Bascom czy C?

część 1

Programiści, podobnie jak użytkownicy komputerów, samochodów, aparatów fotograficznych itp. systematycznie toczą ze sobą spory o to, która z użytkowanych przez nich marek jest najlepsza. Wśród programistów największe emocje wzbudzają dyskusje na temat wyższości języka C/C++ nad Bascomem i vice versa. Często podawane argumenty typu „bo tak i już" nie wprowadzają nic nowego do dyskusji, a raczej jeszcze bardziej ją zaostrzają. Przekonanie o zaletach wykorzystywanego narzędzia nie dopuszcza myśli o ewentualnym zainteresowaniu się innymi. Spróbujmy zatem w miarę obiektywnie porównać wymienione wyżej języki programowania.

Próba obiektywnego przedstawienia tematu nie będzie łatwa, gdyż autor, jak większość programistów, zdecydowanie preferuje jeden z porównywanych języków programowania. Na czas pisania artykułu postanowił jednak zapanować nad emocjami. Zadanie nie jest łatwe, gdyż języki C i Bascom różnią się między sobą dość znacznie. Z założenia też były stworzone do zupełnie innych celów. To wymagania użytkowników spowodowały, że w pewnym zakresie zastosowań stały się porównywalne ze sobą.

Różnice w ogólnych założeniach

W niniejszym artykule w odniesieniu do Bascoma będzie używana nazwa Basic. Na początek popatrzmy na ogólne podobieństwa i różnice w ogólnych założeniach przyjętych dla kompilatorów obu omawianych języków. Bascom powstał specjalnie dla mikrokontrolerów '51 i AVR. Wszystkie dodatkowe słowa kluczowe, których nie ma w innych wersjach Basica (np. `lcd`), odnoszą się tylko do tych rodzin. C jako język jest zawsze taki sam (dla ścisłości - prawie zawsze, ale różnice są małe i na ogół dobrze udokumentowane), niezależnie od tego, czy piszemy dla mikrokontrolera '51, AVR, PIC czy x86. Oczywiście ten sam program napisany dla '51 nie zadziała bezpośrednio na x86. Język C nie zawiera w zasadzie żadnych elementów typowych dla sprzętu, na którym ma działać. Z każdym kompilatorem dostarczany jest natomiast zbiór funkcji w tzw. bibliotece standardowej. Są to funkcje, które dla każdej platformy nazywają się tak samo i robią to samo (zazwyczaj), lecz różnią się swoim kodem. Dlatego niezależnie od tego, czy wywołamy funkcję `printf("Hello, world")` w programie DOS-owym, czy też UNIX-owym, to na konsoli wyświetli nam się napis „Hello, world". Biblioteki standardowe powstawały głównie z myślą o „większych braciach" mikrokontrolerów, dlatego ich implementacje na mikrokontrolery mogą być trochę inne. W powyższym przykładzie program po skompilowaniu np. dla '51 najprawdopodobniej spowoduje wysłanie tekstu do portu szeregowego, ponieważ nie występuje w tym przypadku żadna konsola. Składnia języka pozostaje jednak taka sama dla różnych procesorów. I jest to jedna z największych zalet C. Ktoś, kto nauczy się Basica i zechce przesiąść się na inną platformę niż '51 i AVR, napotka ogromny problem - jego programy nie będą się kompilować. Będzie musiał nauczyć się nowego języka. Z językiem C sytuacja wygląda znacznie lepiej. Praktycznie na każdy mikrokontroler i procesor dostępny jest odpowiedni kompilator.

Polecenia kompilatora

Zajmiemy się teraz poleceniami wydawanymi kompilatorowi. Zarówno w Basicu, jak i w C mamy do czynienia z dwoma rodzajami poleceń - wykonywalnymi i niewykonywalnymi. Pierwsze z nich to elementy programu, które zostaną wykonane w urządzeniu docelowym (np. wywołania funkcji, operacje arytmetyczne itp.). Drugie to takie, które mają wpływ tylko na proces kompilacji. Można z nich ułożyć swego rodzaju „program", który mówi kompilatorowi, co i jak ma skompilować. W Basicu są to polecenia typu `$sim`, `$lib` i wiele innych, jednak większość z nich odnosi się wyłącznie do mikrokontrolerów. Ogromna większość tych poleceń to „pomost" między programem a ustawieniami kompilatora. Polecenia te są przydatne wtedy, gdy różne programy kompilujemy z różnymi opcjami, a także gdy chcemy te ustawienia szybko zmienić bez wchodzenia w różne menu. Język C także oferuje szereg podobnych opcji, jednak ze względu na to, że nie jest z góry powiedziane, na jaką maszynę program będzie kompilowany, odbywa się to w trochę odmienny sposób. Jest to zależne od stosowanych bibliotek, samego kompilatora itp. Dostępne są np. opcje dołączania bibliotek (odpowiednik `$lib`), jak i ustawiania wielu parametrów kompilatora oraz linkera z wnętrza kodu. Niektóre z nich różnią się, gdyż różne są docelowe maszyny. W obu językach występuje konstrukcja `#if #else #endif`. Działa ona bardzo podobnie. W C dostępne są konstrukcje `#define`, które przypominają `Alias` z Basica, jednak mają one większe możliwości. Dzięki aliasom można w Basicu przyporządkować jednej nazwie jakąś inną, co umożliwia wymienne ich stosowanie, na przykład:

```
Input Alias Portb
```

Dzięki temu linia:

```
A = Input
```

w rzeczywistości zostanie skompilowana jako:

```
A = Portb
```

W C można jednej nazwie przyporządkować w zasadzie dowolny tekst (niekoniecznie drugą nazwę):

```
#define DataIn Portb
#define Version 100023
```

Tekst określony przez `#define` może także przyjąć postać o wiele bardziej złożoną, nawet przypominającą funkcję, dlatego nazywany jest makrem np.:

```
#define MAX(a, b) ((a>b)?a:b)
```

Wyrażenie `(a>b)?a:b` jest bardzo charakterystyczne dla C i zostanie omówione później. Jest to przykład

przekazywania parametrów do makra (w tym przypadku: a i b).

Komentarze

Komentarze w Basicu to wszystkie znaki od pojedynczego ' (znak apostrofu) do końca danej linijki:

```
Dim A As Integer 'a to jest komentarz
```

W języku C jest podobnie, przy czym komentarz zaczyna się od dwóch ukośników:

```
int A; //a to jest komentarz
```

Ten typ komentarza wprowadzono w wersji C++. Wcześniej (w wersji C) obowiązywały komentarze blokowe: / * i */. Wszystko, co mieści się pomiędzy tymi parami znaków, jest komentarzem, niezależnie od tego, czy jest to jeden wyraz w pojedynczej linijce, czy też cały plik. Komentarze blokowe nie mogą się jednak zagnieżdżać.

Szkielet programu

Najwyższy czas zająć się samym szkieletem programu. W Basicu program ma budowę ciągłą. W C przyjęto inną koncepcję. Cały program jest podzielony na funkcje. Każdy program w C zaczyna się od wywołania funkcji main. W niektórych środowiskach może mieć ona inną nazwę - np. w systemie Windows jest to z reguły funkcja WinMain. Dla tych, którzy piszą programy na mikrokontrolery, zawsze będzie to funkcja main. Fragment kodu, którego zadaniem jest wywołanie funkcji main już podczas działania programu, jest dostosowany do architektury mikrokontrolera i zostaje wstawiony automatycznie przez kompilator.

Deklaracje zmiennych

Istotną różnicą pomiędzy omawianymi językami jest rozróżnianie wielkości liter. W języku C nazwy a i A oznaczają dwie różne zmienne, w Basicu jest to ta sama zmienna.

Porównajmy teraz deklaracje zmiennych, typów danych itp. W Basicu mamy 7 typów zmiennych:

Całkowite:

Bit - 1/8 bajtu, zakres liczb: [0...1]

Byte - 1 bajt, bez znaku, zakres liczb: [0...255]

Integer - 2 bajty, ze znakiem, zakres liczb: [-32768...32767]

Word - 2 bajty, bez znaku, zakres liczb: [0...65535]

Long - 4 bajty, ze znakiem, zakres liczb: [-2147483648...2147483647]

Zmiennoprzecinkowe:

Single - 4 bajty

Tekstowe:

String - maksymalnie 254 bajty, rozmiar zmiennej to długość tekstu w znakach plus 1 bajt.

W języku C wielkość zmiennych jest dostosowana do sprzętu, na którym chcemy uruchomić program. Podstawowy typ int ma rozmiar 2 bajtów na mikrokontrolerze, ale na platformach takich jak x86 int ze względów czysto praktycznych ma wielkość 4 bajtów. Są 2 podstawowe typy całkowite:

char - zawsze 1 bajt, zazwyczaj ze znakiem (w niektórych kompilatorach char domyślnie jest typem bez znaku)

int - na mikrokontrolerach zazwyczaj 2 bajty, ze znakiem

Można zmienić dany typ na typ ze znakiem lub bez znaku, dodając słowo kluczowe signed lub unsigned:

unsigned char - 1 bajt, bez znaku

signed char - 1 bajt, ze znakiem

signed int - 2 bajty, ze znakiem (równoznaczne z int)

unsigned int - 2 bajty, bez znaku

Dla typu int dostępne są także przedrostki short i long, które można łączyć z signed i unsigned.

short int - w przypadku mikrokontrolerów - równoznaczne z int (tam, gdzie int ma wielkość 4 bajtów, <short int> ma 2 bajty)

long int - 4 bajty, ze znakiem

unsigned long int - 4 bajty, bez znaku

Niektóre kompilatory (np. AVR-GCC) obsługują także typ 64-bitowy:

long long - 8 bajtów, ze znakiem;

unsigned long long - 8 bajtów, bez znaku.

Widać więc dość prosty sposób tworzenia typów zmiennych, w zależności od potrzeb - trzeba wiedzieć, ile bajtów ma mieć zmienna i czy ma to być liczba ze znakiem czy bez. Aby nie pisać za każdym razem np. unsigned int albo long int, można pisać po prostu unsigned albo long.

W C występują dwa typy liczb zmiennoprzecinkowych:

float - 4 bajty, pojedynczej precyzji (odpowiednik single z Basica)

double - 8 bajtów, podwójnej precyzji

Niektóre kompilatory języka C nastawione wyłącznie na mikrokontrolery mogą wprowadzać swoje własne rozszerzenia typów, np. dodając typ bit.

W Basicu występuje następująca składnia przy deklaracji zmiennych:

```
Dim nazwa_zmiennej As typ_zmiennej
```

Przykład:

```
Dim A as Byte
```

W C analogiczna deklaracja ma postać trochę krótszą:

```
typ_zmiennej zmienna;
```

Przykład:

```
char D;
```

W przypadku, gdy deklarujemy kilka zmiennych tego samego typu, w C można wymienić je po przecinku:

```
int A, B, C;
```

W Basicu natomiast trzeba napisać:

```
Dim A As Integer, B As Integer, C
As Integer
```

Ponadto w C dozwolone jest inicjowanie zmiennych podczas deklaracji:

```
float a=12;
```

Basic takiej możliwości nie daje wprost, ale wprowadza mechanizm, dzięki któremu kompilator wie, że jeżeli napotka w programie zmienną, której nazwa zaczyna się od jakiegoś konkretnego znaku, to ma ją automatycznie zadeklarować jako zmienną danego typu. Na przykład niech wszystkie (niezadeklarowane wcześniej) zmienne zaczynające się na literę „i" będą typu Integer:

```
Defint i
i = 123
i2 = 22
```

Możliwe jest przypisanie litery do typów Bit (Defbit), Byte (Defbyte), Integer (Defint), Word (Defword), Long (Deflng) i Single (Defsng).

Lokowanie zmiennych w pamięci

Basic pozwala nam wybrać fizyczne miejsce ulokowania zmiennej: w pamięci wewnętrznej, zewnętrznej lub EEPROM. Dostęp do pamięci EEPROM umożliwiają specjalne funkcje wprowadzane przez dany kompilator. W Basicu można nakazać kompilatorowi, pod jakim adresem pamięci zostanie umieszczona zmienna np.:

```
Dim W As Byte At &H100
```

Jeżeli podany adres (w tym przypadku 100 w zapisie heksadecymalnym) jest już zajęty, to zmienna ta zostanie umieszczona w następnym wolnym miejscu.

Basic umożliwia także przypisanie nazwy do danego, konkretnego miejsca w pamięci - niezależnie od tego, czy została tam ulokowana jakaś zmienna, czy nie:

```
Dim W As Byte At &H100 Overlay
```

Teraz odnosząc się do W jak do „normalnej" zmiennej, możemy czytać i zapisywać z i do pamięci rozpoczynającej się od adresu &H100. W powyższym przypadku przypisanie:

```
W = 67
```

będzie oznaczało: „do komórki o adresie &H100 wpisz liczbę 67". Będzie to tylko jeden bajt, gdyż taki jest rozmiar typu Byte. Jeśli zmienną W zadeklarujemy jako typ Word:

```
Dim W As Word At &H100 Overlay
```

to przypisanie

```
W = 0
```

spowoduje wyzerowanie bajtów pod adresami &H100 i &H101 - dokładnie tak, jakby były one zajmowane przez zmienną typu Word. Domyślnie taka zmienna zostaje ulokowana w wewnętrznej pamięci RAM. Można to zmienić dodając na końcu deklaracji słowo kluczowe Xram lub Eram, np.:

```
Dim W As Byte At &H100 Overlay Xram
```

Jest to jeden ze sposobów wykonywania operacji na określonych fragmentach pamięci. Jego wadą jest to, że podczas działania programu nie możemy zmienić adresu komórek przypisanych do naszej zmiennej. Istnieje także drugi sposób dostępu do pamięci RAM, który nie posiada powyższego ograniczenia - za pomocą instrukcji Inp i Out można zapisać i odczytać wartość do i z dowolnej komórki pamięci, portu itd. Dostępna jest także funkcja zwracająca adres zadeklarowanej zmiennej: Varptr.

W C mechanizmy dostępu do pamięci rozwiązano inaczej. Występują tu tak zwane wskaźniki. Są to po prostu zmienne, które przechowują określony adres. Jest to swego rodzaju „pomost" między programem a pamięcią. Wskaźniki łączą funkcjonalność instrukcji Inp, Out i Varptr oraz zmiennych deklarowanych z użyciem Overlay. Wskaźnik zawsze wskazuje na zmienną takiego typu, jaki jest podany w deklaracji:

```
int *wskaznik;
```

Powyżej jest przedstawiona deklaracja zmiennej wskaźnikowej wskazującej na 2 bajty. Wskaźnikowi można przypisać konkretny adres, np. 0x100:

```
wskaznik = (int*) 0x100;
```

Zapis (int*) oznacza: „to, co jest po prawej stronie zamień na typ int *, czyli wskaźnik na int".

Teraz wykonując np.:

```
*wskaznik = 0x1234;
```

przypiszemy wartość 0x12 bajtowi pod adresem 0x100 i wartość 0x34 bajtowi pod adresem 0x101, zupełnie tak, jakby zajmowała je zmienna 2-bajtowa typu int. Czasami przydatny jest wskaźnik na typ pusty - jest on uniwersalny, może wskazywać na wszystko (niekoniecznie zmienną jakiegoś konkretnego typu), na przykład na blok pamięci. Deklaruje się go tak:

```
void *wskaznik;
```

Tablice

W języku C pojęcie tablicy i wskaźnika są w zasadzie jednoznaczne. Deklaracja tablicy wygląda jednak nieco inaczej niż wskaźnika:

```
char tablica[10];   //deklaracja
      //tablicy 10-elementowej
```

To samo w Basicu:

```
Dim tablica(10) As Byte
```

W Basicu jesteśmy ograniczeni do jednego wymiaru. W C można deklarować tablice wielowymiarowe, np. tablicę liczb typu char o rozmiarach 10 na 10:

```
char tablica[10][10];
```

Teraz żeby odnieść się do pierwszego elementu tablicy w Basicu piszemy:

```
tablica(1) = 13
```

w języku C zapis wygląda następująco:

```
tablica[0] = 13;
```

a w przypadku tablicy wielowymiarowej:

```
tablica[0][0] = 13;
```

W C można przypisać poszczególnym elementom tablicy konkretne wartości podczas deklaracji:

```
char tablica[10] = {53, 62, 105,
99, 113, 24, 16, 95, 100, 42};
```

W Basicu trzeba wszystkim elementom tablicy przypisać te wartości osobno:

```
Dim tablica(10) As Byte
tablica(1) = 53
tablica(2) = 62
tablica(3) = 105
tablica(4) = 99
tablica(5) = 113
tablica(6) = 24
tablica(7) = 16
tablica(8) = 95
tablica(9) = 100
tablica(10) = 42
```

Zmienne tablicowe z punktu widzenia programisty zachowują się bardzo podobnie do wskaźników - tyle tylko, że tak naprawdę podczas działania programu nie będzie im przypisany ani jeden bajt pamięci. Gdybyśmy chcieli „podejrzeć" wartość zmiennej tablica, to okazałoby się, że jest to adres jej pierwszego elementu (o indeksie 0). Na przykład, jeżeli w wyniku wykonania instrukcji:

```
printf("tablica = 0x%p", tablica);
```

otrzymalibyśmy np. tekst „tablica = 0x24", oznaczałoby to, że adres pierwszego jej elementu jest równy 0x24. Dziesięć kolejnych bajtów pamięci począwszy od tego adresu zajmą elementy tablicy. Skoro można traktować zmienne tablicowe jak wskaźniki, prawidłowe będzie takie przypisanie:

```
*(tablica + 0) = 10;
```

Będzie ono równoznaczne ze „standardowym" odniesieniem do konkretnego elementu tablicy:

```
tablica[0] = 10;
```

Indeks, który podajemy w wyrażeniu tablica[0] = 10, jest tak naprawdę przesunięciem względem początku tablicy. Ponieważ przesunięcie od pierwszego elementu tablicy do jej początku jest równe zero, to pierwszy element ma indeks 0.

Wskaźniki dają ogromne możliwości (w tym artykule są one i tak potraktowane dość pobieżnie), stanowią one bowiem jedne z najpotężniejszych mechanizmów w C/C++ - szczególnie, gdy korzysta się np. ze struktur i klas oraz dynamicznego alokowania pamięci podczas działania programu.

Łańcuchy

Typ String (łańcuch) w Basicu to w zasadzie też nic innego jak tablica - przynajmniej biorąc pod uwagę sposób, w jakim tekst jest przechowywany w pamięci. W języku C początek łańcucha, czyli ciągu znaków, wskazuje odpowiedni wskaźnik, koniec zaś to bajt o wartości 0. Jak widać, jest to bardzo podobne rozwiązanie jak w Basicu. Wszystkie funkcje operujące na ciągach znaków potrzebują tylko jednego parametru - wskaźnika na jego pierwszy bajt. W języku C tekst deklarujemy następująco:

```
char *tekst = "Jakiś tekst";
```

lub zapisując inaczej:

```
char tekst[6] = "Hello"; /* musi
być miejsce na ostatni bajt o
wartości 0 */
```

czy też dając kompilatorowi zadanie policzenia wielkości tej tablicy:

```
char tekst[] = "Hello";
```

W Basicu zapiszemy:

```
Dim tekst As String * 5
tekst = "Hello"
```

W tym przykładzie String * 5 oznacza, że w zmiennej tekst można przechować 5 znaków. Tak naprawdę zmienna ta zajmuje 6 bajtów.

Kuba Klimkiewicz