Wokół języka C

Nie bój się paradygmatów – rób swoje, byle dobrze...

Przy okazji dyskusji na temat kursu C potwierdziło się, że część Czytelników, głównie tych młodych, chciałaby otrzymać wszystko „podane gotowe na tacy". Chcą błyskawicznie i bez wysiłku nauczyć się programowania mikrokontrolerów. A ściślej mówiąc, chcą, żeby ktoś błyskawicznie ich tego nauczył. Natomiast u Czytelników starszych wiekiem zwykle głównym problemem jest obawa przed ogromem materiału oraz przed licznymi nowymi pojęciami i zagadnieniami.

W poprzednich odcinkach EdW zamieszczone były materiały, które miały ośmielić i przynajmniej częściowo likwidować obawy związane z programowaniem. Słusznie, ponieważ strach i obawy to dla wielu największe przeszkody. Wszystko jest dla ludzi i praktycznie **każdy może nauczyć się programowania**. Tak, ale... przypomnijmy wypowiedź przedrewolucyjnego rosyjskiego pisarza Lwa Tołstoja: ***Wiedza daje pokorę wielkiemu, dziwi przeciętnego, nadyma małego. Nic tak nie ogranicza prawdziwej wiedzy, jak przekonanie, że się wie to, czego się nie wie***.

Sentencję tę warto przemyśleć w kontekście programowania. Owszem, należy zachęcać do nauki programowania, ale nie można przemilczać pewnych ważnych zagadnień. Informatyka i programowanie to dziś nieprawdopodobnie rozległe dziedziny. **Aby zostać dobrym programistą, trzeba zdobyć bardzo dużo wiedzy**. Trzeba w to włożyć dużo wysiłku i poświęcić dużo czasu. A tak naprawdę, to ***nauka programowania nigdy się nie kończy***...

Uściślijmy: każdy może się nauczyć tworzenia programów, ale bardzo często będą to programy „byle jakie", nieoptymalne. Przy odrobinie szczęścia (dobry nauczyciel, dobra literatura, pomoc kolegów) podstawy programowania można opanować zaskakująco szybko i prosto. Jednak nie trzeba chyba nikogo przekonywać, że program realizujący dane zadanie można zrealizować na wiele sposobów. Są programy lepsze i gorsze (a także dużo lepsze i dużo gorsze). Te gorsze i dużo gorsze też działają, ale mają różne wady, często poważne, niektóre ujawniające się po dłuższym czasie.

Napisanie „dobrego programu" nie jest łatwe, nie sposób bowiem poznać „całej informatyki", podobnie jak nie można poznać „całej elektroniki". Przywołajmy tu bliski nam przykład początkującego elektronika.

Niejeden początkujący zaczyna przygodę z elektroniką od składania gotowych zestawów, zwanych powszechnie kitami oraz od jakiejś książki czy kursu ***Oślej łączki*** lub ***PKE***. Gdy zechce wzbogacić swój dom o jakieś nietypowe urządzenie elektroniczne, jego praca będzie polegać na znalezieniu i zakupie odpowiedniego kitu, a potem polutowaniu elementów na płytce drukowanej. Zapewne doda do tego jakiś zasilacz i ewentualnie prowizorycznie dolutuje do płytki „w pająku" jakieś proste obwody pomocnicze. Powstały układ upchnie w jakiejś przypadkowej obudowie, oczywiście z użyciem „kleju na gorąco" i... będzie ogromnie cieszył się z sukcesu! Najbliższe otoczenie może uważać go za elektronika, nawet bardzo dobrego elektronika, który „sam lutuje układy"!

A tymczasem jego wiedza elektroniczna tak naprawdę jest bliska zeru, nieporównywalnie mniejsza niż u doświadczonego elektronika konstruktora, który od kilkudziesięciu lat zdobywa doświadczenie. Taki konstruktor, gdy zechce stworzyć jakiś układ, zacznie od analizy potrzeb, określenia potrzeb, wybrania eleganckiej obudowy. Potem rozważy opcje zasilania, przemyśli kilka wersji schematu i wybierze optymalną. Następnie zaprojektuje płytkę dla wcześniej wybranej obudowy, sam tę płytkę wykona i zamontuje na niej elementy.

Który z tych dwóch elektroników będzie miał większą satysfakcję ze swojej pracy?

Być może ten początkujący...

A przecież obiektywnie biorąc, w porównaniu z doświadczonym konstruktorem, jego osiągnięcia i umiejętności w dziedzinie elektroniki są być może żenująco małe.

Dokładnie tak samo jest z programowaniem procesorów (i komputerów). Dla jednego wielkim sukcesem będzie „sklecenie" programu, byle tylko działał. Natomiast dla doświadczonego programisty sukcesem będzie napisanie króciutkiego, przejrzystego, eleganckiego programu, który będzie można w przyszłości łatwo modyfikować.

Trzeba przyznać, że dla wielu elektroników ogromnym sukcesem jest sam fakt, że mikroprocesor w ich układzie w ogóle działa. Nie zastanawiają się nad żadną optymalizacją programu, zresztą nie potrafiliby takiej optymalizacji wykonać. Często napotykają na problem zbyt małej pamięci mikroprocesora, bo ich programy „wychodzą" zaskakująco duże.

Oddając sprawiedliwość elektronikom, muszę wspomnieć, że często bywa inaczej: dobry informatyk napisze zwięzły, elegancki program dla procesora, ale realizacja obwodów elektronicznych współpracujących z tym mikroprocesorem *woła o pomstę do nieba*, co też okazuje się źródłem różnych błędów w działaniu systemu.

Podstawowy wniosek jest prosty: *aby zostać **dobrym** programistą (a także dobrym elektronikiem), trzeba zdobyć naprawdę dużo wiedzy, a to wymaga mnóstwa czasu, mnóstwa praktycznych doświadczeń i... porażek*.

Oczywiście można czerpać wiele radości i głębokiej satysfakcji z tego, że program w ogóle działa! Trzeba jednak mieć świadomość, że program jest, najdelikatniej mówiąc, „byle jaki".

Doświadczony programista napisze program, który po pierwsze *zużywa minimalną ilość zasobów procesora* (a konkretnie jest wykonywany szybko i zajmuje mało pamięci, co pozwala wykorzystać tańszy procesor), a po drugie program źródłowy jest *przejrzyście napisany*, *dobrze udokumentowany* komentarzami i w razie potrzeby *można go zmodyfikować* czy inaczej wykorzystać w sposób dziecinnie łatwy.

Dobry program to same korzyści! Ale to jest ideał – cel, do którego należy dążyć. Tymczasem wielu praktycznym programom bardzo daleko do ideału. Warto o tym pamiętać, bowiem z jednej strony jest to zachętą, żeby bez stresu i kompleksów zacząć i cieszyć się nawet kiepskimi programami. Ale z drugiej strony jest to mocna zachęta, żeby wciąż się wysilać, uczyć, robić postępy i zbliżać do ideału.

Niestety, wielu elektroników wpadło w pułapkę. Nauczyli się pisać programy

„byle jak” i utknęli w tym na stałe. Dotyczy to głównie trzymania się skądinąd znakomitego i popularnego BASCOM-a. Jeżeli programujesz w BASCOM-ie i jesteś z tego zadowolony, przerwij lekturę w tym miejscu i nie czytaj dalej. Nie chciałbym zburzyć Twej radości i straszyć strasznymi, tytułowymi paradygmatami.

Jeżeli jednak wykorzystując BASCOM-a, masz niedosyt albo też nie znasz żadnego języka programowania – czytaj dalej. Nawet jeśli nie nauczysz się C, przedstawione rozważania wzbogacą Cię.

Chwyćmy więc byka za rogi!

Te straszne paradygmaty

Informatyka to obecnie ogromnie obszerna dziedzina. Nietrudno zgubić się, nie tylko z uwagi na ogrom dostępnej wiedzy, ale też ma specyficzną terminologię. Spróbujmy oswoić kilka pojęć i ważnych zagadnień (w tym te straszące wielu ***paradygmaty***).

Zacznijmy od czegoś oczywistego: w informatyce wprowadzono mnóstwo nowych pojęć i określeń. Ich zdecydowana większość dotyczy zagadnień łatwych do zrozumienia, tylko używane określenia straszą, i to nie tylko początkujących. Na początku nauki programowania ważnym zadaniem jest, by te straszące nazwy/określenia wytłumaczyć „ludzkim językiem” i zobrazować analogiami, z którymi spotykamy się w codziennym życiu.

Trzeba jednak przyznać, że wraz z burzliwym i gwałtownym rozwojem informatyki pojawiły się też specyficzne problemy i zagadnienia, których nie sposób zobrazować czy wytłumaczyć analogiami z życia codziennego. Po prostu nie ma odpowiednich analogii.

Normalne, a wręcz oczywiste jest, że na zrozumienie tych trudniejszych, abstrakcyjnych kwestii trzeba przygotowania, czasu i doświadczenia.

Nie przejmuj się tym, że niektórych zagadnień od razu nie zrozumiesz. Ale też miej świadomość, że jeśli chcesz robić postęp, powinieneś stopniowo „wgryzać się” także w trudniejsze zagadnienia. Takim postępem jest przejście z BASCOM-a na język C (lub rozpoczęcie nauki od języka C).

Jeżeli chcesz zrozumieć i docenić zalety języka C, musimy wspomnieć o ***paradygmatach***. Określenie to być może straszy, ale nam potrzebne jest tylko po to, żeby... polubić język C. Jeśli koniecznie chcesz, poszukaj definicji paradygmatu, ale może lepiej tego nie rób...

Ostrzegałem!

Najprościej biorąc, *paradygmat* to w naszym przypadku *sposób, metoda, wzorzec, rozwiązanie modelowe, powszechnie uznany sposób działania*. Zanim jednak dojdziemy do sedna sprawy, przypomnijmy, że jeszcze w latach 50. XX wieku komputery były wielce kosztowną rzadkością, dostępną tylko w laboratoriach wojskowych i na najlepszych uniwersytetach. Koszt sprzętu był ogromny. Samo programowanie było ściśle związane ze sprzętem (kod maszynowy), a programowaniem zajmowali się nieliczni naukowcy. Koszt tworzenia oprogramowania był drobnym ułamkiem ceny komputera.

Z biegiem lat komputery stawały się coraz tańsze, bardziej dostępne. Z uwagi na poważne wady programowania w kodzie maszynowym szybko powstawał asembler, a właściwie liczne asemblery, a potem kolejne języki programowania. Asembler to język ściśle związany z danym komputerem (procesorem). Specjaliści od kodu maszynowego i asemblera byli ściśle związani z konkretnym sprzętem. Dużo lepsze okazywały się języki programowania wyższego poziomu, niezależne od sprzętu. Używający ich programiści nie musieli już znać szczegółów budowy sprzętu i mogli się zająć innymi aspektami procesu tworzenia programów. I zaczęło się...

Gwałtownie zaczęła się rozwijać nowa dziedzina: informatyka. Najkrócej mówiąc, *informatyka to nauka o przetwarzaniu informacji*. Z czasem informatyka i programowanie zupełnie oddzieliły się od szczegółów budowy sprzętu. W grę wchodziło wiele aspektów. Wspomnijmy o jednym. Otóż koszty pracy programistów najpierw zrównały się z cenami sprzętu, potem okazały się od nich wyższe. Choćby tylko z uwagi na konkurencję rynkową, trzeba było redukować koszty tworzenia oprogramowania. Tworzono więc programy, które ułatwiały tworzenie nowych programów. Powstawały liczne nowe języki programowania. Programy stały się towarem, który trzeba było wytwarzać szybko, tanio, nie zapominając o w miarę przyzwoitej jakości.

Zgodnie z zasadą, że całość to coś więcej niż suma części, w trakcie rozwoju informatyki ujawniały się zupełnie nowe zagadnienia, którymi nikt wcześniej się nie zajmował. Oczywiście nie był to zaplanowany rozwój, co dziś z perspektywy czasu widzimy bardzo jasno. Był to nie tylko gwałtowny rozwój, ale właśnie tworzenie zupełnie nowych pojęć i zupełnie nowych całych obszernych dziedzin. Często związane było to z konkretnymi potrzebami i problemami, wywołującymi żywe dyskusje. Do dziś w literaturze przywołuje się kontrowersyjną instrukcję ***goto*** (go-to = idź do...), będącą prostym, wygodnym rozkazem skoku do innego miejsca w programie, czyli do określonej komórki pamięci. Instrukcja ta występowała we wszystkich wczesnych językach programowania i była dawniej często używana. Z czasem praktyka pokazała, że programy, gdzie często pojawiała się instrukcja *goto*, były trudne do analizy i modyfikacji. Co jeszcze ważniejsze, trudno było wykorzystać fragmenty takich programów do tworzenia innych programów.

A problem powtórnego wykorzystywania wcześniej napisanych fragmentów narastał. W pierwszych komputerach, a później w pierwszych mikroprocesorach programy składały się z kilkuset, a co najwyżej kilku tysięcy elementarnych rozkazów. Nad takiej wielkości programem można względnie łatwo zapanować. Nie trzeba stosować specjalnych reguł przy tworzeniu niedużego programu. Można stosować tak zwane *programowanie liniowe*. Czyli można napisać program, który niejako według (w miarę prostej) linii zrealizuje dane zadanie. Z czasem kod takich programów, niczym poplątany włoski makaron, nazwano *spaghetti code*. Kod, poplątany często przez obecność rozkazów skoku *goto*, jest trudny do analizy, ale w przypadku małych programów nie jest to poważnym problemem.

Natomiast problem narasta, gdy programy stają się coraz większe. A programy mają realizować coraz bardziej złożone zadania i z konieczności stają się coraz większe. Absolutnie niezbędne staje się odciążenie programisty, żeby nie musiał pamiętać o wszystkich drobnych szczegółach. Trzeba dane zadanie, a tym samym realizujący je program niejako rozłożyć na „mniejsze kawałki”. Jeżeli te „mniejsze kawałki” nazwiemy procedurami, to łatwiej będzie się zorientować w gąszczu rozkazów. Możemy mówić o *programowaniu proceduralnym*. Wyodrębnienie procedur, które realizują określone zadania, niewątpliwie zwiększa przejrzystość, ułatwia analizę i zmiany.

Kolejnym krokiem jest wyodrębnienie w dużych programach struktury. Skomplikowane zadanie praktycznie zawsze możemy rozdzielić na mniejsze „podzadania”. Te mniejsze „podzadania” często można rozłożyć na jeszcze mniejsze „mikrozadania”.

Po takim podziale okaże się, że mamy mnóstwo prostych procedur, z których każda wykona tylko jakieś drobne zadanie, a ściślej biorąc – spełni, wykona tylko jedną, określoną funkcję. A odpowiednia realizacja szeregu tych prostych procedur-funkcji pozwoli wykonać bardzo skomplikowane zadanie.

A program może być napisany tak, żeby łatwo dostrzec jego strukturę i sposób korzystania z tych większych i mniejszych funkcji – będziemy wtedy mówić o *programowaniu strukturalnym*.

Kluczowe znaczenie ma tu podział zadania na drobne, autonomiczne procedury, z których każda realizuje tylko jedną, konkretną funkcję. Podkreślmy, że takie autonomiczne „drobne procedury" nazywane są ***funkcjami***. Oczywiście dana funkcja składa się z szeregu elementarnych instrukcji kodu maszynowego procesora.

Funkcja to „odrębny kawałek". Funkcja ma wykonywać ***jedną*** czynność (realizuje jedną funkcję), a potrzebuje do tego zwykle danych wejściowych (zwanych *argumentami*) i zwykle funkcja daje (fachowo: *zwraca*) wynik swojego działania.

Funkcję można porównać do skrzynki, niekoniecznie czarnej: coś do niej wkładamy (*przekazujemy argumenty do funkcji*), każemy coś zrealizować (*wywołujemy funkcję*) i otrzymujemy jakiś wynik (*funkcja zwraca wartość*). „Skrzynki" mogą być malutkie i realizować bardzo proste zadania. Ale „skrzynki" mogą też być większe i dużo większe: w swym wnętrzu mogą zawierać liczne mniejsze „skrzynki".

Oczywiście w danym programie jedne funkcje będą wykorzystywane dużo częściej niż inne. Niektóre funkcje będą niejako ze sobą spokrewnione lub w jakiś sposób ściśle związane. Zamiast wykorzystywać mnóstwo „pokrewnych i związanych", ale pojedynczych funkcji, można wykorzystać inną koncepcję: coś jakby „wielofunkcje", zwane *klasami* oraz *obiektami*. Wtedy mówimy o *programowaniu obiektowym*. Pojęcia i terminy związane z programowaniem obiektowym (*klasy, instancje, metody, hermetyzacja, dziedziczenie...*) mogą straszyć, ale w sumie idea jest prosta: chodzi o podział programu na odrębne „dość duże kawałki". W przypadku funkcji mamy jedną czynność, jeden wynik i jeden lub kilka wejściowych argumentów. Natomiast można powiedzieć, że w programowaniu obiektowym *obiekt* to coś więcej niż „multifunkcja". To autonomiczny, „oddzielony od otoczenia" spory podprogram. *Obiekt* na życzenie może realizować różne (wcześniej określone) zadania, które nazywamy nie funkcjami, tylko *metodami*. Do funkcji przekazywaliśmy dane, zwane argumentami, natomiast „dane wejściowe" dla obiektu są nazywane *polami*. Można powiedzieć, że językiem obiektowym jest język C++, który jako następca i rozwinięcie języka C dodaje doń bardzo wygodną możliwość korzystania z obiektów.

Wbrew pozorom nie jest to krok wstecz w kierunku programowania liniowego, tylko sposób programowania odpowiadający naszemu, ludzkiemu postrzeganiu rzeczywistości. Cały czas chodzi o podział „dużych zadań" na łatwe do wielokrotnego użycia „mniejsze, autonomiczne kawałki", co ułatwi prace programiście.

W następnym kroku możemy też mówić o *programowaniu wizualnym*. Wiąże się z faktem, że od dawna przy obsłudze komputerów do obsługi aplikacji graficznych używamy myszki (aktualnie także ekranów dotykowych). Mówiąc w dużym skrócie, przy programowaniu wizualnym wykorzystujemy obiekty, a do ich sterowania myszkę i jej wskaźnik (lub odpowiedniki).

Wymieniliśmy tu kilka *sposobów*, *metod*, *koncepcji* programowania, czyli przedstawiliśmy różne...
paradygmaty programowania.

A teraz do naszej praktyki: mówiąc najprościej, język BASIC i popularny wśród elektroników BASCOM sprzyjają programowaniu liniowemu, co bardzo często prowadzi do powstania bałaganiarskiego *spaghetti code*. Natomiast język C sprzyja programowaniu strukturalnemu i proceduralnemu, co pozwala łatwo uzyskać przejrzysty kod.

Mówimy *sprzyja*, ponieważ także w BASCOM-ie można podzielić program na procedury i de facto zrealizować programowanie strukturalne. Z drugiej strony także w języku C można napisać bałaganiarski kod „w jednym kawałku", na dodatek najeżony instrukcjami *goto*, czyli stworzyć wyjątkowo paskudny *spaghetti code*.

W grę wchodzi szereg czynników, ale nie będziemy wgłębiać się w kwestie *złożoności cyklomatycznej* czy problemu *wycieku zasobów*. Zasygnalizuję, i to w ogromnym uproszczeniu, tylko jedno podstawowe zagadnienie:
dlaczego tak ważne jest zagadnienie „podziału programu na kawałki"?

Zagadnienie to jest bardzo szerokie i poważne – my omawiamy je w ogromnym uproszczeniu.

„Dobre kawałki"

Zasadniczo nawet pisząc w asemblerze, można wykorzystywać „kawałki" – fragmenty wcześniej napisanych programów, które realizować będą poszczególne części większego zadania. Ale „składając kawałki" w asemblerze, trzeba dopilnować wszystkich szczegółów. Trzeba zadbać, żeby przy łączeniu programu z takich „kawałków" nie wystąpiły jakiekolwiek konflikty. Nie będziemy wchodzić w szczegóły, ale właśnie uniknięcie błędów jest podstawowym i niewdzięcznym zadaniem przy łączeniu programu „z kawałków".

Przewidując kłopoty, można już wcześniej tak przygotować „standardowe kawałki", by ryzyko konfliktów i błędów zminimalizować. Można mianowicie ***dążyć do tego, żeby takie „standardowe kawałki" były niezależne od reszty programu***.

Najprostszym rozwiązaniem wydaje się pomysł, by programista zapisywał oddzielnie swoje „kawałki programów" w postaci swego rodzaju biblioteki lub bibliotek, a potem pisząc kolejne programy, korzystał z tak przygotowanych zbiorów – zapasów.

Pomysł jest bardzo dobry.

Warto nawet go rozszerzyć: takie biblioteki nie muszą się ograniczać do jednego programisty. Można przygotować ogólnodostępne „standardowe biblioteki", które będą służyć wszystkim programistom. I takie „standardowe biblioteki" istnieją i są wykorzystywane, zwłaszcza w języku C. W języku C „kawałkami" są *funkcje* (w C++ dochodzą do tego *klasy* i *obiekty*, co możemy wykorzystywać w przypadku Arduino).

W sumie chodzi o to, żeby ułatwić zadanie programiście. Żeby nie musiał on całego programu żmudnie pisać od początku, tylko żeby bez kłopotów korzystał z funkcji które sam lub ktoś inny wcześniej napisał, w tym z licznych standardowych funkcji bibliotecznych, dostarczanych wraz z kompilatorem C.

Programowanie w C powinno więc jak najbardziej przypominać budowanie z rozmaitych funkcji – klocków czy raczej wspomnianych skrzynek. Język C nie tylko zachęca o wykorzystywania „skrzynek" – funkcji, ale wręcz narzuca konieczność użycia głównej, największej skrzynki: jest to funkcja ***main*** (główna), którą musi zawierać każdy program pisany w C.

W funkcji *main*, jak w dużym pojemniku, można byłoby od razu umieszczać wszystkie inne drobniejsze funkcje, które finalnie posłużą do wykonania danego zadania. Ostatecznie zrobi to kompilator, który odpowiednio zestawi wszystkie potrzebne funkcje-kawałki i stworzy finalny program w kodzie maszynowym. Ale podczas pisania programu źródłowego w języku C bardzo wygodne jest to, że można i warto wykorzystać funkcje, zawarte w licznych bibliotekach, ale nie umieszczać ich wprost w głównej funkcji *main*, tylko informować kompilator, żeby w razie potrzeby z nich skorzystał. Omówimy to za miesiąc, a oprócz funkcji omówimy też ***zmienne*** oraz związane z nimi ***typy danych***. Wtedy wszystko jeszcze bardziej ułoży się w logiczną całość i ujawni się piękno języka C.

Piotr Górecki