Doskonalimy instrukcje sterujące

Wojciech Myszka

16 marca 2020

Streszczenie

Zadania na pierwszy tydzień zarazy.

Spis treści

Motto *Można doprowadzić konia do wodopoju, ale nie można zmusić go do picia*[1].

Wstęp

W zaistniałej sytuacji pozwalam sobie przedstawić zestaw informacji na pierwszy tydzień. Wysyłam go na adresy studentów uczęszczających na wykłady Informatyki I i Wprowadzenia do informatyki.

Pozostaję do Państwa dyspozycji drogą e-mail (wojciech.myszka@pwr.edu.pl). Teoretycznie istnieją również możliwości innych form kontaktu. Będziemy je rozważali jeżeli się pojawią. Ale zwracam uwagę, że nie jestem obecny w wielu (popularnych) sieciach społecznościowych.

Przypominam, że slajdy do wykładu dostępne są na mojej stronie WWW:

[1] Przysłowie indiańskie (za Wikipedią).

1. Informatyka I (Robotyka i Automatyka Przemysłowa),
2. Wprowadzenie do informatyki (Mechatronika).

Wszystkim zalecam (bardziej uporządkowany zestaw slajdów dla Mechatroniki).

Przypominam też o istnieniu „bryku" zawierającego rozszerzoną wersją slajdów (slajdy + komentarze). Instrukcjom sterującym poświęcony jest rozdział 5.

Język C w domu

Jest kilka sposobów skorzystania ze środowiska C w domu.

1. Stworzyć sobie takie środowisko na własnym komputerze.
    1. Gdy ktoś używa linuksa sprawa jest bardzo prosta, wystarczy doinstalować program geany (lub inne, ulubione środowisko). Kompilator i niezbędne biblioteki powinny już być.
    2. W przypadku windowsa jest nieco gorzej. Opracowałem kiedyś odpowiednią instrukcję (ale nie potrafię powiedzieć na ile opis ten jest przyjazny i wyczerpujący).
    3. W przypadku telefonu komórkowego/tableta z systemem Android — jest kilka(?) aplikacji.
2. Są też kompilatory on-line pozwalające na pracę w przeglądarce.

Instrukcje sterujące

Tematem tego tygodnia są **instrukcje sterujące**. Są to instrukcje pozwalające zmieniać kolejność wykonywania poleceń języka programowania. Z instrukcjami sterującymi związane są **operacje logiczne** (o których nieco w dalszym ciągu).

W języku C należą do nich instrukcje:

1. Warunkowa (`if`) występująca w trzech wariantach:
    - prostym
    - z `else`
    - wielowariantowym.
2. „Wyboru" (`switch`) używana niezbyt często (i nie zawsze dostępna w innych językach programowania).
3. Instrukcje do tworzenia pętli:
    1. `while`,
    2. `do` — `while`,
    3. `for`.

    Zwracam uwagę, że w pętlach **kontynuacja** obliczeń jest wykonywana gdy **warunek jest spełniony**; pętla zatrzymuje się, gdy warunek przestaje być spełniony.
4. Instrukcja `break` (możliwa do użycia wewnątrz instrukcji `switch` i instrukcji pętlowych); efektem jej użycia będzie przerwania/zakończenie aktualnie wykonywanego polecenia `switch` lub wyjście z pętli.

5. Instrukcja `continue` (możliwa do użycia **wyłącznie** wewnątrz pętli), która powoduje natychmiastowe przejście do następnego „cyklu” przetwarzania.
6. Polecenie `goto`. Zostało one, praktycznie, wyklęte przez teoretyków programowania. Nie występuje w wielu językach programowania, lecz występuje w języku C.

Polecenie `goto` funkcjonuje jakoś tak (na przykładzie prostego algorytmu: drukuj kolejne wartości `i` zmieniające się w zakresie od 0 do 9 włącznie):

```
int i;
i = 0;
start:                    \\ to jest "etykieta"
printf("i = %d\n", i);
i = i + 1;
if (i < 10) goto start; \\ w wyniku wykonania tego polecenia
                          \\ przejdziemy na początek
```

Operacje logiczne

Język C w wersji ANSI C nie zna typu logicznego. Występują w nim operatory logiczne:

- suma logiczna (LUB, OR) zapisywana jako `||`,
- iloczyn logiczny (I, AND) zapisywany jako `&&`,
- negacja zapisywana jako `!`.

Oprócz tego występuje zestaw operatorów porównania:

1. Większe (`>`),
2. Większe-równe (`>=`),
3. Mniejsze (`<`),
4. Mniejsze-równe (`<=`),
5. Równe (`==`),
6. Różne (`!=`).

Wynikiem operacji porównania jest **wartość całkowita** `1` (prawda, gdy warunek jest spełniony) lub `0` (fałsz, gdy warunek nie jest spełniony). Dodatkowo umówiono się, że każda wartość **różna od zera** będzie traktowana przez operatory logiczne C jako prawda.

Zatem argumentem poleceń `if`, `while` i `do`—`while` zawsze jest liczba (będąca, na przykład, wynikiem jakiegoś porównani).

Zadanie praktyczne: Binarny algorytm Euklidesa

Zakładam, że wszyscy z czytelników (słuchaczy) mają opanowane polecenia `for`, `while`, `do`—`while`. Jeżeli nie — proponuję zaprogramować poniższy elementarny algorytm (Euklidesa znajdowania Największego Wspólnego Dzielnika (NWD) dwu liczb).

Niech $m > 0$ i $n > 0$ będą dwoma liczbami całkowitymi. Mamy znaleźć taką liczbę, która jest dzielnikiem m i n i równocześnie jest największą ze wszystkich dzielników.

Algorytm Euklidesa wygląda jakoś tak:

1. Niech r będzie resztą z dzielenia m przez n (`r = m % n;`)
2. Jeżeli `(r == 0)` to `n` jest NWD dla `m` i `n`.
3. W przeciwnym razie wykonaj poniższe:

   ```
   m = n;
   n = r;
   ```

   i przejdź do kroku 1.

Powyższy algorytm należy zaprogramować używając poleceń `while`, `do`—`while` i `for`. Należy starać się, aby program był jak najprostszy i najkrótszy.

Zwracam uwagę, że **istotą** tego algorytmu jest to, że polecenia wykonywane są **zawsze** w kolejności 1 — 2 — 3 — 1 — 2 — 3 — 1—... kończąc się **po** wykonaniu kroku 2. Utrzymanie tej kolejności jest istotą algorytmu.

Algorytm jest bardzo prosty (występuje również w wersji z odejmowaniem). Istnieje jednak znacznie bardziej skomplikowana wersja zwana wersją binarną. Wygląda ona jakoś tak ($u > 0$ i $v > 0$ to dwie liczby całkowite, dla których szukamy NWD, t i k to zmienne pomocnicze, całkowite).

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k + 1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy, do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do kroku 3.
5. Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.
6. Przyjmij $t \leftarrow u - v$. Jeśli $t \neq 0$ to wróć do kroku 3. W przeciwnym razie algorytm zatrzymuje się z wynikiem $u \cdot 2^k$.

Do wykonania są następujące zadania:

1. Narysować schemat blokowy powyższego algorytmu (beż tego będzie znacznie trudniej wykonać dalsze zadania).
2. Postarać się zrozumieć działanie algorytmu (bez tego będzie trudniej).
3. Znaleźć na schemacie blokowym wszystkie „powtórzenia obliczeń” (pętle).
4. Zaproponować program w języku C realizujący ten algorytm używając wszędzie tam gdzie występuje polecenie „przejdź do kroku” instrukcję `goto`. W miarę możliwości przetestować jego działanie.
5. W kolejnym kroku należy spróbować tak zmodyfikować algorytm, żeby zrezygnować wszędzie z użycia polecenia `goto`, a wszędzie tam gdzie jest ono użyte do realizacji powtarzania

obliczeń (pętli) skonstruować te pętle z użyciem poleceń `while` lub `do—while`.
6. W miarę możliwości zaprogramować i przetestować ten algorytm.

Czemu taka nazwa (binarny)?

Zwracam uwagę, że wszystkie operacje występujące w algorytmie mogą być wykonane w najprostszy możliwy sposób:

1. dzielenie przez dwa może być wykonane tak samo jak dzielenie przez dziesięć (w układzie dziesiętnym) — metodą przesuwania przecinka.

   Wartość binarna `1010` (dziesięć w układzie dziesiętnym) po przesunięciu przecinka **w lewo** (albo inaczej całej liczby **w prawo**) staje się wartością `101` (czyli pięć w układzie dziesiętnym).

   Operatorem binarnym do przesuwania bitów w prawo jest `>>`, czyli `t = t/2` może być zastąpione przez `t = t>>1;` (przesuń bity w prawo o jedno miejsce); może to być również zapisane jako `t >>= 1;`

2. Sprawdzenie czy liczba jest parzysta czy nie (które najczęściej wykonujemy sprawdzając czy reszta z dzielenia jest równa 0 lub 1) może być zastąpione sprawdzeniem ostatniego bitu wartości (w zapisie binarnym). Żeby wyodrębnić ostatni (najmniej znaczący) bit z liczby trzeba **wyzerować** bity pozostałe.

   Może to być zrealizowane przez bitową operację iloczynu (`&`) z wartością `1`:

   `01101101` (dziesiętnie, to wartość $1 + 0 * 2 + 4 + 8 + 0 * 16 + 32 + 64$ czyli 109)

   `00000001`

   po nałożeniu bitów, w wyniku będzie zero wszędzie tam gdy którakolwiek wartość jest równa 0 i jeden tam, gdzie w obu wartościach jest równa 1;

   zatem `x & 1` jest równe 1 gdy x nieparzyste i zero — gdy parzyste; zatem warunek „gdy x parzyste" może zostać zastąpiony `if (x & 1 == 0)` lub `if (!(x & 1))` (czemu?).

3. Na koniec potęgowanie (a nawet szerzej operacja $u \cdot 2^k$) może być zastąpiona czymś znacznie prostszym. Popatrzmy na analogię z układem dziesiętnym: $3 \cdot 10^3$ (trzy tysiące) to po prostu trójka i trzy zera (3000); wystarczy przesunąć trójkę o trzy pozycje w lewo $3 \to 30 \to 300 \to 3000$. W układzie dwójkowym $u \cdot 2^k$ to u przesunięte w lewo k razy, czyli `u>>k`.

Operacje realizowane jak powyżej są znacznie bardziej efektywne niż operacje realizowane w sposób „normalny". Zatem wersja binarna algorytmu Euklidesa może być znacznie bardziej efektywna niż wersja klasyczna.

Zakończenie

Nie wiem jakie działania podejmą moi koleżanki i koledzy prowadzące zajęcia „przy komputerach"... W każdym razie jeżeli Państwu coś zlecą do wykonania — proszę to wykonać. Jeżeli nie

— trzymajcie się moich instrukcji laboratoryjnych i tych zapisków.

Jako lekturę uzupełniającą polecam (ale tylko wszystkim dorosłym) Dekameron Giovanniego Boccaccio.

Trzymajcie się zdrowo!

Zawartość w postaci pliku PDF...

...jest również dostępna.