Wokół języka C

Krokodyl jest zupełnie inny...

część 4 – zwracanie wartości i efekty uboczne

Wielu elektroników chce programować w C bez wgłębiania się w szczegóły języka. Chętnie wykorzystują oni wszelkie „gotowce”. Ich „programowanie” w rzeczywistości polega na drobnych modyfikacjach programów znalezionych gdzieś w Internecie, a gdy coś nie działa, szukają pomocy na forach. Jeśli i Ty masz takie podejście, NIE czytaj tego artykułu! Jeżeli jednak należysz do tych, którzy chcą rozumieć to, co robią – podejmij trud zrozumienia przedstawionych dalej, 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: w programie nie dostrzegamy podobieństwa do tego, co już znamy. Wszystko dlatego, że...

krokodyl jest zupełnie inny!

Już wcześniej mówiliśmy, że proste analogie mogą wprowadzić w błąd. I tak może być z elegancką i przekonującą koncepcją *skrzynek, które zwracają wartość*. Choć nie w pełni obrazuje ona sytuację w języku C, gdzie mamy pomieszane aspekty matematyczne oraz sprzętowe, w sumie jest atrakcyjna i pożyteczna. Trzeba jednak uściślić kwestię zwracania wartości, efektów ubocznych i roli średnika.

Rola i znaczenie średnika

Wcześniej mówiliśmy, że postawienie średnika przekształca wyrażenie w instrukcję. Co to tak naprawdę znaczy?

Może pomyślisz, że średnik można porównać do „korby”, więc wyrażenie to „skrzynka bez korby”, a instrukcja to „skrzynka z korbą”...

Może i coś w tym jest, ale *krokodyl jest zupełnie inny* i taka analogia, zamiast wyjaśnić, raczej wprowadzi w błąd. Otóż mówiąc najbardziej obrazowo, choć nie do końca ściśle: ***średnik w programie C oznacza: zrób, działaj***.

Owszem, *średnik* można i należy kojarzyć z rozkazem: *zrób*. Jeśli za wyrażeniem (abstrakcyjnym tworem matematycznym) postawimy średnik, to nakazujemy działanie – realizację matematycznej abstrakcji.

Ale generalnie *instrukcji* nie powinniśmy porównywać do „skrzynek z korbą”. Choćby tylko dlatego, że są też proste instrukcje, które nie zawierają wyrażeń (np. `goto; break; continue;` czy instrukcja „pusta” `;`). Ponadto w praktyce *instrukcje*, nawet te zawierające w sobie *wyrażenia*, często mają niewiele wspólnego z „matematycznymi skrzynkami”, dlatego, że celem ich wykonania zwykle ***nie jest*** *uzyskanie zwracanej wartości*, tylko osiągnięcie wspomnianych wcześniej ***efektów ubocznych***. A przy tym ważne są czas i kolejność działania.

Krokodyl jest zupełnie inny, więc pewnie znów Cię zaskoczę: otóż niektóre podręczniki słusznie wskazują, że praktycznym sensem średnika (terminatora instrukcji) jest... **zapomnij**.

Ooops...

Zapomnij?

Jest w tym dużo racji, ale ujmijmy to tak: ***średnik*** *ma znaczenie* ***zrób i zapomnij***.

I tu znów o paradygmatach: możemy powiedzieć w uproszczeniu, że w „czystym języku funkcyjnym” program to jeden wielki przepis matematyczny – jedna wielka funkcja-skrzynka, składająca się z różnych „małych skrzynek bez korby”. Choć może się to wydać dziwne, nie ma tam kolejnych kroków – rozkazów. Wykonanie takiego programu możemy rozumieć jako jedną wielką operację matematyczną, która zostanie zrealizowana zgodnie z zasadami matematyki oraz języka programowania. Owszem, ostatecznie w procesorze zostanie ona zrealizowana jako sekwencja elementarnych rozkazów wykonanych w określonej kolejności. Jednak programista „czystego” języka funkcyjnego zupełnie tego „nie widzi” w pisanym programie. Skupia się tylko na zależnościach matematycznych i regułach używanego języka.

Natomiast w imperatywnym języku C w programie mamy szereg instrukcji, oddzielonych średnikami. Wiele z tych instrukcji zawiera „małe matematyczne skrzynki” – *wyrażenia*. Nie ma tu jednej rozbudowanej operacji matematycznej, tylko wiele małych operacji, nie tylko matematycznych, realizowanych w ramach kolejnych kroków – instrukcji, rozdzielonych średnikami. I właśnie średnik ma sens: **zrób** to, co nakazuje instrukcja i **zapomnij**, co zrobiłeś!

Przykładowo zapis

```
b + 5
```

to proste wyrażenie, które może być częścią jakiegoś bardziej złożonego wyrażenia lub funkcji i może pełnić różną funkcję. Wyrażenie zwraca wartość. Wartością zwracaną przez to wyrażenie jest liczba, będąca sumą zawartości zmiennej b i liczby pięć. Jeżeli bezpośrednio lub gdzieś dalej za tym wyrażeniem będzie stał średnik, mający znaczenie *zrób i zapomnij*, wartość tego wyrażenia zostanie obliczona, wykorzystana i... zapomniana.

Ogólnie biorąc, wartości zwracane przez wyrażenia i inne „skrzynki” są nietrwałe i ulotne. W języku C zwracają wartość w tym sensie, że podczas realizacji programu „ich wartość jest do dyspozycji”, ale tylko do najbliższego średnika, który znaczy też *zapomnij*.

Zwracana wartość nie zostanie zapomniana wtedy, gdy za pomocą operatora przypisania zapamiętamy ją w jakiejś zmiennej, np.:

```
a = b + 5;
a = 5 * x;
c = k + 2;
```

Krokodyl jest zupełnie inny, więc zapominanie niesłusznie może się skojarzyć ze zniknięciem, ze stratą, z przekonaniem, że to jakaś wada, błąd, niedoróbka. A to jest specyfika języka, blisko zresztą związana ze sprzętem.

A teraz omówmy zwracanie wartości.

Skrzynki i zwracanie wartości

Skrzynka realizuje mniej czy bardziej skomplikowaną operację. *Cały program* od biedy możemy potraktować jak „największą” skrzynkę, do której „coś wchodzi i coś wychodzi”. Niewątpliwie „skrzynkami średniej wielkości” są *funkcje* stworzone zgodnie z regułami języka C. Jak widać w deklaracji funkcji

```
typ_zwracanego_wyniku nazwa_funkcji (argumenty_przekazywane_do_funkcji)
```

mamy tu podstawową koncepcję skrzynki: ***do funkcji zawsze przekazujemy argumenty, a funkcja zawsze zwraca wynik określonego typu***. Aby mocno trzymać się tej jasnej koncepcji, to gdy nie mamy argumentów lub wyniku, mówimy o „pustym” typie danych (void), np.

```
void nazwa_funkcji (void)
```

Zasadniczo *zwracanie wartości* wiąże się z abstrakcyjno-matematycznym aspektem programu i jego „składników”. Owszem, w czasie działania programu jest jakoś realizowane fizyczne, ale w sumie chodzi o odciążenie programisty, by nie musiał rozumieć i dbać o wszystkie szczegóły. By mógł skoncentrować się tylko na tym, co najważniejsze. Niemniej warto wiedzieć, do kogo czy do czego przekazywane są wartości – wyniki?

Przypomnijmy informacje podstawowe: „największa skrzynka”, czyli program napisany w C, ma główną, „obowiązkową” funkcję **main** w postaci

```
int main (argumenty_lub_void) {
/* instrukcje */
return 0;
}
```

Umieszczona na końcu instrukcja ***return*** powoduje, że program zwraca wartość-wynik w postaci liczby całkowitej typu *int*, w tym przypadku zwraca liczbę zero. Czy pamiętasz, do kogo (komu) zwraca?

W przypadku komputera sprawa jest prosta: program taki, kończąc swe działanie, jako swą ostatnią operację zwraca liczbę zero *do systemu operacyjnego* i w ten sposób sygnalizuje, że realizacja programu przebiegła prawidłowo. Ma też sens przekazanie argumentów do funkcji *main* – ewentualne argumenty przekaże o niej właśnie system operacyjny, czyli program(-y) zarządzający pracą komputera.

Jednak w przypadku mikrokontrolerów jednoukładowych nie ma komu zwrócić wartości, ponieważ nie ma systemu operacyjnego, a ponadto zazwyczaj podstawą działania jest nieskończona pętla **for** lub **while**. Dlatego główny program dla mikrokontrolera AVR może zwracać wartość „*void*”, czyli nie zwracać niczego, a tym samym może nie zawierać instrukcji ***return***:

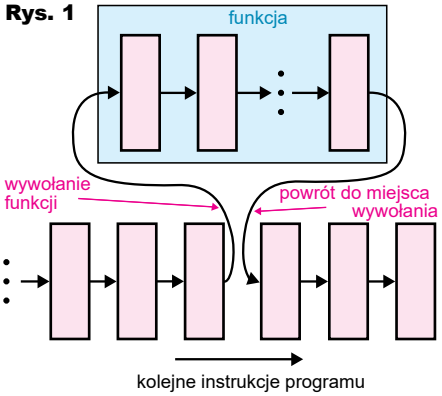
```
void main (void) {
while (1){
   /* instrukcje programu */
   }
}
```

Jednak także w programie dla mikrokontrolera zazwyczaj wykorzystuje się wcześniej podaną postać „coś zwracającą”, więc gdyby konsekwentnie trzymać się koncepcji skrzynek, także funkcja *main*, jak wszystkie inne funkcje, powinna zawierać instrukcję *return*:

```
int main (void) {
   while (1) {
   /* instrukcje */
   }
return 0;
}
```

Zachowamy wtedy reguły języka C, a kompilator i tak pominie ten szczegół. O ile w przypadku mikroprocesorów jest pewien kłopot teoretyczny ze zwracaniem wartości przez funkcję *main()*, o tyle problemu takiego nie ma w przypadku innych funkcji programu, które zwracają wartość. Najogólniej mówiąc, podczas pracy programu funkcje są wywoływane, a po wykonaniu *zwracają wartość* typu określonego w definicji funkcji tam (w tym miejscu programu), dokąd wracają, czyli tam, skąd zostały wywołane (uruchomione). Ilustruje to **rysunek 1**. Logika podpowiada, że zwracana wartość może być wykorzystana do realizacji dalszej części programu. Jednak zwrócenie wartości przez funkcję lub wyrażenie wcale nie oznacza, że ta wartość zostaje automatycznie i trwale zapisana czy wykorzystana. Raczej polega to na tym, że „zwracana wartość jest do dyspozycji” i albo zostanie wykorzystana, albo nie.

Rys. 1



To, że program według rysunku 1 po wykonaniu funkcji wraca do miejsca jej wywołania (uruchomienia), jest proste i oczywiste, tylko czy ktoś lub coś czeka na zwracaną wartość, by ją wykorzystać?

Otóż najprościej biorąc, zwracaną wartość możemy „złapać” do jakiejś zmiennej, określonej w instrukcji wywołania funkcji. W tym celu napiszemy:

```
czekajaca_zmienna = funkcja (argumenty);
```

Wartość zwracaną przez funkcję „łapiemy” w tym przypadku do tej czekającej zmiennej. Ale co ciekawe, bardzo często wywołujemy funkcję

```
funkcja (argumenty);
```

nie podając żadnej „czekającej zmiennej”. Co wtedy?

Jak myślisz?

Możliwości jest kilka. Można się domyślać, że jeśli „nie złapiemy zwracanego wyniku”, to on „zmarnuje się, zginie”. Natomiast gdy dana funkcja z zasady zawraca „pusty” typ ***void***, czyli tak naprawdę nie zwraca niczego, to nie ma problemu, bo nie ma czego „łapać”. Ale tu nasuwa się bardzo ważne pytanie: ***jaki jest sens realizacji funkcji, która albo z zasady niczego nie zwraca, albo gdy „gubimy” zwracany przez nią wynik?***

Wygląda to na poważny zgrzyt w eleganckiej i przekonującej koncepcji skrzynek, do których przekazujemy argumenty i które zwracają wartości. A wydawało się nam, wręcz byliśmy pewni, że język C opiera się właśnie na jasnej i eleganckiej koncepcji większych i mniejszych skrzynek, zwracających wartość...

Dla początkujących jest to nieprzyjemna zagadka. Bo przecież zgodnie z podstawową koncepcją, *funkcja zawsze zwraca wartość*. No tak, tylko często albo jest to wartość *nic*, albo tę zwracaną wartość „gubimy”. Podobnie jest z *wyrażeniami*...

Gdzie tu sens i logika?

Odpowiedź brzmi: *bo krokodyl jest zupełnie inny*... A mówiąc poważniej, musimy omówić kolejne ważne i niełatwe zagadnienie.

Efekty uboczne

Podkreślmy, że w języku C jak najbardziej obowiązuje sprzyjająca porządkowi i ograniczająca błędy koncepcja skrzynek. Według tej koncepcji „efektem podstawowym” działania funkcji (oraz innych „skrzynek”) jest zwracanie wartości.

Tak. Tylko zwracanie wartości.

Ale najprościej biorąc, prawie wszystkie funkcje w języku C, a szczególnie w wersji dla mikrokontrolerów, oprócz zwracania wyniku, realizują też tak zwane

efekty uboczne (*side effects*). I paradoksalnie, bardzo często zależy nam nie na zwracanym wyniku, tylko właśnie na różnorodnych efektach ubocznych działania funkcji. Natomiast zwracanie wyniku okazuje się tylko formalnością, wymaganą przez koncepcję i reguły języka C. Co bardzo ważne, dotyczy to nie tylko „dużych skrzynek" jakimi są *funkcje*, ale też „mniejszych skrzynek".

Może znów Cię zaskoczę, ale bardzo popularnym efektem ubocznym jest coś tak na pozór naturalnego jak *zmiana wartości zmiennej*!

Efektami ubocznymi są wszelkie operacje wejścia/wyjścia, czyli w przypadku mikrokontrolerów wszelkie zmiany stanu rejestrów obsługujących porty i inne urządzenia peryferyjne. Przykładowo efekty uboczne realizują instrukcje

```
DDRB = 0xFF;
PORTB = 0x5A;
```

z których pierwsza ustawi piny portu B jako wyjścia, a druga ustawi ich stan logiczny. Jest to typowy efekt uboczny, bo nie ma związku ze zwracaniem wartości przez „skrzynki". Także wcześniej podana instrukcja

```
a = b + 5;
```

zawarta w jakiejś funkcji realizuje bardzo popularny efekt uboczny. Wyrażenie **b + 5** zwraca wartość – liczbę, ale wartość ta jest ulotna i „istnieje tylko do najbliższego średnika" (podobnie powiemy, że wyrażenie **a = b + 5** zwraca liczbę równą zawartości zmiennej a, która jest sumą **b + 5**). Aby jej nie utracić, wykorzystujemy efekt uboczny: za pomocą operatora przypisania zapamiętujemy ją w zmiennej **a**.

Sama nazwa „efekt uboczny" nie tylko przez skojarzenie z medycyną sugeruje coś podrzędnego, mniej potrzebnego lub nawet niepotrzebnego i może groźnego. A tymczasem tu efekt uboczny w postaci przypisania „zwiększa trwałość" i umożliwia wykorzystanie w innym miejscu programu, czyli wbrew nazwie wydaje się czymś pożądanym, a nawet bardzo ważnym. Znów mamy przykład, że *krokodyl jest zupełnie inny*.

Kto i po co to tak skomplikował?

Patrząc z punktu widzenia mikrokontrolerów, może to wyglądać na bezsensowne i niepotrzebne fanaberie. Jednak w przypadku różnych skomplikowanych programów komputerowych pisanych w C lub innych językach, brak efektów ubocznych jest istotną zaletą. Zmniejsza ryzyko błędów i nieoczekiwanych reakcji. Programista może się bowiem skoncentrować tylko na zwracanych wartościach, a nie musi analizować wszystkich możliwych następstw efektów ubocznych. A zgodnie z nazwą, „efekty uboczne" mogą też mieć nieprzewidziane i nieprzyjemne skutki. I tak bywa zarówno „w dużych programach", jak też w przypadku programowania mikrokontrolerów za pomocą języka C, gdzie z konieczności wykorzystuje się głównie efekty uboczne.

Programując w asemblerze, programista musiał pamiętać o każdym szczególe. Program w asemblerze to wyłącznie „efekty uboczne" polegające na kolejnych zmianach stanu zmiennych, czyli zmianach zawartości pamięci. Języki wyższego poziomu miały odciążyć programistę i uniezależnić go od sprzętu, by nie musiał pamiętać i analizować wszystkich szczegółów dotyczących budowy i działania procesora. Tym ułatwieniem miała być dobra koncepcja języka i odpowiednio inteligentny kompilator. Zgodnie z ogólnym kierunkiem rozwoju języków programowania, języki te miały skutecznie oddzielić programistę od szczegółów sprzętowych i od niuansów specyfiki procesora. Stąd elegancka koncepcja „skrzynek" i stąd dążenie do eliminacji efektów ubocznych. Dlatego na biegunie przeciwnym do asemblera mamy czyste języki funkcyjne, gdzie (prawie) w ogóle nie powinno być efektów ubocznych, a obiektem zainteresowania programisty mają być wyłącznie wartości zwracane przez funkcje.

Ponieważ różne były potrzeby i możliwości, powstało wiele języków programowania, w tym C, a żaden nie jest uniwersalny i doskonały. Język C był dużym i ważnym krokiem w kierunku uniezależnienia programisty od specyfiki sprzętu. Przyniósł koncepcje i mechanizmy bardzo pożyteczne. Tymczasem my wykorzystujemy język C do programowania mikrokontrolerów jednoukładowych, co jest ewidentnym „cofnięciem się w kierunku asemblera" i wtedy absolutnie nie możemy uciec od szczegółów sprzętowych. Ponadto program w języku C zostanie przekompilowany: najpierw na elementarne polecenia asemblera i potem na kod maszynowy procesora. W praktyce często sprawdzamy, jak kompilator zamienił program pisany w C na polecenia asemblera. Niekiedy z ciekawości, ale zwykle po to, by znaleźć trudno wykrywalne błędy (związane właśnie z efektami ubocznymi).

Nam jako elektronikom bliżej do konkretów dotyczących procesora, jego pamięci, rejestrów, peryferii. Gdy więc zaczynamy poznawać język C, w pierwszej chwili cieszymy się z eleganckiej koncepcji skrzynek, ale już za chwilę przytłaczają nas niezrozumiałe aspekty informatyczne i matematyczne – *bo krokodyl jest zupełnie inny*.

Paradoksalnie ktoś, kto nie jest elektronikiem, może szybciej opanować język C, bo nie doszukuje się związku pisanego programu z konkretnym działaniem procesora, jego pamięci i rejestrów. No tak, tylko ktoś, kto nie rozumie szczegółów działania procesora, może mieć i będzie mieć duże problemy z rozmaitymi pułapkami, związanymi ze specyfiką używanego procesora.

Wykorzystanie języka C do programowania mikrokontrolerów okazuje się więc trudniejsze niż pisanie programów na komputery w „czystym języku C". Nie ma na to rady. Z jednej strony trzeba zgłębić i zrozumieć specyficzne reguły języka C, które znakomicie ułatwiają pisanie „programów komputerowych", a z drugiej strony trzeba poznać mikroprocesor i jego specyfikę. Nie sposób w EdW omówić wszystkich problemów osób uczących się i używających języka C do programowania mikrokontrolerów, niemniej w ramach serii „Wokół języka C" nadal będziemy publikować artykuły pokazujące rozmaite aspekty tego problemu. Także i Ty możesz opanować język C w satysfakcjonującym stopniu, o ile tylko nie przestraszysz się *krokodyla, który jest zupełnie inny*.

Piotr Górecki