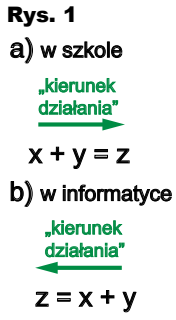
Wokół języka C
Piękno (?) języka C – funkcje

Próbuję pokazać Ci niewątpliwe piękno języka C, ale faktem jest, że język ten jest wyjątkowo nieprzyjazny dla początkujących, dlatego obrazkiem tytułowym jest szklanka, do połowy... pełna albo pusta, zależnie od Twojego nastawienia.

Zanim omówimy funkcje, podam garść informacji wstępnych, niezbyt wyczerpujących (a może nawet zbyt mocno uproszczonych):

1. Na lekcjach matematyki przyzwyczajani jesteśmy do zapisu „od lewej do prawej”, jak pokazuje **rysunek 1a**. Natomiast ***w języku C, jak generalnie w informatyce, wykorzystuje się „odwrotny sposób zapisu”*** (od prawej do lewej), co ilustruje **rysunek 1b**.

Rys. 1



2. Wcześniej podkreślaliśmy zalety podziału programu na „mniejsze kawałki” (funkcje), które można porównać do skrzynek, niekoniecznie czarnych, do których coś wkładamy (*przekazujemy argumenty*) i otrzymujemy jakiś wynik (*zwracaną wartość*). Zwróć uwagę, że cały program też możemy potraktować jak taką skrzynkę: wprowadzamy jakieś dane wejściowe, program je przetwarza i otrzymujemy jakieś dane wyjściowe. Jako podobne „skrzynki” możemy też potraktować „mniejsze elementy” języka C. Ilustruje to **rysunek 2**. Do tego jeszcze będziemy wracać, a na razie najważniejsze:

- ***do „skrzynki” zawsze wkładane są jakieś dane (przekazywane są argumenty)***
- ***„skrzynka” zawsze daje wynik działania (zwraca jakąś wartość).***

W języku C operujemy wyłącznie na liczbach. Napisy-litery też są traktowane jak liczby, a konkretnie jak kody ASCII. Także wartości logiczne (prawda/fałsz) reprezentowane są przez liczby. Mianowicie logiczny „fałsz” to liczba zero, natomiast logiczna „prawda” to nie tylko liczba 1, ale też (co na początku może wydawać się dziwne) każda inna liczba różna od zera. Czyli do „skrzynki wkładamy liczby” i „skrzynka jako wynik działania też zwraca liczbę”.

3. Instrukcja, **blok**. ***Instrukcja*** słusznie kojarzy się ze: *zrób coś* lub *przeprowadź jakąś operację*. Ogólnie biorąc, *instrukcje w języku C zakończone są znakiem średnika (;)*. Najprostsza instrukcja to sam średnik, co możemy rozumieć jako... *zrób nic*. W programie zawsze występuje mnóstwo instrukcji, zazwyczaj krótkich. Na przykład instrukcja `a=3;` powoduje wpisanie liczby trzy do zmiennej **a** (*przypisanie wartości* zmiennej **a**). W jednej linii programu można umieścić wiele „pojedynczych” instrukcji (każda zakończona średnikiem), ale dla przejrzystości kodu często pisze się każdą instrukcję w oddzielnym wierszu.

Jeżeli zestaw instrukcji tworzy jakąś całość, służącą określonemu celowi, można ją potraktować jako jedną *instrukcję złożoną*, jako tak zwany *blok*. ***Blok*** *jest grupą instrukcji, która jest traktowana jak całość*. Blok zawiera się pomiędzy nawiasami klamrowymi { }. Blok może także zawierać kolejne (zagnieżdżone) bloki, oczywiście też ujęte w klamry { }. Blok to nie tylko twór, ułatwiający człowiekowi analizę programu i jego „kawałków”: co bardzo ważne, ***zmienne zdefiniowane na początku danego bloku to zmienne lokalne „widoczne” jedynie w obrębie tego bloku*** (i bloków w nim zawartych) i ***dostępne tylko w czasie realizacji instrukcji tego bloku***. Czyli jeżeli mamy dwa odrębne, niezależne bloki, to zawarte w nich dwie zmienne o tych samych nazwach „nie pogryzą się”, bo każda jest widoczna tylko w obrębie „swojego” bloku. Wyposażeni w taką podstawową wiedzę, możemy omówić funkcje.

Funkcje

Dany program może składać się z mnóstwa ***funkcji***, które są „autonomicznymi, gotowymi kawałkami kodu z nazwą własną” – identyfikatorem. Co ważne, w programie możesz (i czy chcesz, czy nie, i tak będziesz) wykorzystywać jak najwięcej funkcji, które już wcześniej napisałeś Ty lub (częściej) które napisał ktoś inny. Czyli z założenia twój program będzie złożeniem mnóstwa wcześniej przygotowanych „gotowych kawałków” oraz „głównego kawałka”, w którym opiszesz, co Twój program ma zrobić. Cały problem w tym, „żeby te kawałki się

Rys. 2

nie pogryzły”, tylko ze sobą harmonijnie współpracowały. I właśnie reguły języka C znakomicie sprzyjają takiej harmonijnej współpracy poszczególnych funkcji składowych, ponieważ ***funkcja to blok (blok instrukcji) o określonych właściwościach i nazwie***. Funkcja to instrukcja złożona, ale o właściwościach „autonomicznej skrzynki”. Można byłoby powiedzieć, że *funkcja to „blok, który ma nazwę oraz wejście i wyjście”*.

Funkcje porównaliśmy do większych i mniejszych skrzynek-pudełek. Do pudełka coś wkładamy i z pudełka „wyskakuje wynik”. Mówiąc językiem informatyków, ***wywołujemy funkcję***, jednocześnie ***przekazując do niej argumenty***, a ***funkcja zwraca wynik*** swego działania.

Ale zanim użyjemy takiej czy innej funkcji, należałoby tę funkcję w programie ***zadeklarować***, to znaczy poinformować kompilator, że będziemy używać funkcji o danej nazwie (identyfikatorze), że funkcja ta potrzebuje argumentów określonego typu i że zwróci wartość określonego typu. Najogólniej biorąc, ***deklaracja*** (ze średnikiem na końcu), nazywana czasem *prototypem funkcji*, ma postać:

```
zwracany_wynik nazwa_funkcji (argumenty);
```

Zwróć uwagę, że podobnie jak na rysunku 1b z prawej strony mamy (w nawiasie) argumenty, czyli „dane wejściowe”, a z lewej zwracany wynik.

Pamiętaj też, że w *deklaracji* chodzi tylko o ***nazwę*** – *identyfikator* oraz o ***typy danych*** argumentów i zwracanego „wyniku”, by kompilator pilnował, czy prawidłowe są wszystkie argumenty i przetwarzane wyniki. Deklaracja (podanie *prototypu*) funkcji ułatwia więc „pilnowanie porządku”, ale nie mówi, co robi

```
void   nazwa_funkcji   (typ argument1, typ argument2, ... typ argumentN)
{
  /* zrób coś      */
}
```

dana funkcja. W języku C określamy to oddzielnie w ***definicji funkcji***, która najogólniej biorąc, zawiera ***nagłówek*** taki jak w deklaracji (*prototyp*, bez średnika) i do tego w nawiasach klamrowych „opis działania”, czyli ***ciało funkcji***. Podstawowy szkielet funkcji wygląda tak:

```
typ_zwracanego_wyniku nazwa_funkcji (typ argument1, typ argument2, ... typ argumentN)
{
/* tu jest blok, grupa instrukcji traktowana jako całość
ale mówimy, że to jest ciało funkcji
blok może zawierać inne bloki w nawiasach { }        */
}
```

W praktyce pisząc własny program, nie musimy *deklarować* funkcji, bowiem nagłówek *definicji* funkcji będzie od razu jej *deklaracją*. By „uspokoić kompilator”, trzeba natomiast wcześniej deklarować te funkcje, o których kompilator nic nie wie, na przykład które zostaną „dołączone z zewnątrz”. Oto przykład definicji i jednocześnie deklaracji funkcji:

```
int   mnozenie   (char a, char b)
{
int iloczyn; //deklarujemy zmienną lokalną
iloczyn = a*b; //instrukcja mnożenia
return iloczyn; //funkcja zwraca wynik
}
```

Definiujemy tu funkcję o nazwie **mnozenie**, do której będziemy przekazywać dwa argumenty (**a, b**), oba 8-bitowego typu ***char*** i która zwróci wynik typu ***int***. W ciele funkcji (w bloku w klamrach { }) najpierw definiujemy pomocniczą zmienną lokalną **iloczyn** typu *int*. W niej zostanie umieszczony wynik mnożenia **a*b**, ten wynik zostanie zwrócony (niejako na zewnątrz funkcji) instrukcją **return**, a wtedy zmienna lokalna **iloczyn** zostanie zlikwidowana.

Do ważnej kwestii, *co i gdzie jest zwracane,* jeszcze wrócimy. Na razie odnotujmy, że w języku C każda funkcja – „skrzynka” ma przyjmować argumenty i zwracać wynik. Ale niektóre funkcje nie zwracają wyniku (w innych językach są nazywane *procedurami*). Z kolei niektóre funkcje nie potrzebują żadnych argumentów. Jednak jak mówi bardzo stare, podobno rosyjskie przysłowie: *Ordnung muss sein* (*porządek musi być*): surowe piękno i porządek języka C polega też na tym, że mamy w nim wyłącznie *funkcje*, czyli „skrzynki z wejściem i wyjściem”. I właśnie żeby zachować tak przyjęty logiczny porządek skrzynek z wejściem i wyjściem, przewidziany jest „pusty typ danych”, nazywany ***void***. Jeśli funkcja (procedura) tak naprawdę nie zwraca niczego, to w języku C napiszemy:

A jeżeli dodatkowo nie przyjmuje (nie potrzebuje) żadnych argumentów, napiszemy:

```
void   nazwa_funkcji   (void)
{
/* zrób coś      */
}
```

Często spotyka się postać „z pustym nawiasem”: **typ nazwa_funkcji()**, a czasem postać bez podania typu - jedynie **nazwa_funkcji() - zwracanym typem bedzie** domyślnie *int*, jednak dla porządku i jasności warto podawać wszystkie omówione informacje i nie stosować takich skrótów.

A teraz kolejna ważna sprawa:

Main, czyli główna

W każdym programie C obowiązkowo musi być główna funkcja o nazwie **main**. Właśnie ***od niej rozpoczyna się praca programu***. Możesz spotkać ją w postaci „coś zwracającej”, np.:

```
int main (void)
{
/* ...tu kod...   */
return 0;
}
```

która zwraca (*return*) liczbę całkowitą 0, gdy... wszystko przebiegło dobrze, czyli gdy program został zrealizowany bezbłędnie. Ma to sens w komputerze, gdzie jest system operacyjny (program nadrzędny), natomiast w mikrokontrolerze program *main* „nie ma gdzie zwracać”, więc mógłby występować w postaci która niczego nie przyjmuje i nie zwraca:

```
void main (void) {
 /* kod */ }
```

Jednak najczęściej w programach dla mikrokontrolerów jednoukładowych też używamy postaci **int main (void)**. Kompilator „wie”, że nie ma gdzie zwrócić wartości, więc i jedną i drugą wersję zamieni na taki sam kod maszynowy.

I kolejna sprawa podstawowa. Wiele programów dla mikroprocesorów działa w nieskończonej pętli albo z wykorzystaniem pętli *while*:

```
void main (void) {
  /* instrukcje wstępne */
  while (1)
     {
     /* główny program */
     }
}
```

albo z użyciem pętli *for*:

```
void main (void) {
  /* instrukcje wstępne */
  for (;;)
     {
     /* główny program */
     }
}
```

Powiedzieliśmy tu krótko o *deklarowaniu* i *definiowaniu* funkcji, natomiast sprawę ich wykorzystania i zwracanych wartości omówimy oddzielnie.

Mam nadzieje, że jak na razie wszystko jest proste.

W sumie zrozumienie podstaw takich ogromnie ważnych zagadnień jak *typy danych*, *wskaźniki* i *funkcje* nie jest trudne. Przy nauce języka C trudniejsze okazuje się coś innego. Dlatego za miesiąc porozmawiamy o… krokodylu.

Piotr Górecki