* postawimy znak **;** (średnik), to *wyrażenie* przekształcamy w *instrukcję*. Co to tak naprawdę znaczy? Jak to przystępnie wytłumaczyć?

Niestety, *krokodyl jest zupełnie inny* – nie można odpowiedzieć jednym zdaniem. Gdybyśmy kurczowo trzymali się analogii „skrzynkowej”, można byłoby próbować to zobrazować obecnością korby w skrzynce. Może i Tobie nasunęło się pytanie, czy omawiane skrzynki mają korbę i czy wynik „wyskakuje” dopiero po „pokręceniu korbą”? Czy może po podaniu „danych wejściowych” dają (zwracają) wynik ot tak, same z siebie?

W innych dziedzinach wydaje się naturalne, że aby uzyskać efekt, trzeba „pokręcić korbą”, ale w przypadku programowania jest to bardziej skomplikowane, ponieważ mamy do czynienia z dwoma aspektami sprawy. Jeden aspekt sprawy to praktyczne działanie procesora, a drugi to zagadnienia matematyczne, a więc abstrakcyjne, praktycznie niezwiązane z procesorem i jego działaniem. „Programiści komputerowi” piszący w tak zwanych „czystych językach funkcyjnych” mogą zajmować się programem źródłowym i jego sensem matematycznym, zupełnie pomijając procesor. W języku C też po części mamy taką sytuację, bo w programach mamy różne *wyrażenia*, realizujące operacje matematyczne. Tu powinniśmy zwracać uwagę na aspekt matematyczny, a nie na praktyczne działanie procesora, więc analogią *wyrażeń* są „skrzynki bez korby”.

Wyrażenie zawsze zwraca wartość, ale w abstrakcyjnym sensie matematycznym, a nie praktycznym. Choćby Cię korciło, lepiej nie pytaj, *na ile wartości zwracane przez wyrażenia realnie istnieją* w pamięci lub w procesorze. Bo gdybyśmy chcieli to zbadać, dojdziemy do jeszcze trudniejszych pojęć, do wartości „lewych i prawych” (*L-wartości* i *R-wartości*, *lvalue*, *rvalue*), a tego chciałbym Ci oszczędzić, przynajmniej na razie...

Ale niestety, musimy omówić kwestie obecności i pomieszania w programie C abstrakcyjnych zagadnień czysto matematycznych z zagadnieniami czysto praktycznymi. A jeśli mamy poznać tego krokodyla, znów muszę wrócić do (mam nadzieję już nie)strasznych paradygmatów.

Znów o paradygmatach

W niewielkim uproszczeniu *paradygmat* to *wzorzec*, *przykład*, *model*, *ogólnie przyjęty sposób*. Otóż większość języków programowania, a w szczególności język maszynowy i asembler, są zgodne z paradygmatem ***programowania imperatywnego***, gdzie *mamy do czynienia z* ***sekwencją rozkazów – instrukcji, zmieniających stan programu***. A ten *stan programu* to zawartość zmiennych (czyli w sumie pamięci RAM). Cechą charakterystyczną programowania imperatywnego są po pierwsze *zmienne*, po drugie *instrukcje – rozkazy*, a w szczególności *instrukcja przypisania*, zawierająca symbol = operator przypisania. Właśnie *instrukcja przypisania ustala i modyfikuje stan zmiennych*. Można też powiedzieć, że obliczenia imperatywne to wykonanie określonej sekwencji *przypisań*. Inną ważną sprawą jest tu *kolejność* realizacji instrukcji, czyli w sumie sprawa *czasu* i *kolejności zdarzeń w czasie*. Zapis programów odzwierciedla przede wszystkim

działanie i *zmiany w czasie*. Dlatego w imperatywnych językach programowania mamy realizowane w czasie *sekwencje*, *instrukcje wyboru* i *pętle*.

Język maszynowy oraz asembler są klasycznymi, „czystymi" przykładami programowania imperatywnego: programy składają się z kolejno wykonywanych rozkazów – instrukcji. Rozwinięciem prostego programowania imperatywnego są: programowanie *proceduralne* i *strukturalne*, gdzie kładzie się nacisk na podział programu na niezależne części (procedury – funkcje). Dalszym rozwinięciem jest *programowanie obiektowe*, gdzie program to zbiór jeszcze większych autonomicznych bytów, zwanych *obiektami, które są połączeniem danych i funkcji-metod*. Są to wszystko jakby ulepszone postacie programowania imperatywnego – rozkazowego.

A na przeciwległym biegunie leży paradygmat *programowania deklaratywnego*, w tym ***programowania funkcyjnego*** (*funkcjonalnego*). Tu kładzie się nacisk nie na to ***jak***, tylko ***co*** ma być wykonane. Mówiąc w największym skrócie, program funkcyjny to nie opis „kroków działania", a jedynie opis matematycznych zależności, które z danych wejściowych pozwolą uzyskać wynik.

I tu musimy nawiązać do matematycznego pojęcia funkcji. W matematyce **funkcję** zasadniczo definiuje się jako *przyporządkowanie każdemu elementowi ze zbioru X dokładnie jednego elementu ze zbioru Y*. A więc *przyporządkowanie* to inaczej ustalenie *relacji*, czyli *zależności*. Funkcję można też rozumieć, i w informatyce tak ją rozumiemy: jako abstrakcyjny twór, do którego „coś wchodzi i coś wychodzi". I właśnie *w matematyce funkcja to zależność, przepis, relacja między tym „co wchodzi" a tym „co wychodzi"*. Ta zależność (przepis) może być i zwykle jest zapisana wzorem matematycznym (zwykle jednak w sposób inny, niż na szkolnych lekcjach matematyki). Możemy powiedzieć, że ***w matematyce funkcja to abstrakcyjny twór, który pobiera pewną liczbę argumentów i zwraca wynik***.

Można w uproszczeniu powiedzieć, że *programowanie funkcyjne to czysta matematyka*. Program funkcyjny przedstawia tylko matematyczną zależność pomiędzy danymi wejściowymi i wyjściowymi. Język „funkcyjny" to przede wszystkim Haskell, po części także LISP, ML, Ocaml, F# (w sumie jak na razie bardzo mało popularne w porównaniu z imperatywnymi, takimi jak choćby C, C++ i pokrewne). Wypadałoby też nadmienić, że „matematyczne programy funkcyjne" też ostatecznie realizowane są przez procesory krok po kroku, czyli „imperatywnie", ale wcześniej, w programie, tej „imperatywności nie widać", bo program funkcyjny to w sumie opis funkcji matematycznych. Tę „czystą matematykę" programu kompilator albo interpreter zamieni na sekwencję rozkazów dla procesora, ale programista, pisząc kod, zupełnie nie zawraca sobie głowy, jak to będzie zrealizowane – skupia się tylko na zależnościach matematycznych i ich zapisie.

Co bardzo ważne, *w „czystym" programie funkcyjnym nie ma instrukcji*, nie widać *przypisań* ani *sekwencji* – kolejności, nie mówi się o czasie ani o zmianie stanu programu. Co jeszcze ciekawsze, w czystym języku funkcyjnym wartości zmiennych są ustalone *przed* wykonaniem programu i *zmienne wbrew nazwie nie zmieniają wartości podczas wykonywania programu*! Program w „czystym języku funkcyjnym" jest więc matematycznym zapisem koncepcji mniejszych i większych skrzynek, do których przekazujemy argumenty i które zwracają wartości, które stają się argumentami dla innych skrzynek. Na etapie pisania programu „funkcyjnego" nie jest ważne, w jaki sposób zwracane wartości są przekazywane dalej. Po prostu są przekazywane. Interesujące jest tylko to, co „wchodzi do programu", jakie są zależności matematyczne, i co „z programu wychodzi". I nie zwraca się uwagi na żadne „stany pośrednie" czy „wyniki pośrednie". Ale tak może być tylko w „czystych językach funkcyjnych".

W każdym razie w informatyce z jednej strony mamy „czysto matematyczne programy funkcyjne", gdzie mamy zapisane tylko idee funkcji, do których coś przekazujemy i które zwracają wartości (wyniki), co może kojarzyć się z naszymi „skrzynkami bez korby", a także ze znanymi z matematyki grafami. Z drugiej strony mamy „czysto imperatywny asembler", czyli ciąg konkretnych rozkazów operujących na konkretnych rejestrach i innych komórkach pamięci, co słusznie może się kojarzyć z „pokręcaniem korbą".

Język C trzeba umieścić gdzieś pomiędzy. W języku C niewątpliwie mamy elegancką, matematyczną koncepcję „skrzynek", które pobierają argumenty i zwracają wartość – to przede wszystkim ***wyrażenia***. Ale C to jednak „język imperatywny", więc mówimy o zmianach stanu programu, czyli zmianach zawartości zmiennych (i rejestrów) w czasie. A to jest wynikiem realizacji ***instrukcji***. Co jeszcze dziwniejsze: *głównym celem większości programów, zwłaszcza dla mikrokontekstów, wcale* ***nie są wyniki operacji matematycznych****, realizowanych przez „skrzynki"*. Zazwyczaj ***głównym celem są efekty uboczne***: odpowiednie zmiany stanów pamięci (rejestrów), co w mikrokontrolerach oznacza wykorzystanie różnych urządzeń peryferyjnych.

Elektronikowi łatwiej zrozumieć zagadnienia asemblera, odzwierciedlające 1:1 działanie procesora. Trudniej zrozumieć matematyczno-abstrakcyjne aspekty zapisu w języku C. Tym bardziej że te „matematyczne" zapisy w programach C nie są (a przynajmniej nie muszą być) zamieniane na „konkretne działanie" procesora w łatwo zrozumiały sposób. Elektronikom przeszkadza w nauce języka to, że w programie w C mamy połączone, a wręcz pomieszane dwie sprawy: kwestie czysto *matematyczne* oraz kwestie ich *praktycznej realizacji* przez mikroprocesor.

Nie ma rady: trzeba to zrozumieć i przyzwyczaić się do takiego podwójnego podejścia. Między innymi trzeba poznać matematyczne *wyrażenia* i praktyczne *instrukcje*. Aby to zrozumieć, musimy dokładniej omówić ***zwracanie wartości*** oraz to, co się z tą zwracaną wartością dzieje.

Piotr Górecki