# Wokół języka C
# Krokodyl jest zupełnie inny...
## część 2 – piękne (i mylące) analogie

**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!**

A jeśli nawet dostrzeżemy podobieństwa i analogie, to może się okazać, że są one mylące i zamiast pomagać, będą nas wprowadzić w błąd! O tym też mówi ten artykuł. Ale na początek spróbujmy nakreślić ogólny obraz języka C.

## „Cegiełki" języka C

Wcześniej mówiliśmy o większych i mniejszych skrzynkach, do których coś wkładamy (*argumenty*) i które dają coś na wyjściu (*zwracają wartość*). Możemy to zilustrować za pomocą **rysunku 4**. Taka prosta analogia „skrzynkowa" ilustruje podstawową koncepcję języka C i jego „składników", ale niestety może wprowadzić w błąd, ponieważ... *krokodyl jest zupełnie inny*. W życiu codziennym nie mamy analogii, które ściśle odpowiadałyby wszystkim abstrakcyjnym aspektom języka C. Dlatego musimy odbyć wycieczkę do krainy abstrakcji.

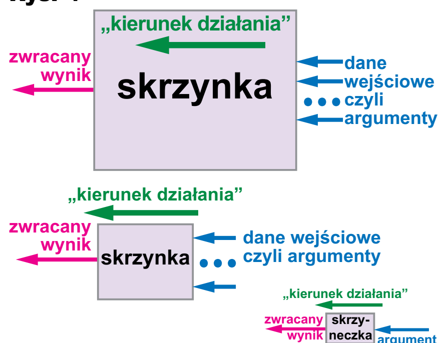
Nie będziemy jednak szczegółowo analizować *jednostek leksykalnych* (*leksemów*), takich jak *komentarze*, *słowa kluczowe* danego języka, *identyfikatory* (np. nazwy zmiennych, nazwy funkcji), *literały* (np. ustalone wartości liczbowe zapisane w programie), *separatory* (np. spacje, nawiasy).

Na wpół intuicyjnie powiedzmy tak: *istotą programu jest **przetwarzanie liczb***. A właśnie ***w języku C mamy do czynienia wyłącznie z liczbami***. Co bardzo ważne, także napisy są tu liczbami, podobnie jak wartości logiczne (prawda/fałsz). Nawet porty procesora i wszelkie inne peryferie w mikrokontrolerach obsługuje się z wykorzystaniem liczb, bo zbiory niezależnych bitów też można traktować jako liczby dwójkowe. Na marginesie: to również jest zaleta i element piękna języka C.

Mamy więc ***liczby***, które ogólnie nazywamy ***operandami*** (zawarte np. w zmiennych, stałych i w literałach). Natomiast ich różnorodne ***przetwarzanie*** jest opisane w programie za pomocą różnych „dziwnych znaczków", które nazywamy ***operatorami***. Jednak częściej mówimy, że ***operatory*** działają na ***argumentach*** (na konkretnych wartościach).

**Operatory**. Krótka definicja z Wikipedii (*Operator – konstrukcja językowa jedno- bądź wieloargumentowa zwracająca wartość*) może przestraszyć początkujących, niemniej wskazuje na zasygnalizowane na rysunku 4 bardzo ważne zagadnienie *zwracania wartości*, które trzeba dobrze rozumieć.

**Rys. 4**




Na pewno nie powinieneś bać się określenia ***operator***! Sedno sprawy jest proste: ***operator to symbol, znak (lub kilka znaków), który określa jakąś operację***, *akcję*. Ogólnie biorąc, *operację matematyczną*, np. *arytmetyczną*, *logiczną*, *bitową*, *porównania*, ale też niezmiernie ważną operację *przypisania*.

W przykładzie `a = 5` zmiennej **a *przypisujemy*** wartość **5**. Natomiast w przypisaniu `b = c` zmiennej **b** przypisujemy wartość zmiennej **c**, inaczej mówiąc, do zmiennej **b** wpisujemy wartość, jaka wcześniej była w zmiennej **c**.

Uwaga! Wbrew pozorom, zapis ten ***nie oznacza porównania*** wartości zmiennych, tylko właśnie jest **przypisaniem**. W języku C *znak = jest operatorem przypisania*, a nie operatorem porównania.

Innym prostym przykładem jest arytmetyczny *operator dodawania*, który ma postać znaku + (plus). W jak najbardziej sensownym przykładzie: `b = b + 1` mamy dwa operatory: najpierw arytmetyczny *operator dodawania* dodaje liczbę 1 do zawartości zmiennej **b**, a *operator przypisania* wpisuje wynik tej operacji do zmiennej **b**. Po prostu następuje zwiększenie zawartości zmiennej **b** o jeden.

Do operatorów będziemy jeszcze wracać. Na razie nie chcę Cię straszyć ani *priorytetem i łącznością operatorów*, ani tak przerażającymi początkujących skrótami (wspomnę tylko, że w języku C odpowiednikami zapisu `b = b + 1` są postacie skrócone `b += 1` oraz jeszcze krótsze: `++b` i poniekąd `b++`).

Wspomnę natomiast o czymś znacznie łatwiejszym: jeśli mówimy, że *operatory działają na argumentach*, to oczywiste, że arytmetyczny *operator dodawania* + jest ***dwuargumentowy***,

bo dodajemy dwie liczby. Z kolei logiczny *operator negacji* (mający postać wykrzyknika **!**) jest oczywiście ***jednoargumentowy***. W języku C istnieje też jeden operator *trójargumentowy*. A odnośnie do utrudniania nauki wspomnijmy o różnicach w terminologii: otóż „dla ułatwienia", inne źródła mówią o operatorach *dwuoperandowych*, *trójoperandowych* oraz o *arności* operatorów (*unarny*, *binarny*, *ternarny*).

Poszczególne operatory moglibyśmy uznać za najmniejsze „skrzynki", do których coś wchodzi (jeden, dwa lub trzy argumenty) i które dają jakiś wynik działania (zwracają wartość). Jednak w praktyce za najmniejsze „kompletne i samodzielne skrzynki" tworzące program należy raczej uznać...

**Wyrażenia**. Podstawowe zasady są proste: w programie mamy liczne ***operatory*** *– symbole opisujące pewne działania* oraz mamy ***argumenty*** (***operandy***), które finalnie są (będą) liczbami. Jeżeli zgodnie ze składnią, czyli z zasadami języka, połączymy *operatory* i *argumenty*, to otrzymamy **wyrażenie** (ang. ***expression***). Wyrażenia mogą być bardzo proste, na przykład:

```
b=3
a>1
!1
```

albo bardziej złożone:

```
a<b&&b>c
!(a||b>c)
a<b*2&&c+1>d-2
a*=a+2
```

Nie przejmuj się, jeśli „ani w ząb" nie rozumiesz ich sensu (celowo zapisałem je bez spacji, żeby bardziej „straszyły"). Ogólnie biorąc, ***wyrażenie opisuje, co operator (operatory) ma zrobić z argumentami***.

***Wyrażenie*** jest „pełnowartościową skrzynką", która zawiera *operatory*, pobiera *argumenty* i co bardzo ważne ***zwraca wartość***. Do zwracania wartości wrócimy. A na razie zauważ, że *wyrażenie* to zapis w programie, konstrukcja abstrakcyjna, dla której nie powinniśmy od razu szukać jednoznacznego utożsamienia z konkretnym działaniem procesora.

Wyrażenia wykorzystujemy w programie w różny sposób. Można zaryzykować stwierdzenie, że w programie samo *wyrażenie* jest jedynie „półproduktem". „Półproduktem", który może być i często jest wykorzystany jako część czegoś większego. I tak na przykład *wyrażenia* mogą być częścią *instrukcji selekcji* i *pętli*, gdzie służą m.in. do badania najróżniejszych warunków. Wyrażenie można też „usamodzielnić"...

**Instrukcje**. A oto bardzo ważny szczegół: *wyrażenie* stanie się „pełnowartościowym samodzielnym produktem" wtedy, gdy postawimy za nim znak **;** (średnik), bowiem w języku C ***wyrażenie*** *zakończone średnikiem to* ***instrukcja*** (w tym wypadku tzw. *instrukcja wyrażeniowa*). W języku C średnik nie jest znakiem interpunkcyjnym, tylko tzw. ***terminatorem instrukcji***. Wskazuje, gdzie kończy się jedna instrukcja, a zaczyna inna. Bez średników kompilator nie potrafiłby prawidłowo zinterpretować i zrealizować programu.

Czy wobec tego ***instrukcja*** (ang. ***statement***) jest najmniejszym „samodzielnym" elementem programu?

Odpowiedź brzmi: i tak, i nie, ponieważ... *krokodyl jest zupełnie inny*!

Wcześniej mówiliśmy o „najmniejszych skrzynkach": *operatorach* i *argumentach*, które wchodzą w skład *wyrażeń*. Wszystko to są konstrukcje matematyczne. Natomiast wprowadzając pojęcie instrukcji, zaczynamy mówić o zupełnie innym aspekcie zagadnienia. Spróbuję Ci to przybliżyć w dalszej części artykułu, a na razie pozostańmy przy instrukcjach.

Może się dziwisz, że najkrótszą *instrukcją* w języku C jest...

```
;
```

sam średnik, czyli *instrukcja pusta*. „Trochę większe" są *instrukcje proste*, składające się z jednego, krótkiego *wyrażenia*, zakończonego średnikiem, na przykład:

```
b=3;
a=b+1;
a*=a+2;
```

„Nieco większe" są instrukcje sterujące. I tak na przykład instrukcja `if`:

```
if (a>9) a=0;
```

zawiera w sobie dwa prościutkie wyrażenia, w tym przypadku: `a>9` oraz `a=0`.

W języku C w jednej linii może być umieszczonych wiele instrukcji, każda ze średnikiem:

```
b=3; a=b+1; a*=a+2;
```

jednak dla przejrzystości kodu nawet krótkie instrukcje raczej pisze się w oddzielnych liniach.

**Bloki**. Często jednak szereg instrukcji realizuje jakieś określone zadanie i wszystkie je można traktować jako jedną ***instrukcję złożoną***. W szczególności bardzo często wykorzystuje się ***instrukcje blokowe***, zwane krótko ***blokami*** (o których już wspominaliśmy). Instrukcja blokowa, umieszczona w nawiasach klamrowych { }, może zawierać w sobie wiele instrukcji prostych. Znaki { } nie są jedynie wskazówką dla człowieka, że dany fragment programu stanowi pewną całość. Zapamiętaj ważny szczegół:

***zmienne*** *(ogólnie: identyfikatory) zdefiniowane w* ***bloku { } są zmiennymi lokalnymi*** *widocznymi tylko w tym bloku, istniejącymi tylko przez czas realizacji tego bloku.*

**Funkcje**. A jeżeli mamy blok, to jak już wiesz, można go „usamodzielnić", tworząc z niego **funkcję** przez nadanie nazwy (*identyfikatora*) i określenie *typu* zwracanej wartości oraz typów *argumentów* do funkcji przekazywanych.

## Piękne (i mylące) analogie

Program w języku C porównaliśmy do zestawu większych i mniejszych skrzynek, które przyjmują argumenty i zwracają wartość. Jest to ładne porównanie i wydaje się, że pozwoli ono uporządkować i zrozumieć zasady języka.

I tak wiemy, że program w języku C przetwarza **argumenty**, czyli operandy (zmienne, stałe) oraz zawarte w tekście programu literały. Wszystko to są liczby i tylko liczby – także wartości logiczne i napisy są liczbami. Jeśli chodzi o „przetwarzanie", to z grubsza omówiliśmy składniki programu języka C, poczynając od najmniejszych:

- **operatory**, czyli symbole działań na argumentach,
- z argumentów i operatorów tworzone są **wyrażenia**, które jako elementarne skrzynki *zwracają wartość*,
- wyrażenia albo mogą stanowić część większych konstrukcji, albo można je „usamodzielnić", stawiając za nimi średnik, co zrobi z nich **instrukcje**, związane z działaniem. Instrukcje to podstawowe „jednostki działania" programu.

Szereg drobnych *instrukcji*, realizujących konkretne zadanie, można potraktować jako jedną *instrukcję złożoną* – jako **blok** (zawarty w parze nawiasów klamrowych). Z kolei blok łatwo zamienić w **funkcję**, nadając mu nazwę i określając „co wchodzi, co wychodzi".

Może i Tobie się wydaje, że koncepcja „większych i mniejszych skrzynek", do których coś wchodzi (argumenty) i które zwracają wartość-wynik, to elegancka droga do zrozumienia języka C. Niestety, nawet jeśli potrafisz wyodrębnić wymienione właśnie podstawowe „cegiełki", z których składa się program w języku C (**funkcje**, **instrukcje**, **wyrażenia**, ***operatory***), to i tak niewiele zrozumiesz. Po pierwsze, zdecydowanie zbyt mało uwagi poświęciliśmy operatorom. Po drugie, wnikliwym Czytelnikom na pewno nasunęły się wątpliwości i pytania związane ze *zwracaniem wartości* oraz dotyczące różnic między *wyrażeniem* a *instrukcją* i roli średnika. Bez zrozumienia tych zagadnień nie można dobrze opanować języka C.

W zasadzie należałoby teraz poznać bliżej bardzo ważne składniki programu,

mianowicie *operatory*. Jednak języka C nie można opanować bez zrozumienia pewnych kluczowych zagadnień ogólnych, między innymi dotyczących *zwracania wartości* oraz różnic między *wyrażeniem* a *instrukcją* i roli średnika.

I tu dochodzimy do sprawy ogromnie ważnej, ale trudnej do zrozumienia, zwłaszcza dla elektroników uczących się języka C. Otóż według podręczników ***wyrażenie zawsze zwraca wartość, a instrukcja – niekoniecznie***...

Przywołajmy tu, co Wikipedia mówi o instrukcjach i wyrażeniach (https://goo.gl/4ZvM8y): *Instrukcja – w programowaniu jest to najmniejszy samodzielny element imperatywnego języka programowania. (...) Program jest tworzony jako zbiór różnych instrukcji. Instrukcja może zawierać wewnętrzne komponenty (np. wyrażenia). (…) Wiele języków (np. C) odróżnia instrukcje i definicje — instrukcja zawiera kod wykonywalny, a definicja deklarację identyfikatora. (…) W większości języków instrukcje różnią się od wyrażeń tym, że niekoniecznie zwracają wyniki i mogą być wykonywane dla osiągnięcia określonych skutków ubocznych, podczas gdy wyrażenia zawsze zwracają wynik i zwykle nie powodują żadnych efektów ubocznych. (...) W językach czysto funkcyjnych nie ma instrukcji – wszystko jest wyrażeniem*.

Uff! I co z tego zrozumie elektronik, zaczynający się uczyć języka C?

Aby wyjaśnić szczegóły, należałoby zagłębić się w analizę obszernych, niełatwych abstrakcyjnych zagadnień informatycznych oraz przedstawić cechy charakterystyczne i różnice między poszczególnymi językami programowania. Jeśli tylko masz na to chęci, siły i czas – zachęcam do poznawania podstaw i różnych aspektów informatyki. Jednak w związku ze specyfiką naszego czasopisma zakładam, że jesteś elektronikiem, a nie informatykiem, i że chciałbyś nauczyć się języka C tylko po to, by programować mikrokontrolery AVR.

Niestety, nie jest to wszystko takie łatwe, bo *krokodyl jest zupełnie inny*. Spróbuję pokazać Ci te zagadnienia w sposób uproszczony, ale i tak wymagać to będzie znacznego wysiłku i najprawdopodobniej w miarę jasny obraz uzyskasz dopiero za jakiś czas, gdy zagadnienia omawiane w kolejnych artykułach „Wokół języka C" ułożą się w logiczną całość. A gdy zrozumiesz, co znaczą poszczególne „składniki" programów i dlaczego to wszystko jest tak zaplanowane i realizowane, wtedy na pewno zaprzyjaźnisz się z językiem C. Uwierz mi na słowo, że wbrew pozorom podstawowa koncepcja języka C na tle innych języków, choć niedoskonała, to jednak okazuje się zaskakująco prosta, logiczna, a wręcz piękna w swej prostocie. Tylko z początku zupełnie tego nie widać...

Dlatego nadal będziemy zajmować się krokodylem...

**Piotr Górecki**

R E K L A M A

AVT 2210 Najprostszy regulator mocy 230V

Podstawową funkcją proponowanego układu jest regulacja siły światła żarówki bądź żarówek zasilanych z sieci energetycznej 230 V. Można go także wykorzystać do płynnej regulacji mocy grzałek a także do regulacji mocy komutatorowych silników (np. w odkurzaczach).

A: 5zł B: 25zł C: 34zł

Znajdź nas na