Wokół języka C
Krokodyl jest zupełnie inny...
Część 1 – C z lotu ptaka

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, proszą o pomoc 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.

Uznałem, że idealnym wprowadzeniem do problemu poruszanego w artykule (i tytułu) jest stary dowcip o krokodylu. Ja kiedyś dawno słyszałem go w gwarze śląskiej, ale przytaczam tu inną wersję:

Władek pojechał do miasta i był w ZOO. Po powrocie Stasiek pyta go, co widział.

– Ano, zebrę widziałem.
– Co to ta zebra?
– A wiesz, jak wygląda koń?
– A jakże
– No to zebra jest jak koń, tylko w czarne i białe paski.
– I co jeszcze widziałeś?
– Ano, żyrafę widziałem.
– Co to ta żyrafa?
– A wiesz, jak wygląda koń?
– A jakże.
– No to żyrafa jest jak koń na długich nogach i ze strasznie długą szyją.
– I co jeszcze widziałeś?
– Krokodyla.
– A co to takiego?
– A wiesz, jak wygląda koń?
– A jakże.
– No to krokodyl jest zupełnie inny!

No właśnie! Chcąc poznać coś nowego, zazwyczaj próbujemy znaleźć analogie do tego, co już znamy. Świadomie czy nie, tak właśnie próbujemy też robić, ucząc się programowania. Próbujemy dopasować „to nowe, nieznane” do tego, co już wiemy i znamy. Względnie dobrze udaje się to w przypadku języka Basic i BASCOM-a. Dostrzegamy tam pewne podobieństwo do tego, co już znamy. Podobnie, widząc listingi języka C, rozpaczliwie próbujemy się doszukać podobieństwa do tego, co już znamy z matematyki, fizyki, z BASCOM-a. Daremnie! Wszystko dlatego, że...

krokodyl jest zupełnie inny!

Język C jest zupełnie inny niż wszystko, z czym mieliśmy do czynienia w domu, w szkole, w pracy... Zupełnie inny niż w miarę przyjazny BASCOM. Tu wyjaśnienie, dlaczego proponuję Ci szereg artykułów *Wokół języka C* zamiast intensywnego cyklu „prostej praktyki”: tak jak już pisałem wcześniej, informatyka i programowanie to nowe, bardzo obszerne dziedziny. Pojawiły się w nich liczne pojęcia, dla których nie znajdziemy odpowiedników w codziennym życiu. A nawet słowa, pojęcia i operacje, wykorzystywane na co dzień, w informatyce przedstawiane (zapisywane) są inaczej. Oto dwa główne problemy osoby uczącej się języka C:

1 – konieczność opanowania szeregu zupełnie dotąd obcych pojęć i zagadnień
2 – konieczność przyzwyczajenia się do innego sposobu zapisu tego, co już znamy.

Dodatkowym poważnym problemem jest ogrom dostępnych informacji, zarówno dotyczących języka C, jak i mikrokontrolerów AVR. Niestety, **początki *samodzielnego* programowania w C zawsze są ogromnie frustrujące** z uwagi na mnóstwo popełnianych błędów. I właśnie zarówno **BASCOM**, jak też **Ardiuno** są próbą uproszczenia, „pójścia na skróty”, bez wnikania w szczegóły. Jak widać, daje to szybkie efekty i dla wielu jest źródłem ogromnej satysfakcji. Dla amatorów, „domorosłych majsterkowiczów” bywa to znakomitym rozwiązaniem. Ale niestety, takie uproszczenie z kilku powodów okazuje się ślepą uliczką dla tych, którzy chcieliby pisać porządne, profesjonalne programy i którzy chcieliby rozumieć, co tak naprawdę robią. Choć wejście w świat Arduino (gdzie wykorzystuje się mocno uproszczoną wersję C, ściślej C++) byłoby dużo szybsze i prostsze, celowo proponuję najpierw zadanie trudniejsze: „zapoznanie z krokodylem”. Owszem, zajmiemy się też Arduino (nieprzypadkowo tak często wspominamy o ATmega328), ale dopiero w drugiej kolejności, po przynajmniej częściowym zaprzyjaźnieniu z „klasycznym” językiem C. A ponieważ temat jest ogromny, będziemy przedstawiać kolejne artykuły „oswajające” z serii *Wokół języka C*.

Znakomitą metodą w procesie uczenia się jest poznanie najpierw ogólnego zarysu, szkieletu, podstaw zagadnienia, zasady działania, a dopiero potem stopniowe poznawanie bardzo licznych, obszernych szczegółów. Trzeba też ustalić i uściślić terminologię, ponieważ w publikacjach dotyczących programowania spotyka się liczne niejasności i niekonsekwencje. Czy można krótko pokazać zarys tak ogromnej dziedziny jak informatyka?

Spróbujmy to zrobić, choćby tylko częściowo, w sposób uproszczony, uwzględniając szczególne potrzeby elektroników, Czytelników EdW.

Widok z lotu ptaka

Zacznijmy od tego, że w sumie chodzi o zrealizowanie pewnych zadań i rozwiązanie pewnych problemów z wykorzystaniem procesorów i oczywiście programu.

Aby jednak powstał program dla procesora, trzeba najpierw ***dobrze zrozumieć zadanie i problem***. Następnie trzeba ***przyjąć jakiś sposób rozwiązania***. Takich różnych sposobów rozwiązania jest zawsze mnóstwo, więc trzeba się zdecydować na konkretny *ciąg jasno zdefiniowanych czynności, koniecznych do wykonania zadania*, czyli na... ***algorytm***, bo taka właśnie jest definicja algorytmu.

Algorytm to przepis, który można zapisać na różne sposoby: słownie (na ogół mało dokładnie), w postaci listy kroków, w postaci *schematu blokowego* lub w postaci grafu-drzewa. Inaczej mówiąc, trzeba postawione zadanie rozłożyć na coraz mniejsze podzadania.

Dopiero mając sensowny *algorytm, czyli przepis na zrealizowanie zadania*, możemy ***przystąpić do pisania kodu programu***, czyli do kodowania. ***Program komputerowy to algorytm (przepis) napisany w konkretnym języku programowania.***

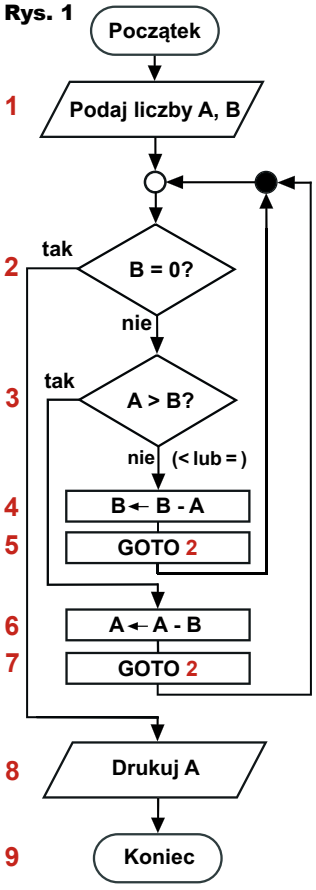
Błędem (z wyjątkiem najprostszych przypadków) jest zaczynanie pracy od pisania programu, bez wcześniejszego przemyślenia i przygotowania przepisu – algorytmu.

I tu ważna dygresja. Wielu początkujących z obrzydzeniem podchodzi do algorytmów, co nawet może być zrozumiałe w kontekście różnorodnych szkolnych doświadczeń. Niecierpliwi młodzi chcą zaczynać od upragnionego *pisania programów*, bez wcześniejszego gruntownego przeanalizowania problemu/zadania i bez tworzenia jakiegokolwiek przepisu-algorytmu. W przypadku prostych programów jako tako się to udaje. Ale przy bardziej skomplikowanych, już po pierwszych testach wychodzą na jaw najróżniejsze błędy i trzeba program zmieniać. Wtedy nie jest to zwyczajny *debugging*, czyli *odpluskwianie* programu, ponieważ konieczne jest wprowadzenie nie tylko mnóstwa poprawek, ale też poważniejszych zmian.

Zaczynając „budowę od dachu”, czyli od pisania kodu programu, a nie od fundamentów, czyli od analizy zadania i przepisu-algorytmu, kandydat na programistę na własne życzenie robi sobie poważną szkodę. Uczy się bałaganiarskiego podejścia, które będzie bardzo poważną barierą przy trudniejszych zadaniach. Słuszne jest stwierdzenie, że **nie każdy może zostać dobrym programistą**. Nieprzypadkowo też mówi się, że do „prawdziwego” programowania i *do języka C trzeba dorosnąć, a niektórzy nigdy do tego nie dorastają*. Dobry programista musi posiąść zdolność logicznego myślenia, analizy sytuacji, jasnego określania problemów. To jest niezbędne do stworzenia dobrego przepisu-algorytmu, a to co najmniej połowa sukcesu. Natomiast *zakodowanie* opracowanego algorytmu w którymś z języków programowania okazuje się zadaniem... drugorzędnym, łatwiejszym. Bo *kodowania* (*pisania programów*) można się lepiej czy gorzej nauczyć, a umiejętność analizy problemu i jego różnych aspektów oraz zdolność rozkładania dużych zadań na mniejsze fragmenty to naprawdę trudna sztuka. Na pewno sporo zależy tu od wrodzonych zdolności, ale wiele można opanować świadomym wysiłkiem.

Spotyka się stwierdzenia, że *programista musi opanować sztukę „myślenia algorytmicznego”*. Owszem, w przypadku wielu poważnych zadań programistycznych rzeczywiście jest to konieczne. Jednak nie trzeba być ekspertem „myślenia algorytmicznego” w przypadku programowania procesorów AVR, czyli zadań w sumie niezbyt trudnych. Niemniej i tu naprawdę warto zaczynać nie od pisania programu, tylko od gruntownego przemyślenia zadania/problemu, a w szczególności od możliwych przykrych niespodzianek. Te przykre niespodzianki wynikają ze specyficznych właściwości zarówno procesora AVR, jak też języka C, a ich przyczyną jest brak wiedzy i doświadczenia. Liczba takich ujawniających się niespodzianek maleje wraz z nabieraniem wprawy i nie można się tego nauczyć „na sucho”, bez praktyki.

Rys. 1



Niestety, wspomniany problem „zaczynania budowy od dachu” w dużym stopniu dotyczy elektroników, którzy chcą programować mikroprocesory. Tu trzeba przyznać, że programy dla mikrokontrolerów w większości zawierają powtarzalne, „standardowe” rozwiązania, gotowe bloki kodu i są nieporównanie prostsze od różnych skomplikowanych programów „komputerowych”. Podobnie jak prosty jest schemat układu, tak samo proste bywa napisanie programu, zawierającego standardowe „gotowe kawałki”. Efektem jest zalew prostych projektów mikroprocesorowych, zawierających wyświetlacz, przyciski i jakiś czujnik.

Korzystając z „gotowców”, często wystarczy napisać, a raczej dopisać kilka czy kilkanaście linijek kodu i...
układ działa!

Czy to sukces? Po części tak. Ale problemy zaczynają się, gdy układ i program są bardziej skomplikowane...

I tu pierwsza rada: **nie zaczynaj od pisania programu! Najpierw starannie przeanalizuj zadanie** z różnych stron. Stwórz schemat ideowy urządzenia mikroprocesorowego, zastanów się, jakie będą sygnały wejściowe i wyjściowe mikroprocesora. Jakiej „wielkości” mikrokontroler będzie potrzebny? Które końcówki do czego wykorzystać? Zastanów się nad różnymi możliwymi przypadkami, jakie mogą wystąpić, zwłaszcza niekorzystnymi. Następnie nie śpiesząc się, obmyśl *przepis* na działanie procesora (żeby nie straszyć Cię słowem *algorytm*). Taka analiza i przygotowanie przepisu-algorytmu zwykle się opłaca, bo pozwala uniknąć mnóstwa błędów.

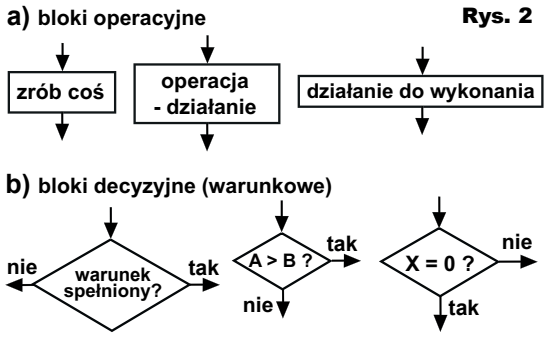
Algorytm

Można stworzyć opis słowny realizacji zadania, ale dobrze byłoby narysować ten przepis-algorytm w klasycznej postaci *schematu blokowego*. Przykład na **rysunku 1** to schemat blokowy tzw. algorytmu Euklidesa – przepisu znajdowania największego wspólnego dzielnika dwóch liczb (sprzed ponad 2000 lat, wykorzystywanego do dziś).

Nie musisz rozumieć szczegółów tego przykładu. Wiedz tylko, że najważniejsze elementy algorytmu na schemacie blokowym to:
- (prostokąt) ***blok operacyjny*** reprezentujący ściśle określone działanie
- (romb) ***blok decyzyjny – warunkowy***, gdzie działanie zależy od spełnienia jakiegoś warunku (tak/nie)

Na **rysunku 2** pokazane są przykłady. Te dwa podstawowe składniki pozwalają przedstawić trzy podsta-

Rys. 2

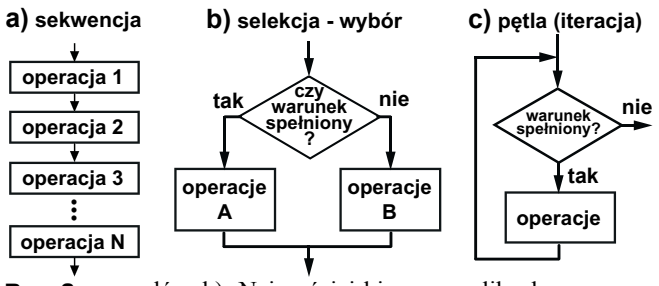


wowe konstrukcje strukturalne algorytmów, którymi są:

sekwencja – cykl poleceń wykonywanych w ściśle określonej kolejności

selekcja – wybór, zależny od spełnienia warunku

pętla (cykl, iteracja) – powtarzanie czynności: albo do spełnienia warunku, albo z określoną z góry liczbą powtórzeń. **Rysunek 3** pokazuje proste przykłady.



Rys. 3

Każdy *algorytm* można *zapisać*, wykorzystując tylko *sekwencje*, *selekcje* i *pętle*. Potem można go *zakodować* w jakimś języku programowania, uzyskując elegancki *program* strukturalny. Ze wspomnianymi podstawowymi konstrukcjami mamy oczywiście do czynienia także w języku C. Jeśli nawet nie pomyślisz o algorytmie-przepisie, tylko „wprost" napiszesz program, to i tak ten program w rzeczywistości będzie wykorzystywał przedstawione właśnie trzy podstawowe konstrukcje. Tylko tyle o algorytmach. A teraz o programie.

Program w języku C

Najogólniej biorąc, program w języku C składa się z:

1 – **komentarzy**,
2 – **dyrektyw preprocesora**,
3 – **deklaracji**,
4 – **instrukcji**.

1. ***Komentarze*** (chyba nie wymagają komentarza) – są przeznaczone dla człowieka, nie dla kompilatora. W dowolnej linii programu za znakami // możesz dodać komentarz liniowy

```
c = a+b; // tu jest komentarz (jedno)liniowy
```

Możesz też wstawić (wieloliniowy) komentarz blokowy, między znakami /* */

```
/* to jest
komentarz
blokowy */
```

Warto w programie umieszczać zwięzłe, sensowne komentarze wskazujące na najważniejsze szczegóły – nie tylko dla kogoś innego, ale za jakiś czas i dla Ciebie będą one ogromną pomocą przy analizie i ulepszaniu programu.

2. ***Preprocesor*** to program, który przed właściwą kompilacją, „w sobie znany sposób" realizuje dyrektywy, poprzedzone w programie znaczkiem # (hash). Najczęściej stosowane są dwie. Dyrektywa

```
#include <plik_naglowkowy.h>
```

lub też

```
#include "plik_naglowkowy.h"
```

powoduje dołączenie do Twojego programu treści zawartej w pliku nagłówkowym z rozszerzeniem **.h** (h – od header – nagłówek). Najprościej biorąc, w plikach nagłówkowych (niekiedy obszernych) zawarte są liczne napisane w języku C „gotowce", znakomicie ułatwiające pisanie Twojego programu – po prostu większość roboty ktoś już zrobił za Ciebie.

Jeżeli nazwa pliku nagłówkowego będzie ujęta w nawiasy ostre < >, to kompilator poszuka go wśród własnych plików nagłówkowych (zwykle w podkatalogu *.../includes/* katalogu kompilatora). Jeśli nazwa będzie ujęta w podwójne cudzysłowy **" "** kompilator poszuka pliku w katalogu, w którym znajduje się plik z Twoim głównym programem.

Druga często spotykana dyrektywa preprocesora **`#define`** dziwi początkujących. Otóż jeżeli na przykład wstawisz dyrektywę

```
#define parametr 215
```

to preprocesor przed kompilacją zmieni treść Twojego programu: wszędzie tam, gdzie Ty żmudnie w programie wpisałeś **`parametr`**, on wstawi liczbę 215. Jest to bardzo wygodne na przykład, gdy **`parametr`** w treści Twojego programu występuje wielokrotnie, a w chwili pisania programu jeszcze nie wiadomo dokładnie, jaką ma mieć wartość. Warto z tego korzystać także do określania wykorzystywanych portów – przy zmianie procesora wystarczy zmienić treść dyrektywy **`#define...`**

A oto inne dyrektywy preprocesora, o których na razie nie musisz nic wiedzieć: *#undef, #if, #elif, #else, #endif, #ifdef, #ifndef, #else, #endif, #error, #warning, #line,* często występujące w plikach nagłówkowych.

3. ***Deklaracje i definicje***. Gdy preprocesor wstępnie przetworzy plik z Twoim programem i zgodnie z dyrektywami **`#include`** doda do niego zawartość plików nagłówkowych, wtedy właściwy kompilator kolejno linijka po linijce przeanalizuje tak przygotowany program.

Niektórzy mówią, że kompilator języka C to strasznie wredny służbista, który zanim zrobi swoją właściwą robotę, wcześniej *niczym nietoperz, wszystkiego się czepia*. Ta „właściwa robota" kompilatora to zamiana kodu Twojego programu (na rozkazy asemblera i) na kod maszynowy. Natomiast „czepianie się" to pilne wyszukiwanie nie tylko błędów, ale też wszelkich, niewielkich nawet niejasności. Jeżeli kompilator napotka cokolwiek, czego „nie zna" lub co wyda mu się „podejrzane", zgłosi błąd (*error*) lub ostrzeżenie (*warning*). Wystąpienie *ostrzeżenia* nie zablokuje wprawdzie pracy kompilatora, ale dla swojego dobra **ostrzeżenia (warnings) traktuj jak błędy** i dotąd poprawiaj program, aż kompilator przestanie je zgłaszać.

Kompilator zna tylko tak zwane *słowa kluczowe* (których w języku C jest sumie niewiele: ***char, double, float, int, long, short, signed, unsigned, void, enum, struct, typedef, union, sizeof, auto, const, extern, static, volatile, break, case, continue, do, default, else, for, goto, if, return, switch, while***). Więc zgłosi problem, jeśli znajdzie w programie jakąkolwiek inną, nieznaną nazwę (identyfikator). Aby usunąć problem, trzeba wcześniej zadeklarować wszelkie identyfikatory: stałe, zmienne i nazwy funkcji.

I nie złość się, że kompilator jest tak strasznie upierdliwy w wyszukiwaniu błędów i wszelkich wątpliwości. Wprost przeciwnie, potraktuj to jako surowe piękno języka C: to wszystko jest dla Twojego dobra!

Dlatego wszystkie *zmienne* w programie C muszą być prawidłowo zadeklarowane (we właściwym miejscu w programie, z podaniem typu i ewentualnych modyfikatorów). *Zadeklarowanie zmiennych przed główną funkcją* ***main*** *spowoduje, że te zmienne będą globalne, dostępne dla wszystkich funkcji programu*. Natomiast *zadeklarowanie zmiennych (zawsze na początku danego) bloku/funkcji spowoduje, że będą to zmienne lokalne, tylko dla tego bloku/funkcji*. Kompilator sprawdzi też typy danych wszystkich zmiennych i stałych – czy na przykład niektóre zmienne (rezerwowane fragmenty w pamięci RAM) nie są aby „za małe" do przewidzianych dla nich zadań.

Nie tylko *zmienne*, ale też wszystkie *funkcje* muszą być znane kompilatorowi – zadeklarowane. Już wiesz, że w przypadku funkcji deklaracją jest jej definicja (nagłówek funkcji, zwany prototypem). Ale jeżeli chcesz w swoim programie użyć funkcji „obcych", dołączonych z zewnątrz, musisz je wcześniej w programie zadeklarować. Jest to dość obszerne zagadnienie. Często problem wynika z faktu, że kompilator „nie ogarnia całości", tylko analizuje program i szuka błędów kolejno linijka po linijce. Jeśli więc napotka identyfikator, który jest zadeklarowany/zdefiniowany dalej

w programie, to zgłosi błąd, bo nie widzi tego, co jest „dalej”. Dlatego zmienne lokalne powinny być deklarowane na początku bloku/funkcji. Ponadto często Twój program, oprócz głównej funkcji ***main***, zawiera też inne funkcje. Jeżeli te inne funkcje miałyby być wykorzystywane w funkcji *main*, to albo w programie ich *definicje* musisz umieścić przed funkcją *main*, albo najpierw musisz je *zadeklarować*, by potem móc je umieścić (*zdefiniować*) za funkcją *main*.

Warto dodać, że kompilator języka C rozróżnia małe i WIELKIE litery, więc *zmienna*, *Zmienna* i *zmiennA* to trzy różne zmienne. W programach C w nazwach używamy zasadniczo tylko małych liter (i znaku _ jako łącznika), a WIELKIE LITERY stosujemy praktycznie tylko w odniesieniu do stałych i w dyrektywach preprocesora.

4. ***Instrukcje*** to najbardziej nas interesujące części programu. To one tworzą „właściwy program”, a omówiona już reszta to „składniki pomocnicze”. Słowo *instrukcja* kojarzy się z *działaniem*, *rozkazem*, *poleceniem*, *akcją*. Program C składa się z mnóstwa prostszych i bardziej złożonych instrukcji. Można powiedzieć, że *program jest zestawem instrukcji*.

Wcześniej powiedzieliśmy, że *każdy algorytm można zapisać, wykorzystując tylko* ***sekwencje***, ***selekcje*** i ***pętle***. Nic więc dziwnego, że w programie znajdziemy ***instrukcje selekcji*** (wyboru), ***instrukcje pętli*** (iteracyjne) oraz liczne „drobne” instrukcje, które składają się na ***sekwencje*** instrukcji. W języku C mamy dwie *instrukcje selekcji*: **if** (**if – else**) oraz **switch**, gdzie działanie zależy od spełnienia/niespełnienia warunku (czy warunek jest prawdziwy, czy nie). Do dyspozycji mamy też trzy *pętle*: **while**, **do...while** oraz **for** (gdzie też często uzależniamy działanie pętli od spełnienia warunku) oraz związane z nimi instrukcje pomocnicze: **break**, **goto** służące do wychodzenia z pętli i **continue**, powodująca pominięcie rozkazów pętli zawartych za tą instrukcją.

Natomiast *sekwencje* to różne (a nawet najróżniejsze) instrukcje, wykonywane kolejno jedna po drugiej w ustalonej kolejności. O ile *instrukcje selekcji* i *pętli* są nieliczne (i jako słowa kluczowe), ściśle określone przez twórców języka, o tyle w skład *sekwencji* mogą wchodzić prostsze i bardziej złożone *instrukcje tworzone przez programistę* według określonych reguł.

I tu należałoby bliżej przyjrzeć się tym „określonym regułom” języka C, w tym jego *syntaktyce* (*składni*), *semantyce*, a może nawet *pragmatyce*. Jeśli składnia (zapis) będzie niepoprawna, niezgodna z zasadami języka, to kompilator oczywiście zgłosi błąd. Ale nawet jeśli formalnie zapis będzie prawidłowy (poprawny składniowo), nie znaczy to, że będzie sensowny, czyli poprawny semantycznie. Program z błędami semantycznymi być może skompiluje się (na pozór) prawidłowo, bez błędów i ostrzeżeń, ale... nie będzie działał. Z kolei nawet poprawny składniowo i semantycznie program wcale nie musi realizować dokładnie tego, czego od niego oczekiwaliśmy. Problemy te występują we wszystkich językach, a surowym pięknem jest to, że „bardzo upierdliwy” kompilator C swoimi ostrzeżeniami pozwala uniknąć wielu błędów. Ale mnóstwo innych błędów przepuści, bowiem kompilator nie wniknie w Twoje myśli, więc nie może sprawdzić, czy program robi dokładnie to, czego od niego oczekujesz.

W zasadzie powinniśmy tu więcej pomówić o *składni* (prawidłowości zapisu według reguł języka), o *semantyce* (co tak naprawdę znaczy zapis programu), a potem także o *pragmatyce* (na ile dokładnie program realizuje postawione, z życia wzięte zadanie). Są to jednak obszerne zagadnienia. Tym bardziej że istnieje mnóstwo języków programowania, opartych na różnych założeniach, mających inne zestawy reguł. To ogromny świat zagadnień bardzo interesujących, trudnych, do których opisu potrzebnych jest mnóstwo nowych abstrakcyjnych pojęć, których nie mamy w życiu codziennym.

Tymczasem my chcemy programować jedynie mikrokontrolery AVR i jeśli już mamy poznawać tego krokodyla, czyli musimy wejść w ten ogromny informatyczny świat, to tylko tyle, ile to naprawdę konieczne. A konieczne jest między innymi dlatego, że najwięcej czasu tracimy na wyszukiwanie błędów, wynikających ze słabej znajomości koncepcji, pojęć oraz reguł języka C. Problemem jest też to, że niektóre słowa, których używamy w języku potocznym, w informatyce mają inne, ściśle określone znaczenie. Ale niestety, nie wszystko jest tam ścisłe i jednoznaczne. Dodatkowo w polskojęzycznych publikacjach z dziedziny informatyki, zwłaszcza w Internecie, można zauważyć liczne niekonsekwencje odnośnie do używanych pojęć, określeń i nazw. Po części jest to efektem różnego tłumaczenia określeń angielskich, po części niedbałością, a czasem niewiedzą. Poszczególni autorzy mają też różną umiejętność tłumaczenia spraw trudnych w sposób przystępny. Pół biedy, gdy opis dotyczy spraw ogólnych. Gorzej, gdy początkujący zupełnie nie rozumie znaczenia najważniejszych określeń i pojęć języka C albo gdy rozumie je błędnie. Dlatego spróbujmy uściślić przynajmniej niektóre szczegóły dotyczące elementów, z których składają się programy, takich jak: ***operator***, ***operand***, ***argument***, ***wyrażenie***, ***instrukcja***, ***rozkaz***, ***polecenie***, ***deklaracja***, ***dyrektywa***, ***blok***, ***procedura*** i ***funkcja***.

Nie będziemy jednak głęboko wchodzić w zagadnienia językoznawstwa, bo przecież... my chcemy tylko programować mikrokontrolery AVR. Spróbujemy więc podejść do sprawy w sposób uproszczony. Przypominam też, że ten artykuł i następne „o krokodylu” przeznaczone są tylko dla dociekliwych, którzy chcą rozumieć wszystko, co robią.

Piotr Górecki