Funkcje w czasach zarazy

Wojciech Myszka

21 marca 2020

Streszczenie

Zadania na drugi tydzień samoizolacji

Spis treści

Funkcje **1**
- Deklaracja funkcji . . . 3
- Polecenie `return` . . . 3
- Parametry funkcji . . . 4

Pierwsza funkcja **5**
- „Normalna” deklaracja funkcji . . . 7
- Deklaracja z wykorzystaniem prototypu . . . 7

Funkcje matematyczne **8**

Zadania do wykonania **9**

Wersja pdf... **10**

Literatura **10**

Funkcje

Przy pewnej dozie pomysłowości, również funkcje można zaliczyć do grupy „instrukcji sterujących”. Służą one bowiem do takiej organizacji algorytmu,

aby podzielić go na mniejsze „pod-algorytmy”.

Dla każdego z nich definiujemy wymagania jakim muszą odpowiadać **dane wejściowe** oraz **wyniki**, programujemy, wykazujemy, że działa poprawnie i przestajemy się zajmować jego wnętrzem, tylko używamy wszędzie tam gdzie jest przydatny.

Idea (oraz nazwa) została zaczerpnięta z matematyki. Dobrym przykładem są *funkcje trygonometryczne*. I tak funkcja **sinus** kąta α (w trójkącie) może być zdefiniowana jako:

> stosunek długości przyprostokątnej leżącej naprzeciw tego kąta do przeciwprostokątnej.

Nie jest to jedyna możliwa definicja tej funkcji (ale większość z nich chyba sięga do różnych pojęć geometrycznych).

Niezależnie od tego, którą definicję wybierzemy, aby wyliczyć wartość tej funkcji dla 70° nikt chyba nie konstruuje odpowiedniego trójkąta i nie przeprowadza odpowiednich pomiarów. Już raczej sięgamy po kalkulator albo (kiedyś) po tablice trygonometryczne, zakładając, że toś już to za nas zrobił. Jako naturalne przyjmujemy, że każdy (chyba) język programowania funkcję taką zna.

Co więcej, nie warto chyba programować tego samemu, zawsze gdy chcemy z tej funkcji korzystać, gdyż może okazać się, że nie zrobimy tego w sposób wystarczająco staranny, optymalny albo dokładny.

Z funkcji korzystamy

- aby ułatwić sobie życie[1],
- aby uczynić algorytm bardziej czytelnym[2],
- aby uczynić go mniejszym[3].

Ostatnim, może nie najważniejszym argumentem za używaniem funkcji jest to, że **ja tego wymagam**.

[1] Korzystamy z „gotowców” zamiast programować wszystko od początku.

[2] Osoba analizująca algorytm widząc odwołanie do funkcji `sin()` może mieć jakieś intuicje, znacznie lepsze niż widząc szereg pętli, które organizują jakieś obliczenia (i nie jest specjalnie jasne, że jest to wzór Taylora).

[3] Gdy już znajdziemy wszystkie miejsca, w których realizowane są identyczne obliczenia, możemy zaprogramować je tylko raz.

Deklaracja funkcji

Deklaracja funkcji powinna określać:

- typ zwracanej wartości
- nazwę funkcji
- wszystkie parametry funkcji (ich nazwy i typy)
- kod algorytmu realizującego obliczenia.

Deklaracja powinna znajdować się **przed pierwszym użyciem funkcji**.

Deklaracja **nie może** znajdować się wewnątrz innej funkcji.

Deklarację można podzielić na dwie części, najpierw definiując:

- typ zwracanej wartości,
- nazwę funkcji,
- typy wszystkich parametrów funkcji.

Jest to, tak zwany **prototyp** funkcji.

Ta informacja **musi** znaleźć się przed pierwszym użyciem funkcji.

W kolejnym etapie deklarujemy funkcję wraz z kodem algorytmu. Ta definicja może znaleźć się gdziekolwiek.

Pliki nagłówkowe (na przykład `stdio.h`) zawierają właśnie prototypy funkcji.

Polecenie `return`

Istotną (jeżeli nie najistotniejszą) częścią każdej funkcji jest polecenie `return` . Mówi ono która z wartości wyliczanych wewnątrz funkcji ma być traktowana jako jej wynik.

To o czym należy pamiętać to fakt, że w ten sposób można z funkcji „wyprowadzić na zewnątrz" wartość **jednej zmiennej**. Typ może być dowolny, a zmienna tylko pojedyncza. Stwarza to sporo problemów w przypadku konstruowania bardziej złożonych funkcji.

W języku C dopuszcza się funkcje nie posiadające polecenia `return`. Wówczas nie może być także zdefiniowany typ zwracanej wartości (określa się go jako `void`). Zatem konstrukcja taka:

```
void testowa(int x)
```

```
{
    printf("x=%d\n", x);
}
```

ma sens[4] i może być użyta w sposób następujący:

```
int z;
// Bardzo skomplikowane obliczenia
testowa(z);
// Ciąg dalszy
```

W niektórych językach programowania funkcje nie zwracające wartości nazywane są *procedurami*.

Parametry funkcji

Mimo, że dopuszczalne jest istnienie funkcji bez parametrów, na przykład:

```
float zmierz_temperatue()
{
    int sensorPin = 0;
    int reading = analogRead(sensorPin);
    return (reading * 5./ 1024. -0.5) * 100.;
}
```

(zakładam, że nasz komputer wyposażony jest w jeden czujnik temperatury podłączony do wejścia o numerze 0; konwerter analogowo-cyfrowy zwraca wartość całkowitą, którą trzeba przekształcić na temperaturę za pomocą prostego przekształcenia).

Używamy funkcji

```
printf("temperatura=%fC\n", zmierz_temperature());
```

Zwracam uwagę, że w każdym wywołaniu funkcji (nawet takiej, która nie wymaga parametrów) trzeba umieścić nawiasy po nazwie funkcji.

[4] Choć zasadnym może być pytanie po co stosować takie zabiegi, gdy można, prościej, bezpośrednio w kodzie programu umieścić polecenie `printf()`. Drobnym uzasadnieniem może być konieczność drukowana pewnych wartości podczas uruchamiania programu. Gdy skończymy uruchamianie wystarczy zmodyfikować funkcję `testowa()` i program będzie „cichszy".

W przypadku gdy funkcja **ma** jakieś parametry trzeba pamiętać o paru sprawach. Jako przykład wybierzmy funkcję `sin()`, która ma jeden parametr typu `double` i zwraca wartości typu `double`.

1. Dokonywana jest wyliczenie wartości parametru i konwersja typu aby doprowadzić do sytuacji, w której wartość będąca parametrem funkcji jest właściwego typu; zatem gdy wpiszemy `sin(10*5/11)` wyliczana jest wartość wyrażenia (`10*5/11`), która wynosi `4` (gdyż wszystkie wartości w tym wyrażeniu są typu całkowitego, `int`); następnie wyliczona wartość promowana jest do typu `double` (bo taki powinien być argument funkcji sinus) czyli, w efekcie sprowadza się to do wywołania o postaci `sin(4.0)`.
2. W kolejnym kroku wartość argumentu **kopiowana jest** do wnętrza funkcji sinus, która prowadzi odpowiednie obliczenia.
3. Gdy nasze wywołanie wyglądało tak:

   ```
   double z;
   // tu jakieś obliczenia
   z = sin(10*5/11);
   ```

   wyliczona przez funkcję sinus wartość podstawiana jest do zmiennej `z`. (W zależności od typu zmiennej po lewej stronie znaku równości może być dokonywana promocja/degradacja typu.)

Wszystkie zmienne zadeklarowane wewnątrz funkcji są zmiennymi lokalnym i nie interferują z innymi zmiennymi o tych samych nazwach. Dotyczy to również parametrów funkcji:

```
int s( int x)
```

deklarowana jest tu zmienna `x` typu `int` **lokalna** dla funkcji `s`.

Pierwsza funkcja

Załóżmy, że w wielu miejscach korzystamy z operacji podnoszenia do kwadratu. Nie jest to operacja specjalnie złożona, ale język C nie zna operatora „podnieś do potęgi". Jest oczywiście funkcja `pow()`, ale to bardzo uniwersalna funkcja, która operacje wykonuje korzystając z logarytmów...

Można z niej korzystać, ale można uznać, że jest to *overkill*.

Zatem zamiast pisać

```
y = a * a + b * b;
```

stworzymy sobie funkcję `s()`podnoszącą do kwadratu i zapiszemy to tak:

```
y = s(a) + s(b);
```

Definicja tej funkcji może wyglądać tak:

```
int s( int x)
{
    return x * x;
}
```

a program z niej korzystający może wyglądać tak:

```
int main()
{
    int y;
    y = s(2) + s(3);
    printf("y = %d\n", y);
    return 0;
}
```

W wyniku wykonania programu powinniśmy otrzymać feralną wartość 13.

Zwracam uwagę, że gdy linię `y=s(2)+s(3);` zastąpimy

```
y = s(2.5) + s(3.5);
```

wynik się nie zmieni, gdyż funkcja `s()` zakłada że jej argumenty to wartości całkowite (`int`). Zatem w wywołaniu funkcji `s(2.5)` najpierw wartość `2.5` (**stała typu double**) zostanie zdegradowana do `2` i dalsze obliczenia zostaną wykonane już na tej wartości. (Podobnie i w przypadku `s(3.5)`.)

Gdy jednak funkcję zdefiniujemy inaczej:

```
double s(double x)
[
    return x * x;
]
```

obliczenia programu

```
int main()
{
    iny y;
    y = s(2.5) + s(3.5);
    printf("y=%d\n", y);
    return 0;
}
```

przebiegną inaczej. Funkcja `s(2.5)` podniesie do kwadratu wartość `2.5` dając w efekcie `6.25`; `s(3.5)` da `12.25`; po zsumowaniu otrzymamy `18.5` i ta wartość zostanie zdegradowana do wartości całkowitej `18` podczas podstawiania do `y`.

Niestety trzeba o tym pamiętać podczas projektowania funkcji.

„Normalna" deklaracja funkcji

Kod będzie wglądał jakoś tak:

```
#include<stdio.h>
int s(int x) // s musi być zadeklarowane przed main gdzi ejest użyte
{
    return x * x;
}
int main()
{
    int y;
    y = s(2) + s(3);
    printf("y=%d\n", y);
    return 0;
}
```

Deklaracja z wykorzystaniem prototypu

```
#include<stdio.h>
int s (int); // prototyp, średnik na końcu niezbędny
int main()
{
    int y;
```

```
    y = s(2) + s(3);
    printf("y=5d\n", y);
    return 0;
}
int s (int x) // to jest właściwa deklaracja funkcji
{
    return x * x;
}
```

Funkcje matematyczne

Aby móc korzystać z funkcji matematycznych należy na samym początku pliku umieścić polecenie:

```
#include <math.h>
```

Powoduje ono wczytanie pliku zawierającego „definicje”[5] (ale nie kod!) wielu, powszechnie używanych, funkcji matematycznych[6]. W tym i funkcji `sin()`.

Natomiast kod wszystkich podstawowych funkcji realizujących operacja matematyczne zawarty jest w specjalnej bibliotece. Nazywa się ona `libm`. Z różnych historycznych powodów nie jest ona standardowo[7] przeglądana podczas kompilacji programu.

Prowadzi to do dziwnych nieco sytuacji, w których aby skorzystać z funkcji `rand()` trzeba wczytać plik `stdlib.h` (zawiera jej definicję, ale nie kod); kod tej funkcji znajduje się w bibliotece funkcji (`glibc`), która **zawsze** jest przeglądana podczas kompilacji i nie trzeba robić nic więcej.

W przypadku chęci skorzystania z funkcji sinus dodatkowo trzeba **poprosić** kompilator aby „zajrzał” do biblioteki `libm`. Kompilując „z ręki“ piszemy:

```
gcc test.c -o test -lm
```

`-lm` jest to prośba o zajrzenie do biblioteki (`-l`) która nazywa się lib**m**.

[5] Czyli prototypy.

[6] Wykaz wszystkich funkcji (symboli) zdefiniowanych w pliku `math.h` znaleźć można, na przykład, w Wikibooks. Jest tam całkiem niezły podręcznik do języka C („C” 2020).

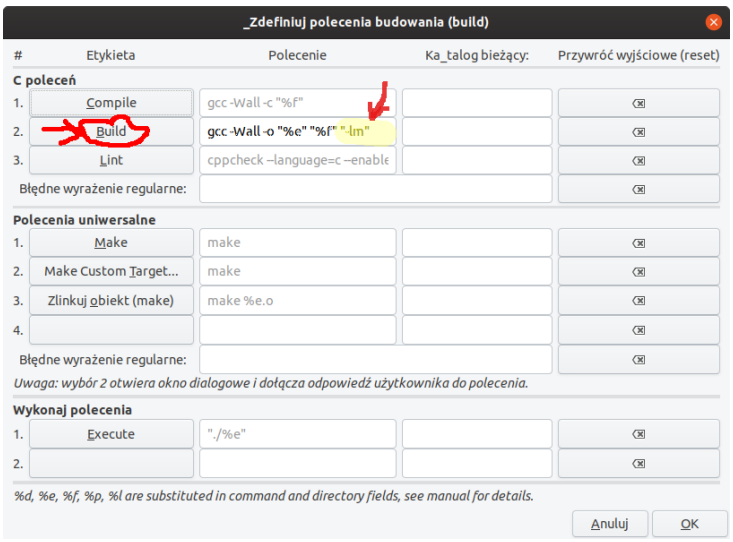
[7] Choć, bardzo często w implementacjach języka C, w systemie Windows jest ona przeglądana.

Gdy korzystamy ze środowiska graficznego (*gui*) sprawa jest nieco bardziej skomplikowana. W przypadku geany, z górnego menu wybieramy „Zbuduj”, a następnie na samym dole rozwiniętego menu „Zdefiniuj polecenie budowania”. W otwartym okienku znajdujemy pozycję **Build** i po `gcc -Wall -o "%e" "%f"` dopisujemy `"-lm"`, żeby dostać coś takiego:

```
gcc -Wall -o "%e" "%f" "-lm"
```

klikamy OK i od teraz podczas kompilacji biblioteka matematyczna **zawsze** będzie przeglądana.

Ilustruje to poniższy obrazek



Rysunek 1: Dodanie biblioteki matematycznej

Zadania do wykonania

1. Zapoznać się z instrukcją laboratoryjną Laboratorium 4: Funkcje metoda połowienia.
2. Zapoznać się dokładniej ze zmiennymi typu `double` (bryk, rozdziały 4.4.3, 4.6, 4.9, 4.10).
3. Zapoznać się z materiałem dotyczącym funkcji (wykład 5, bryk rozdział 7).

4. Napisać (i przetestować) prostą funkcję o nazwie `my_sin()` mająca jeden argument typu `double` i zwracającą wartości `double` czyli `double my_sin(double)`[8] wyliczającą wartość funkcji sinus dla argumentu podanego w stopniach, a nie radianach (czego standardowo wymaga funkcja sinus).
5. Przemyśleć zasadę działania metody połowienia.
6. Spróbować zaprogramować metodę połowienia:
   - bez funkcji,
   - zamykając algorytm jako zwykłą funkcję,
   - metodą rekurencyjną (wywołanie funkcji przez samą siebie zastępuje przejście na początek algorytmu); **wymóg ekstra**.

   Jako funkcję, której miejsca zerowego będziecie szukać proponuję wybrać jakąś funkcję, dla której znacie jego położenie. Może to być funkcja `sin(x)`, która w przedziale $0 < x < 2\pi$ spełnia wymagania metody (jedno miejsce zerowe).

Wersja pdf...

...jest również dostępna.

Literatura

„C”. 2020. Wikibooks. https://pl.wikibooks.org/wiki/C.

[8] To jest prototyp funkcji.