Gospodarka pamięcią i zmienne w C. Część pierwsza [wersja robocza]

Pamięć jest straszliwa. Człowiek może o czymś zapomnieć — ona nie. Po prostu odkłada rzeczy do odpowiednich przegródek. Przechowuje dla ciebie różne sprawy albo je przed tobą skrywa — i kiedy chce, to ci to przypomina. Wydaje ci się, że jesteś panem swojej pamięci, ale to odwrotnie — pamięć jest twoim panem.

John Irving

Modlitwa za Owena

Obok samego procesora, najważniejszym komponentem systemu komputerowego jest niewątpliwie pamięć. Możemy ją klasyfikować na wiele sposobów: ze względu na czas dostępu, przepustowość, pojemność, koszt w przeliczeniu na MiB, możliwość odczytu/zapisu, sposób dostępu (sekwencyjny lub swobodny).

Z punktu widzenia programu napisanemu w języku C zajmiemy się trochę innym kryterium podziału, jakim jest przynależność do pewnych specjalnie zdefiniowanych obszarów takich jak rejestry, stos, sterta, obszar danych.

W programie napisanym w C dostęp do pamięci uzyskujemy za pośrednictwem zmiennych różnych typów (np. `int`, `char`, `double`, `int[]`, `char[]` i wiele innych). Zmienna w uruchomionym programie to pewien obszar w pamięci operacyjnej komputera, najczęściej w pamięci RAM. Obszar pamięci RAM, który należy do uruchomionej aplikacji (procesu), jest podzielony na kilka logicznych obszarów. W zależności od tego, w jaki sposób zadeklarujemy zmienną oraz czy będzie to zmienna alokowana statycznie czy dynamicznie, zostanie ona umieszczona w różnych obszarach.

Czy to ma dla nas jakieś praktyczne znaczenie? Czy ta wiedza jest nam potrzebna do szczęścia?

Celem powstania języków programowania wysokiego poziomu (a do takich zaliczamy C), i systemów operacyjnych było ułatwienie programiście posługiwania się zasobami pamięciowymi komputera. Najbardziej komfortowa sytuacja to taka, kiedy możemy założyć, że mamy nieskończenie dużo pamięci, jest ona nieskończenie szybka, a kompilator w taki sposób kompiluje kod, że nie jest możliwe wystąpienie problemów związanych z przypadkowym, niechcianym zamazaniem zawartości pewnych komórek pamięci przez źle napisany kod.

Tak to jednak w życiu bywa, że w praktyce musimy się liczyć z pewnymi ograniczeniami. Pamięci jest skończona ilość (w niektórych mikrokontrolerach zaledwie kilkadziesiąt bajtów), dostęp do pamięci może zauważalnie długo trwać (np. jeśli dane zostaną przeniesione przez system do obszaru *swap*), pewne rodzaje pamięci są dostępne tylko do odczytu (np. obszar pamięci kodu programu w mikrokontrolerach typowo znajduje się we Flashu, który wtedy działa jako ROM, a nie w RAM). Bywa i tak, że coś, co wygląda jak zwykła zmienna, np. identyfikator PTL

w mikrokontrolerze MC9S12NE64, w rzeczywistości nie znajduje się w pamięci RAM, tylko stanowi fizycznie port L, do którego wyprowadzeń przylutowane są diody świecące. Np. instrukcja PTL = 0x0F; spowoduje zapalenie kilku LEDów pomimo tego, że wygląda to jak zwykłe podstawienie wartości pod zmienną.

Zmienne globalne a lokalne

```
int i = 0; //to jest zmienna globalna

int main()
{
     int i = 1; //to jest zmienna lokalna

     {
          int i = 7; //to też jest zmienna lokalna
     }

     {
          int i = 999; //… kolejna zmienna lokalna
          printf("%d\n", i); //co się wyświetli?
     }

     printf("%d\n", i); //a co tutaj się wyświetli?
     return 0;
}
```

Zmienne lokalne to te, które zostały zdefiniowane między klamerkami { }. W szczególności wszystkie zmienne zdefiniowane w ciele funkcji (a więc właśnie między { }) to zmienne lokalne. Charakterystyczne dla zmiennych lokalnych jest to, że są one typowo tworzone na stosie (w obszarze pamięci programu, który nosi nazwę stos).

Polowanie na czarownice?

Niezupełnie, nazwa wzięła się raczej stąd, że stos to taka specyficznie zorganizowana pamięć, która przypomina stos książek. Typowy sposób obsługi tego rodzaju pamięci do odkładanie kolejnych książek na wierzchołek stosu, oraz zdejmowanie kolejnych książek z wierzchołka. Nie jest wskazana próba wkładania książki bezpośrednio w środek stosu (bo się on rozsypie) lub wyjmowanie książki z samego spodu.

Dlaczego zmienne lokalne są umieszczane właśnie tam? Ma to związek z mechanizmem wywołania funkcji w języku C i chwilowo zostawimy na boku to zagadnienie, wrócimy do niego później. Na razie istotniejszy jest fakt, że zmienne lokalne są umieszczane w pamięci dopiero wtedy, gdy wykonywany jest fragment kodu między { }, a znikają, kiedy miniemy klamrę zamykającą.

Jeśli zmienna lokalna jest umieszczona wewnątrz ciała funkcji, to „żyje" ta zmienna tylko wtedy, kiedy funkcja jest w wykonywana. Po opuszczeniu funkcji zmienna jest zdejmowana ze stosu.

To oznacza, że w programie możemy mieć tysiące funkcji, każda z nich może potencjalnie używać bardzo wielu zmiennych. Jeśli jednak w danej chwili wykonywane jest przykładowo 10 z nich, to tylko zmienne lokalne tych właśnie funkcji są umieszczane w pamięci. Jest to bardzo przydatne wtedy, kiedy tej pamięci mamy bardzo mało, ma to też inne zalety.

Zmienna lokalna **przesłania** zmienną o tej samej nazwie, która leży wyżej w hierarchii. To dobrze czy źle?

Gdybyśmy mieli tylko zmienne globalne, szybko zaczęłoby nam brakować pomysłów na krótkie, zwięzłe nazwy dla nich. Weźmy popularną, chętnie używaną zmienną `int i` z przykładu powyżej. Do zmiennej globalnej o tej nazwie potencjalnie możemy się odwołać (tzn. zapisać, odczytać) w dowolnym miejscu kodu w dowolnej funkcji. Łatwo tutaj o tragedię polegającą na tym, że w funkcji f_882() podstawiliśmy pod tą zmienną pracowicie wyliczone i odpowiednio zakodowane wyniki najbliższego losowania totolotka, a chwilę później w funkcji f_129() użyliśmy tej samej zmiennej w pętli for(i=0; i<1000; i++) licz_barany(i); Niestety, w takiej sytuacji zamiast wyników losowania w zmiennej znalazłaby się ostatecznie wartość 1000, imponująca jeśli chodzi o hodowlę baranów, ale mniej przydatna jako skuteczny środek do szybkiego wzbogacenia się.

Zdecydowanie lepszym pomysłem jest unikanie zmiennych globalnych na rzecz zmiennych lokalnych, nie wystąpią wtedy przypadkowe kolizje i nadpisywanie ich wartości przez różne fragmenty kodu.

Wracając do przykładu powyżej: nawet, jeśli w programie użyliśmy zmiennej globalnej o nazwie `i`, to wewnątrz funkcji można ponownie zdefiniować zmienną o tej samej nazwie i **przesłoni** nam ona wersję globalną. Poniżej inny fragment kodu, który to demonstruje:

```
int i = 0; //to jest zmienna globalna

void funkcja()
{
     int i; //to jest zmienna lokalna
     i = 123; //zmieniamy (zapisujemy) wartość zmiennej
     printf("lokalne i w funkcji ma wartość %d\n", i); //odczytujemy i
}

int main()
{
     printf("globalne i ma wartość %d\n", i);
     funkcja();
     printf("globalne i ma wartość %d\n", i);
     return 0;
}
```

Można pójść jeszcze dalej:

```
int i = 0; //to jest zmienna globalna

void funkcja()
{
     int i; //to jest zmienna lokalna
     i = 123; //zmieniamy (zapisujemy) wartość zmiennej
     {
          int i;
          i = 999999;
          printf("bardziej lokalne i ma wartość %d\n", i);
     }
     printf("mniej lokalne i ma wartość %d\n", i);
}

int main()
{
     printf("globalne i ma wartość %d\n", i);
     funkcja();
     printf("globalne i ma wartość %d\n", i);
     return 0;
}
```

Czy istnieje sposób sięgnięcia poza granice widoczności zmiennej? Konkretnie: czy z wnętrza funkcji można jakoś odczytać/zapisać wartość zmiennej globalnej, jeśli nastąpiło jej przesłonięcie zmienną lokalną o tej samej nazwie? Na szczęście jest to możliwe (aczkolwiek w ciągu pierwszego roku używania C raczej z tej możliwości nie będziecie korzystać).

Do takich dalekowzrocznych odwołań służy operator `::` (dwa dwukropki, co w sumie daje cztery kropki). Pod tym względem jesteśmy krok do przodu przed polonistami, którzy każą nam „tam sięgać, gdzie wzrok nie sięga”, ale jeśli chodzi o interpunkcję, to zatrzymali się na wielokropku złożonym zaledwie z trzech kropek. Programiści C rozwiązują ten poetycki problem następująco:

```
int i = 0; //to jest zmienna globalna
void funkcja()
{
     int i = 2011; //to jest zmienna lokalna
     printf("lokalne i ma wartość %d\n", i);
     printf("globalne i ma wartość %d\n", ::i);
     //no i sięgnęliśmy tam, gdzie normalnie się nie sięga.
}
```

Dlaczego funkcje mają zaniki pamięci?

Załóżmy, że chcemy zbudować funkcję, która zlicza ile razy została uruchomiona. Zwykle wywołania funkcji są bezpłatne, ale niedługo poznacie np. funkcję, która wysyła SMS. Każde jej wywołanie będzie nasz kosztować określoną liczbę groszy, warto zatem zadbać o finanse.

```
void wyslij_sms(char *odbiorca, char *tresc)
{
     int licznik_sms;
     licznik_sms++;
     printf("licznik_sms = %d\n", licznik_sms);
     //nawiązanie połączenia z bramka SMS i wysłanie wiadomości
     //...
}

int main()
{
  wyslij_sms("kolejny_idiota_z_listy", "Super wiesci! Nie wybrales 1, ale
     E*A & H*YAH daja Ci nowa szanse wygrania miliona zl!...");
  wyslij_sms("kolejny_idiota_z_listy", "Super wiesci! Nie wybrales 1, ale
     E*A & H*YAH daja Ci nowa szanse wygrania miliona zl!...");
  wyslij_sms("kolejny_idiota_z_listy", "Super wiesci! Nie wybrales 1, ale
     E*A & H*YAH daja Ci nowa szanse wygrania miliona zl!...");
  return 0;
}
```

Problem! Zmienna lokalna nie ma zerowej wartości początkowej. Wyzerujmy ją więc…

```
     int licznik_sms = 0;
```

…i mamy kolejny problem: tym razem licznik zawsze startuje od zera, można więc powiedzieć, że skuteczność takiego rozwiązania jest zerowa. Moglibyśmy zmienną **`licznik_sms`** uczynić zmienną globalną, wtedy faktycznie program będzie działać jak należy, jednak nauczeni wcześniejszymi złymi doświadczeniami nie chcemy, aby ten licznik był dostępny dla innych funkcji. Ma on być widoczny tylko i wyłącznie z wnętrza funkcji **`wyslij_sms(...)`**. Jak to zrobić?

Jak zjeść ciastko i mieć ciastko…

…czyli jak spowodować, żeby zmienna była lokalna, ale zachowywała się jak globalna (tzn. nie znikała jej wartość po opuszczeniu funkcji).

Jest to bardzo łatwe – wystarczy dodać słowo kluczowe **`static`** przed definicją tej zmiennej:

```
     static int licznik_sms = 0;
```

Tak naprawdę to zerowanie zmiennej w tym momencie nie jest już konieczne (automatycznie zostanie ona wyzerowana), ale nie rezygnujmy z dobrych nawyków.

W przypadku zmiennych lokalnych modyfikator **static** oznacza, że zmienna nie będzie umieszczana na stosie, lecz w obszarze pamięci, gdzie są przechowywane zmienne globalne. Ważne jest to, że taka zmienna zachowuje swoją wartość pomiędzy kolejnymi wywołaniami funkcji (może więc być użyta np. jako licznik wywołań funkcji), natomiast pod względem zasięgu zachowuje się taka zmienna jak zmienna lokalna (można więc zabezpieczyć się przed przypadkowym nadpisaniem jej zawartości przez inne fragmenty kodu).

Nic za darmo

Wydaje się, że zmienne lokalne mają więcej zalet niż wad: pozwalają uporządkować kod i „schować" zmienne wewnątrz funkcji, pozwalają oszczędniej gospodarować pamięcią (zmienne są alokowane wtedy, gdy funkcja lub blok kodu jest wykonywany i automatycznie usuwane z pamięci (zdejmowane ze stosu), kiedy opuszczamy funkcję lub blok.

Ograniczenia stosu (i zmiennych lokalnych)

Gdzie tu tkwi kruczek? Problemem jest najczęściej zbyt mały rozmiar stosu. Ten obszar nie jest przystosowany do przechowywania dużych ilości danych. Co oznacza „zbyt mały"? To zależy od systemu operacyjnego, kompilatora, sposobu kompilacji i kilku innych czynników. Możesz sam zmierzyć efektywny rozmiar twojego stosu, próbując wykonać poniższy fragment kodu.

Ważne! **Wykonać**, a nie **skompilować**. Z kompilacją nie powinno być problemu, problem może wystąpić w chwili wykonania tego kodu.

```
int main()
{
     char t[100]; //zmienna jest lokalna, wiec pójdzie na stos
     t[0] = 123;
     printf("udalo się!\n");
     return 0;
}
```

…

```
int main()
{
     char t[100000000]; //takich stosów nie mieli nawet w średniowieczu
     t[0] = 123;
     printf("udalo się!\n");
     return 0;
}
```

Jaka jest graniczna wartość rozmiaru tablicy t[], kiedy program poprawnie działa i nie zgłasza błędu? Co się stanie, kiedy dodamy **static** przed definicją tej tablicy?