Wokół języka C

Piękno (?) języka C – zmienne

W poprzednim numerze poruszaliśmy ważne kwestie dotyczące paradygmatów, czyli metod programowania i omawialiśmy zalety podziału programu na „autonomiczne kawałki”. Obiecałem pokazać Ci piękno języka C.

Wiem, że znajdzie się mnóstwo osób mniej i bardziej zaawansowanych, które (nie bez przyczyny) wątpią w piękno języka C. A może nie tylko wątpią, ale potrafią wskazać jego poważne wady. Właśnie dlatego obrazkiem tytułowym jest szklanka, do połowy...

...pełna czy pusta?

To zależy od punktu widzenia, a ściślej od Twojego nastawienia do sprawy.

Faktem jest, że język C jest wyjątkowo nieprzyjazny dla początkujących, m.in. dlatego, że ma rygorystyczne wymagania i nie wybacza błędów. To prawda, ale ma też ogromne zalety, które ujawniają się po nabraniu wprawy.

O ile pisząc programy dla komputerów można wybrać inne języki programowania, o tyle w przypadku mikroprocesorów AVR wybór jest skromny, w praktyce tylko: *asembler, BASCOM* i *język C*. Zachęcam więc, żebyś wykazał maksimum dobrej woli, by dostrzec i docenić zalety oraz surowe piękno języka C. Wtedy chętniej będziesz go poznawać i zgłębiać jego tajniki.

Zaletą jest, że w języku C operujemy wyłącznie na liczbach. Przetwarzane liczby mogą być wpisane w tekście programu (literały), ale mogą też być zawarte w ***zmiennych*** i w ***stałych***.

Zmienne

Doskonale wiesz, że w procesorach AVR program (w postaci kodu maszynowego) umieszczony jest na stałe w nieulotnej pamięci FLASH. Rozkazy programu operują natomiast na komórkach ulotnej pamięci danych RAM. Operują, to znaczy przeprowadzają obliczenia, odczytują i zapisują komórki. Jeżeli więc program ma przeprowadzać jakieś obliczenia (operacje), to żeby nie było bałaganu, *trzeba w pamięci RAM przygotować miejsce na dane*.

I właśnie **zmienna** to te „przygotowane w pamięci miejsce” dla jakiegoś rodzaju danych. W najprostszym przypadku zmienną może być jedna komórka pamięci RAM. Zwróć uwagę, że w czasie realizacji programu będzie wykorzystywana zarówno **zawartość** tej komórki pamięci (*wartość* zmiennej), jak też jej **adres** (*adres* zmiennej – *wskaźnik* do tej zmiennej). Podczas pisania programu w języku C nie musimy martwić się o adres i inne szczegóły, a tylko musimy poinformować kompilator, że w pisanym programie potrzebna będzie taka czy inna zmienna. W tym celu podajemy nazwę zmiennej (identyfikator), informujemy, jaki ma mieć rozmiar. W języku C dodatkowo możemy określić *czas życia* zmiennej oraz *widoczność* (to, jakie części programu będą ją „widzieć”). Tę zmienną, czyli miejsce w pamięci RAM na dane, „zaplanuje i przydzieli” kompilator.

Typy danych

Zmienna to ostatecznie fragment pamięci RAM procesora. Jedna komórka?

Niekoniecznie!

Jedna komórka RAM, czyli osiem bitów, daje 256 możliwości (kodów). Pomieści liczbę dwójkową z zakresu 0...255 albo liczbę ze znakiem (***signed***) z zakresu –128...+127 (w kodzie U2 – uzupełnienia do 2), albo jedną literę – znak (***char*** – *character*) zapisaną w kodzie ASCII. Do „większych” liczb potrzeba więcej miejsca.

W powszechnie wykorzystywanym kompilatorze języka C, zwanym AVR-GCC, przewidziano dla liczb całkowitych cztery „podstawowe rozmiary”:

char – 1 komórka RAM,
int – 2 komórki,
long – 4 komórki,
long long – 8 komórek RAM.

Zawartość może być traktowana jako liczba ze znakiem (***signed***) w kodzie U2 albo jako „zwykła” liczba dwójkowa bez znaku (***unsigned***). Zamiast kombinacji signed/unsigned z „rozmiarem”, zaleca się stosowanie w programach oznaczeń *int/uint* według **tabeli 1**.

Warto pamiętać, że popularny typ *char* jest domyślnie *signed*, czyli ze znakiem, w kodzie U2 (*int8_t*), gdzie najstarszy bit jest bitem znaku minus. Jednak w zakresie liczb 0...127 (dwójkowo 0…01111111) odpowiada *unsigned char* (*uint8_t*). To samo dotyczy typu *int* w zakresie 0...32767 oraz *long*.

To dotyczy liczb całkowitych. Dla liczb ułamkowych w kompilatorze AVR przewidziany jest 4-bajtowy typ zmiennoprzecinkowy (***float***) tzw. pojedynczej precyzji. Spośród 32 bitów jeden to bit znaku, 8 bitów wykładnika i 23 bity mantysy, co w zapisie dziesiętnym daje zakres $–3{,}4•10^{–38}$ … $+3{,}4•10^{38}$ i dokładność 6...7 miejsc dziesiętnych.

Wspomnę jeszcze, że niekiedy w programie podczas obliczeń trzeba stosować zamianę (*konwersję*) typów liczbowych, co nie jest żadnym problemem, ale nosi straszącą początkujących nazwę ***rzutowanie*** (*casting*).

W przeciwieństwie do wielu innych języków, zasadniczo C nie przewiduje ani jednobitowego typu logicznego *boolean* (ale można zrealizować samemu jego odpowiednik), ani oddzielnego typu dla napisów – liter. Dla pojedynczego znaku-litery wykorzystuje się 8-bitowy typ ***char***, w którym umieszczana jest liczba – kod ASCII. Dla napisów i do innych celów wykorzystuje się *tablice*. ***Tablica*** (*array*) *to ciąg zmiennych tego samego typu zajmujących ciągły obszar w pamięci,* np. kilkanaście kolejnych komórek pamięci.

Tablica to szereg komórek określonego *jednakowego* typu. Pokrewny typ danych to ***struktura*** (***struct***). To jakby „bogatsza tablica”: w jednej zmiennej typu *struct* można przechowywać wiele wartości o *różnych typach* (określonych wcześniej). Programista według potrzeb definiuje wielkość tablic i struktur oraz z jakiego typu zmiennych mają się składać. Nie będę Ci mącił w głowie tablicami wielowymiarowymi czy podobnymi do struktur *uniami* ani możliwością definiowania własnych typów danych (*typedef*).

Tabela 1

nazwa		zakres („pojemność”)
„stara”	zalecana	
char	int8_t	-128...127
unsigned char	uint8_t	0…255
int (short)	int16_t	-32768…32767
unsigned int	uint16_t	0…65535
long	int32_t	-2147483648...2147483647
unsigned long	uint32_t	0…4294967295
long long	int64_t	-9223372036854775808... 9223372036854775807
unsigned	uint64_t	0...18446744073709551615

(Nie)Straszne modyfikatory

Oprócz wielkości, w C mamy też dwa dodatkowe aspekty zmiennej: *czas życia* oraz *widzialność*, czyli dostępność zmiennej dla innych fragmentów programu.

Generalna zasada jest następująca: ponieważ większość zmiennych nie jest wykorzystywana przez cały czas pracy programu, miejsca w pamięci nie trzeba dla nich rezerwować „na stałe”. Jeśli są potrzebne tylko do realizacji określonego zadania (funkcji) – nazywane są ***zmiennymi lokalnymi***. Jednak niektóre zmienne mogą być potrzebne przez cały czas pracy programu. Wystarczy odpowiednio poinformować o tym kompilator, by stworzył tak zwaną ***zmienną globalną***: po pierwsze „trwałą”, po drugie dostępną dla wszystkich zadań (funkcji) całego programu.

Nie ma natomiast potrzeby rezerwowania „na stałe” miejsca dla wszystkich lokalnych zmiennych użytych w programie. Ogólnie biorąc, zmienne (lokalne) wymagane do realizacji konkretnej funkcji są według potrzeb „powoływane do życia”, a po realizacji danej funkcji – likwidowane. Mówiąc dokładniej, zmienne lokalne są tworzone i umieszczane na stosie (o którym mówiliśmy przy okazji poznawania biura słabo rozgarniętego urzędnika). Po zrealizowaniu danej funkcji zmienne są usuwane ze stosu, by niepotrzebnie nie zajmować miejsca. Co ważne, *zmienna lokalna stworzona na krótko do realizacji danego zadania (funkcji) nie jest dostępna dla innych zadań (funkcji)*.

Można jednak zmienną lokalną nazwać *statyczną* (dodać modyfikator ***static***), a wtedy kompilator umieści ją nie na stosie, tylko tam, gdzie są „trwałe” zmienne globalne (na tzw. stercie). Po realizacji danego zadania (funkcji) zmienna lokalna typu *static* nie tylko nie zniknie, ale też *zachowa zawartą w niej wartość*, gdy dane zadanie (funkcja) ma być wykonane jeszcze co najmniej raz.

Jeśli szukamy piękna i elegancji w języku C, odnotujmy: zasady języka pozwalają oszczędnie zarządzać pamięcią, ponieważ większość to zmienne lokalne, a nie globalne. Ma to także inne zalety, których pewnie na razie może nie docenisz: w tworzonym programie może być kilka zmiennych o tej samej nazwie i... „nie pogryzą się” one między innymi dlatego, że są tworzone i dostępne tylko na potrzeby określonego zadania (funkcji, bloku). Oczywiście używanie kilku zmiennych o tej samej nazwie nie jest godne polecenia z uwagi na zrozumiałość kodu dla człowieka, ale w rozbudowanych programach korzystających z „obcych” bibliotek jest to istotna zaleta!

A oto inna ważna sprawa: w przypadku „dużych komputerów” i program, i dane są umieszczone w pamięci RAM. W procesorach AVR jest inaczej. Pamiętamy, że ich pamięć RAM jest w sumie bardzo mała i dzieli się na trzy główne części: 32 rejestry bardzo ściśle związane z „kalkulatorem” CPU i obliczeniami bieżącymi, rejestry I/O do obsługi peryferii (63 podstawowe + ew. 160 dodatkowych) i „zwykłą” pamięć RAM, w której „najwyższej części” w razie potrzeby wydzielone jest miejsce na stos. Najczęściej zmienna to jedna lub więcej komórek w „zwykłej” pamięci RAM (raczej nie w obszarze rejestrów I/O, które są powiązane z różnymi peryferiami). Dane zawarte w zmiennych są przetwarzane przez „kalkulator” CPU, a ten, jak wiesz, ściśle związany jest z 32 rejestrami roboczymi (R0...R31). Zazwyczaj więc zawartość zmiennej, umieszczonej w „zwykłej” pamięci RAM, jest kopiowana do rejestru roboczego i dopiero tam przetwarzana.

I tu mamy bardzo ważny szczegół: najczęściej *operacje są więc wykonywane na kopii zmiennej*. Gdy dana zmienna ma być za chwilę znów wykorzystana, to dla oszczędności czasu nie jest powtórnie odczytywana i kopiowana ze „zwykłej” pamięci RAM, tylko wykorzystywana jest wcześniej stworzona kopia w rejestrze. Zazwyczaj nie ma z tym problemu, ale jeżeli „oryginalna” zmienna „bez wiedzy programu” zostanie zmodyfikowana, to *zawartość kopii z rejestru nie będzie odpowiadać prawdziwej wartości zmiennej*, co zaowocuje błędem. W przypadku omawianych mikrokontrolerów problem jest istotny wtedy, gdy zmienna w pamięci może zostać zmodyfikowana wskutek przerwania. *Aby uniknąć błędów, te zmienne, które mogą być modyfikowane podczas zgłoszenia i obsługi przerwania, należy w programie oznaczyć jako* ***volatile***.

Oznaczenie ***volatile*** wymusi na kompilatorze, by w programie wynikowym procesor za każdym razem odczytywał „oryginalną” zmienną ze „zwykłej” pamięci RAM, a nie posługiwał się jej kopią z rejestru.

Wielu Czytelników zdziwi tak bardzo duża rola kompilatora. Otóż przy programowaniu w asemblerze to człowiek – programista musi wszystko ogarnąć i szczegółowo zaplanować. Jest to poważna bariera, gdy programy stają się coraz większe. Natomiast programując w C, w bardzo zwięzły sposób opisujemy to, co procesor ma zrobić. A o szczegóły troszczy się kompilator języka C. Kompilator to potężny program o dużych możliwościach i „uprawnieniach”. O ile program napisany w asemblerze jest tłumaczony na kod maszynowy w pewnym sensie „dosłownie”, niejako 1:1, o tyle kompilator języka C nie jest prostym tłumaczem. On analizuje nasz skrótowy zapis, sprawdza, czy nie ma tam błędów, a potem stara się zrealizować zadanie w optymalny sposób. Ściślej biorąc, nie ma jednego „optymalnego sposobu”. W kompilatorze ustawiamy jeden z dostępnych stopni optymalizacji (-O1, -O2, -O3, -Os) i zależnie od tego kompilator podejmuje różne kroki, między innymi po to, by przyspieszyć działanie programu wynikowego. A szybkość zależy między innymi od rozmieszczenia zmiennych. Jeśli zmiennych jest mało, to mogą być umieszczone... bezpośrednio w „szybkich” rejestrach roboczych R0...R31 – wtedy nie trzeba tracić czasu na przesyłanie ze „zwykłej” pamięci RAM (w nielicznych przypadkach zmienne mogą być niektórymi rejestrami I/O). W języku C możemy „zachęcić” kompilator do umieszczenia danej zmiennej w „szybkich” rejestrach roboczych dyrektywą ***register***. Ale tylko „zachęcić”, a nie zmusić – to kompilator po analizie sytuacji zdecyduje, czy jest to możliwe. Na marginesie: niektóre mikrokontrolery AVR (np. ATtiny11, 12, 15, 28) w ogóle nie mają „zwykłej” pamięci RAM, a jedynie rejestry robocze (R0...R31) oraz (nieliczne) rejestry I/O; wtedy zmienne muszą być umieszczone w rejestrach roboczych.

Jeżeli wszystkiego nie rozumiesz, nie przejmuj się – z czasem wszystko Ci się „poukłada”. A na razie omówimy...

Deklarowanie i inicjalizacja zmiennych

Omówiliśmy różne typy zmiennych, od najprostszej *char* do *struktur*. W sumie w procesorze zmienne to zarezerwowane na krócej lub dłużej „kawałki” pamięci RAM, a typy zmiennych i omówione modyfikatory określają rozmiar oraz to, jak mają być rozumiane i traktowane zawarte tam zera i jedynki. Język C wymaga, żeby wszystkie wykorzystywane zmienne były wcześniej *zadeklarowane* (zgłoszone). Aby kompilator konkretnie zarezerwował „kawałek pamięci”, czyli aby stworzył zmienną, trzeba ją opisać - zdefiniować. Nie będę tłumaczył wszystkich szczegółów: **definicja (i deklaracja) zmiennej** to podanie jej ***typu*** (i ewentualnie modyfikatora), identyfikatora, czyli ***nazwy***, która jak najwięcej mówi o roli tej zmiennej oraz... ***średnika*** na końcu:

```
typ nazwa_zmiennej;
```

Przykłady:

```
char osmiobitowa_zmienna;
int stan_licznika;
float liczba_ulamkowa;
```

Nie jest dobrym zwyczajem pomijanie typu zmiennej – wtedy kompilator domyślnie przyjmie, że jest to 16-bitowy typ ***int*** ze znakiem (int16_t).

Ponieważ w zmiennej (w komórkach pamięci RAM) mogą być na początku potrzebne określone, a nie przypadkowe dane, podczas definiowania można ***inicjalizować zmienną***, to znaczy wpisać do niej wartość początkową:

```
char osmiobitowa_zmienna = 127;
uint8_t licznik = 255;
int16_t licznik = 32765; // (signed int)
float liczba = 2.25; // (zmiennoprzecinkowa)
float wspolczynnik = 3.16E-1; // =0.316
```

Zmienne tablicowe (tablice) deklarujemy bardzo podobnie, dodatkowo podając w nawiasie kwadratowym liczbę elementów tablicy i ewentualnie je inicjalizując, podając w klamrach zawartość:

```
char napis[10]; // 10 znaków typu char
int konwersja[32]; // 32 elementy typu int
int tab_pomocnicza[6] = {0,3,9,12,15,18};
char tekst[5] = {"Napis"};
```

Omówione wcześniej specyfikatory i modyfikatory wykorzystujemy właśnie przy deklarowaniu zmiennych:

```
unsigned char osmiobitowa_zmienna = 250;
static int16_t licznik;
volatile char licznik2;
register uint8_t wspolczynnik_pomoc;
```

Stałe

Programując w C, można wykorzystywać stałe. Ale trzeba wiedzieć, że ***stałe to w rzeczywistości zmienne, których zawartość nie powinna zmieniać się w trakcie pracy programu***. Czyli to też są zarezerwowane komórki RAM, do których rozkazy programu coś wpiszą, a potem nie będą tego zmieniać. Stałe deklarujemy tak jak zmienne, poprzedzając je słowem kluczowym **const** i oczywiście podając ich wartość. Na przykład:

```
const char nowa_stala = 127;
const uint8_t licznik = 255;
const int16_t licznik = 32765; // (signed int)
const float liczba = 2.25; // (zmiennoprzecinkowa)
const float wspolczynnik = 3.16E-1; // =0.316
```

W przeciwieństwie do zmiennych, stałe są wykorzystywane stosunkowo rzadko, bo zwykle można je bez problemu zastąpić po prostu liczbami w programie (tzw. literałami), co ma tę zaletę, że kod będzie wtedy krótszy.

Podsumujmy: Temat zmiennych i stałych wcale nie jest trudny. Język C narzuca dość rygorystyczne zasady dotyczące typów i deklarowania zmiennych, ale za to zapewnia ścisłą kontrolę nad zmiennymi, a to jest cenna właściwość. Fakt, że kompilator C wymaga od programisty skrupulatnego deklarowania/definiowania właściwości zmiennych, a potem ich pilnuje, to duża zaleta, ponieważ pozwala uniknąć mnóstwa błędów – to kolejny przykład „surowego piękna" języka C.

W następnym miesiącu zajmiemy się blokami i funkcjami, a w związku z drugą częścią artykułu o asemblerze wspomnimy też o wskaźnikach.

Piotr Górecki