Podstawy Programowania 1

Funkcje

Arkadiusz Chrobot

Katedra Systemów Informatycznych

27 października 2019

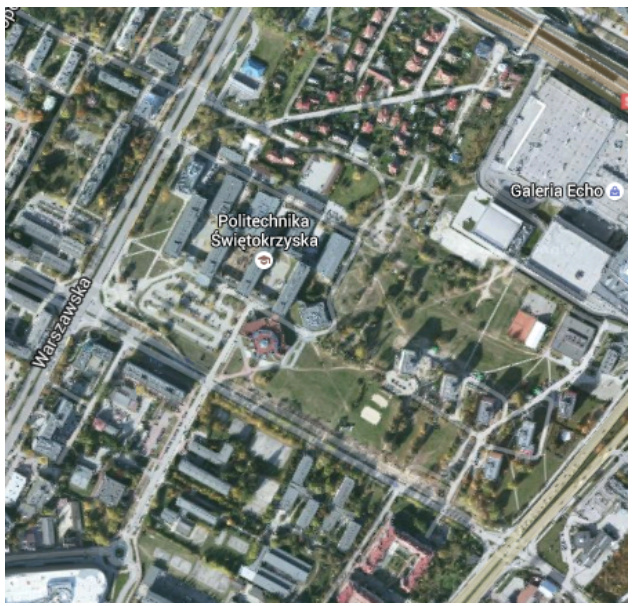
Plan

Programowanie strukturalne

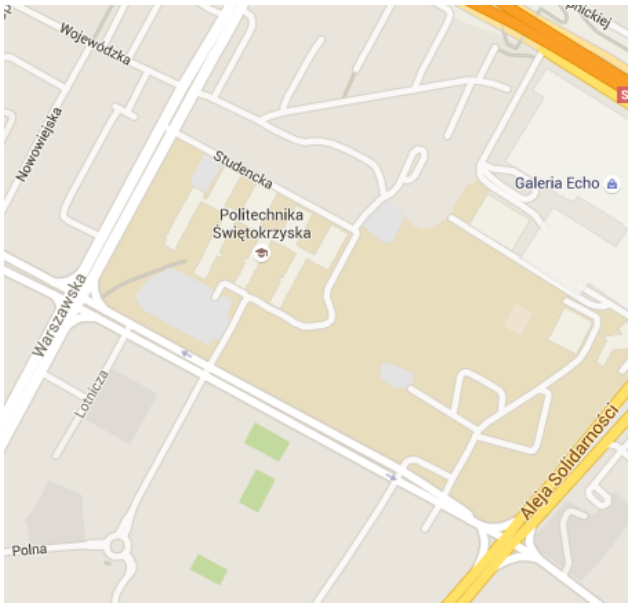
Głównym elementem paradygmatu (modelu) programowania strukturalnego jest uproszczenie zapisu programu komputerowego poprzez jego podział na mniejsze części i nadanie każdej z nich osobnej nazwy. Ważną cechą tego podziału jest hierarchiczność - elementy programu mogą korzystać z innych elementów, pod warunkiem, że te ostatnie zostały wcześniej zdefiniowane lub zadeklarowane. Pozwala to zastosować do problemów informatycznych zasadę analizy Kartezjusza, czyli dzielenia problemu na mniejsze zagadnienia, takie, których rozwiązania można łatwo znaleźć i łącząc ich wyniki otrzymać rozwiązanie problemu wyjściowego. Taki model programowania pozwala także na zastosowanie abstrakcji.

Abstrakcja

Abstrakcją w programowaniu, ale nie tylko, nazywamy proces upraszczania określonych elementów rozpatrywanego problemu, w taki sposób, aby uwypuklić jego najważniejsze cechy, a ukryć te, które są mniej istotne lub w ogóle nieistotne dla znalezienia rozwiązania. Przykładem zastosowania abstrakcji są mapy:



(a) Zdjęcie satelitarne



(b) Mapa

Źródło: Google Maps

Funkcje

Funkcje w języku C są realizacją *podprogramów*. Pozwalają one zgrupować instrukcje, które tworzą określoną logiczną całość i nadać im nazwę. Patrząc na to zagadnienie od strony abstrakcji można stwierdzić, że funkcje pozwalają z istniejących w języku programowania instrukcji stworzyć nowe, które będą bardziej pomocne w napisaniu programu rozwiązującego zadany problem. To zastosowanie funkcji daje dodatkową korzyść w postaci wyeliminowania powtarzających się fragmentów kodu (powtórne użycie kodu). Funkcje mogą zwracać wartość, będącą wynikiem działania, w czym przypominają funkcje matematyczne.

Funkcje

Wzorzec funkcji

Ogólny wzorzec *definicji* funkcji jest następujący:

```
typ_wartości_zwracanej nazwa_funkcji(lista_parametrów)
{
    ...
}
```

Pierwszy wiersz w tym wzorcu jest nazywany *prototypem* lub *nagłówkiem* funkcji. Rozpoczyna się on typem wartości zwracanej przez funkcje (typem danych), po którym następuje nazwa funkcji, która podlega tym samym zasadom tworzenia, co inne identyfikatory w języku C (np. nazwy zmiennych). Po nazwie, w nawiasach okrągłych umieszczana jest lista parametrów, które są specjalnymi zmiennymi umożliwiającymi funkcji wymianę danych z innymi częściami programu. Po liście parametrów znajduje się blok instrukcji, który stanowi *ciało* lub *treść* funkcji, czyli zasadniczą część decydującą o jej działaniu.

Funkcje

Zwracanie wartości

Aby funkcja zwracała wynik należy w jej ciele użyć słowa kluczowego `return`. Za nim umieszcza się wyrażenie, którego wartość funkcja przekaże na zewnątrz. Wyrażenie to może, ale nie musi być zamknięte w nawiasach okrągłych. Ponadto, niekoniecznie musi ono być rozbudowane. Może nim być pojedyncza stała, zmienna, a nawet pojedyncza wartość. Jedyny wymóg, to zgodność typu wartości wyrażenia z typem wartości zwracanej przez funkcję umieszczonym w jej nagłówku. Słowo `return` oprócz zwracania wartości ma też inną własność - kończy działanie funkcji, a więc po jego wykonaniu żadne inne instrukcje nie są wykonywane.

Funkcje

Wywołanie funkcji

Aby uruchomić funkcję należy ją *wywołać* w innej funkcji, w szczególności może być ona wywołana w głównej funkcji programu, czyli w `main()`[1]. Wywołanie funkcji polega na umieszczeniu w programie jej nazwy i dodaniu za nią operatora wywołania funkcji, który w języku C jest oznaczany parą nawisów okrągłych. Jeśli funkcja nie ma parametrów, to ta para pozostaje pusta, w przeciwnym przypadku należy umieścić tam tyle *argumentów wywołania* (krótko: *argumentów*) ile jest parametrów. Argumenty to zmienne, stałe lub wyrażenia, których wartość będzie przekazana za pośrednictwem parametrów do funkcji. Typy parametrów i argumentów muszą być zgodne. Jeśli funkcja zwraca wartość, to można ją zapisać do zmiennej, o typie zgodnym z typem tej wartości. Wywołanie funkcji może być umieszczone także wewnątrz wyrażenia. Po zakończeniu funkcji wykonanie programu (sterowanie) wraca do instrukcji znajdującej się tuż za jej wywołaniem.

[1] Proszę zwrócić uwagę, że jeśli w tekście piszemy nazwę funkcji, to za nią umieszczamy pustą parę nawiasów okrągłych.

Funkcje

Zastosowanie słowa kluczowego `void`

Funkcje są implementacjami (zapisami) algorytmów, w związku z tym, w niektórych przypadkach mogą nie posiadać żadnych parametrów, co odpowiada sytuacji, kiedy algorytm nie ma żadnych danych wejściowych. W definicji funkcji napisanej w języku C oznacza się taką sytuację umieszczając słowo kluczowe `void` między nawiasami okrągłymi, zamiast listy parametrów. Często można spotkać zapis, w którym te nawiasy pozostają puste. Nie jest on jednak adekwatny użyciu słowa `void` na liście parametrów. Oznacza on, że funkcja przyjmuje nieokreśloną liczbę argumentów wywołania. Słowo kluczowe `void` może zostać użyte także na określenie typu wartości zwracanej przez funkcję. Oznacza to, że ta funkcja nic nie będzie zwracała. W niektórych programach można spotkać słowo kluczowe `void` umieszczone w nawiasach okrągłych funkcji przed jej wywołaniem. To oznacza, że funkcja zwraca wartość, ale jest ona ignorowana. Zwykle jednak nie stosuje się takiego zapisu, a jedynie nigdzie nie przypisuje się tej wartości. Taki rodzaj wywołania nazywamy wywołaniem funkcji dla *efektu ubocznego jej działania.*

Funkcje

Pierwszy prosty przykład

```
#include<stdio.h>

int f1(void)
{
        puts("Jestem funkcją f1(), zwracam wartość 5.");
        return(5);
}

int variable_1, variable_2;

int main(void)
{
        variable_1 = f1();
        variable_2 = 5*f1();
        (void) f1();
        f1();
        return 0;
}
```

Funkcje

Komentarz do pierwszego przykładu

Na poprzednim slajdzie został zaprezentowany program z funkcją, która nie przyjmuje żadnych argumentów wywołania, a jedynie wypisuje na ekranie komunikat i zwraca liczbę 5 jako swoją wartość. Funkcja ta jest czterokrotnie wywołana w programie. Za pierwszym razem jej wartość jest przypisana zmiennej `variable_1`. W drugim przypadku wywołanie funkcji jest częścią wyrażenia. Wartość przez nią zwrócona jest mnożona przez 5 i zapisywana w zmiennej `variable_2`. W dwóch ostatnich przypadkach wartość zwrócona przez funkcję jest ignorowana. O tym, że ona w ogóle się wykonała świadczą widoczne na ekranie komunikaty. Nazwa funkcji użyta w programie nie jest czytelna, a zatem dopuszczalna tylko w prostych przykładach. W złożonych programach powinna być ona bardziej opisowa. Najczęściej powinna ona także zawierać czasownik odzwierciedlający to, do czego ta funkcja służy.

Funkcje

Drugi prosty przykłady

```c
#include<stdio.h>

int f1()
{
        puts("Jestem funkcją f1(), zwracam wartość 5.");
        return(5);
}

int variable_1, variable_2;

int main(void)
{
        variable_1 = f1(1,2,3,4,5,6,7);
        variable_2 = 5*f1(variable_1, variable_2);
        return 0;
}
```

Funkcje

Komentarz do drugiego przykładu

Przykład pokazuje różnice miedzy definicją funkcji, w której pozostawiono pustą listę parametrów, oraz taką, w której użyto słowa kluczowego `void`. Wprawdzie w obu przypadkach lista parametrów jest pusta, ale w tym, który został zastosowany w przykładzie można w miejscu wywołania funkcji przekazać jej dowolne argumenty, które nie będą w niej w ogóle używane. Choć nie jest to zabronione (kompilator nawet nie zgłosi ostrzeżenia), to może prowadzić do błędów w programie. Proszę także zwrócić uwagę, że wartość zwracana jest zapisana po słowie kluczowym `return` w nawiasach okrągłych (funkcja `f1()`), jak i bez (funkcja `main()`).

Funkcje

Trzeci prosty przykład

```c
#include<stdio.h>

void f2(void)
{
        puts("Jestem funkcją f2() i nic nie zwracam.");
}

int a;

int main(void)
{
        f2();
        /*a = f2();*/ // To nie jest dozwolone.
        return 0;
}
```

Funkcje

Komentarz do trzeciego przykładu

Ten przykład demonstruje użycie funkcji, która nic nie zwraca. Nie można przypisać jej wartości do żadnej zmiennej, bo ona po prostu nie istnieje. Jedynym efektem działania funkcji `f2()` jest wypisanie komunikatu na ekranie.

Funkcje

Czwarty prosty przykład

```c
#include<stdio.h>

void f3(void)
{
        puts("Jestem funkcją f3().");
        return;
        puts("Nic nie zwracam i nie wypiszę tego komunikatu.");
}

int main(void)
{
        f3();
        return 0;
}
```

Funkcje

Komentarz do czwartego przykładu

Mimo, iż może się to wydać dziwne, to w funkcji, która nic nie zwraca możliwe jest użycie słowa kluczowego `return`. Po tym słowie musi występować od razu średnik. Instrukcja `return` w tym przypadku po prostu kończy wykonanie funkcji. Wszystkie instrukcje, które znajdą się za nią nie zostaną wykonane.

Funkcje

Deklaracja funkcji

Zadeklarowanie funkcji polega na umieszczeniu w kodzie źródłowym programu jej nagłówka zakończonego średnikiem. Tak zadeklarowaną funkcję należy zdefiniować w innym miejscu programu. Deklaracja funkcji pozwala na użycie jej w programie, zanim zostanie ona zdefiniowana. Takie rozwiązanie ma kilka zastosowań. Jednym z nich jest odwrócenie hierarchii definiowania funkcji, tak aby program można było czytać w ten sam sposób, jak tekst w języku naturalnym - „z góry na dół". Otrzymuje się ten efekt tworząc najpierw definicję funkcji `main()`, a potem definicje pozostałych funkcji, w takiej kolejności, aby te które korzystają z innych znajdowały się wyżej w kodzie źródłowym. Wszystkie takie funkcje muszą zostać zadeklarowane przed funkcją `main()`. Deklaracja funkcji jest także używana w sytuacji, gdy chcemy wywołać funkcję, której z jakiś powodów nie da się wcześniej zdefiniować.

Funkcje

Deklaracja funkcji - przykład

```c
#include<stdio.h>

void f4(void); //Prototyp funkcji

int main(void)
{
        f4();
        return 0;
}

void f4(void)
{
        puts("Jestem funkcją f4() zostałam zdefiniowana po funkcji main()");
}
```

Zmienne lokalne

Dotychczas posługiwaliśmy się zmiennymi globalnymi. Język C pozwala na deklarowanie zmiennych w funkcjach, a nawet w każdym bloku instrukcji. Począwszy od standardu ISO C99 język ten pozwala na deklarowanie zmiennych tuż przed ich użyciem, co zwiększa czytelność programu. Stosowanie zmiennych lokalnych ma szereg zalet. Jedną z nich jest lepsza gospodarka pamięcią operacyjną, niż ma to miejsce w przypadku zmiennych globalnych. Zmienne lokalne, istnieją w pamięci komputera tylko wtedy, gdy wykonywana jest funkcja, w której są zadeklarowane. Dlatego też, w języku C nazywa się je zmiennymi *automatycznymi*. Dodatkowo zmienne te dostępne są tylko wewnątrz bloku instrukcji, w którym zostały zadeklarowane i począwszy od miejsca deklaracji, natomiast instrukcje z tego bloku mają dostęp do wszystkich zmiennych wcześniej zadeklarowanych w otoczeniu tego bloku. Ta reguła widoczności obowiązuje również w przypadku innych elementów programu, które mogą być zadeklarowane lub zdefiniowane lokalnie, w tym także w przypadku stałych deklarowanych z użyciem słowa kluczowego `const`.

Zmienne lokalne

Przykrywanie nazw zmiennych

Zmienne lokalne mogą mieć takie same nazwy jak zmienne globalne, lub nawet inne zmienne lokalne, ale zadeklarowane w innych blokach instrukcji. W takim wypadku, jeśli istnieje w programie kilka zmiennych o takich samych nazwach, to w określonym bloku pod tą nazwą dostępna jest ta zmienna, której deklaracja jest najbliższa. Pozostałe są poza zasięgiem instrukcji z tego bloku. Własność ta nazywa się *przykrywaniem nazw* i dotyczy nie tylko zmiennych, ale również innych elementów programów.

Zmienne lokalne

Przydzielanie i zwalnianie pamięci na zmienne lokalne

Pamięć na zmienne lokalne jest przydzielana i zwalniana automatycznie i czynności te, tak jak w przypadku zmiennych globalnych, nie wymagają dodatkowych zabiegów ze strony programisty, oprócz deklaracji tych zmiennych. Takie rozwiązanie jest możliwe dzięki podziałowi pamięci, do której jest ładowany program w postaci wykonywalnej, na kilka obszarów, o różnych własnościach. Jeden z tych obszarów nazywa się segmentem kodu i zawiera instrukcje do wykonania, drugi nazywa się obszarem danych i w nim są umieszczane zmienne globalne, dlatego istnieją przez cały czas wykonania programu. Trzeci obszar to obszar stosu. W tym obszarze przy wywołaniu dowolnej funkcji są tworzone tzw. *ramki stosu* lub *rekordy aktywacji*, czyli struktury zawierające miejsce na zmienne lokalne, parametry (które są formą takich zmiennych) i adres powrotu do instrukcji, która ma zostać zrealizowana po zakończeniu wykonania funkcji.

Zmienne lokalne

Przydzielanie i zwalnianie pamięci na zmienne lokalne

Jeśli z wnętrza funkcji jest wywoływana inna funkcja, to rekord aktywacji dla niej jest tworzony na stosie zgodnie z regułą „ostatni nadszedł, pierwszy wychodzi", czyli LIFO, od angielskich słów *Last In First Out*. Oznacza, to że będzie on usunięty wcześniej niż rekord dla funkcji wywołującej, co jest zgodne z logiką wykonywania funkcji - najpierw kończy działanie funkcja wywoływana, później wywołująca. Miejsce po zwolnionych rekordach aktywacji może być wykorzystane przez rekordy aktywacji innych funkcji. **Konsekwencją takiego tworzenia zmiennych lokalnych jest brak ich domyślnej inicjacji - musi o to zadbać programista.** W przeciwieństwie do zmiennych globalnych nie mają one początkowo wartości zerowych. Pamięć na zmienne lokalne deklarowane w blokach instrukcji znajdujących się w funkcjach też jest przydzielana w rekordach aktywacji tych funkcji, ale w taki sposób, aby zużyć jak najmniej miejsca na stosie.

Zmienne lokalne

Słowo kluczowe `static`

Ze względu na to, że zmienne lokalne istnieją w pamięci tylko wtedy, gdy wykonywana jest funkcja z nimi związana, to zaleca się stosować je zamiast zmiennych globalnych, wszędzie tam, gdzie jest to możliwe. Słowo kluczowe `static` może być stosowane wraz ze zmiennymi lokalnymi i pozwala połączyć cechy zmiennej lokalnej oraz zmiennej globalnej. Tak zadeklarowana zmienna jest widoczna tylko w funkcji, w której została stworzona. Jednakże istnieje ona przez cały czas wykonania programu. Jej wartość początkowa jest zerowa, ale jeśli zostanie zmodyfikowana przez jedną z instancji (jedno z wywołań) funkcji, to następne będą tę zmianę widziały.

Zmienne lokalne

Przykład

```c
#include<stdio.h>

void count_instances(void)
{
    static unsigned int sum;
    sum+=1;
    printf("Funkcja była wywołana %d razy.\n",sum);
    int i;
    for(i=0; i<5; i++) {
        int i = 1;
        printf("Ta zmienna ,,i'' nie jest licznikiem pętli: %d\n",i);
    }
}

int main(void)
{
    count_instances();
    count_instances();
    return 0;
}
```

Bezpośrednie korzystanie ze zmiennych globalnych

Funkcje mogą bezpośredno korzystać ze zmiennych globalny, ale takie rozwiązanie ma szereg wad i nie zaleca się korzystania z niego. Aby zilustrować jedną z tych wad postarajmy się odpowiedzieć na pytanie co robi funkcja, która wywoływana jest w programie następująco:

Wywołanie funkcji `add_numbers()`

```
int sum = add_numbers();
```

Nazwa funkcji sugeruje, że dodaje ona do siebie liczby. Typ zmiennej, do której zapisany jest wynik, wskazuje, że prawdopodobnie będą to liczby całkowite. Nie wiemy jednak, ile będzie tych liczb, bo są one zapisane w zmiennych globalnych. Mogą to być dwie liczby, może ich być więcej. Inną wadą tej funkcji jest to, że nierozerwalnie związana jest z programem, w którym znajduje się jej definicja. Nie da się jej przenieść do innego programu nie przenosząc deklaracji zmiennych globalnych, z których korzysta. To tylko dwie z wielu wad bezpośredniego korzystania ze zmiennych globalnych w funkcji.

Parametry

Aby uniknąć opisanych wcześniej problemów należy użyć parametrów. Parametr jest specjalną zmienną lokalną, dzięki której funkcja może komunikować się ze swoim otoczeniem. Funkcje mogą posiadać więcej niż jeden parametr. Każdy z parametrów funkcji może mieć inny typ lub mogą się one powtarzać. Parametry nie mogą mieć takich samych nazw, jak zmienne lokalne definiowane bezpośrednio w ciele funkcji (poza innymi blokami instrukcji). W miejscu wywołania funkcji każdemu z jej parametrów należy przypisać odpowiadający mu argument wywołania, o typie kompatybilnym (zgodnym) z typem parametru. Istnieją trzy sposoby na przekazanie argumentów do funkcji przez parametry. Jedną z zalet parametrów jest to, że czynią one funkcje bardziej uniwersalnymi.

Przekazanie przez wartość

Parametry dla takiego przekazania tworzymy na liście parametrów funkcji tak, jak zwykłe zmienne, ale rozdzielając je przecinkami, nie średnikami. Wyjątkiem jest sytuacja, gdy chcemy stworzyć kilka parametrów o takim samym typie. Wówczas dla każdego z nich musimy podać typ zmiennej. Zazwyczaj wszystkie deklaracje parametrów umieszczamy w jednym wierszu. Za takie parametry w miejscu wywołania wolno nam podstawić argumenty, które mogą być wartościami, stałymi, zmiennymi (zarówno globalnymi, jak i lokalnymi) lub całymi wyrażeniami. Parametry dla przekazania przez wartość są parametrami wejściowymi, tzn. jeśli podstawimy pod nie argumenty będące zmiennymi, to po wykonaniu funkcji wartości tych argumentów nie ulegną zmianie, mimo, że wartości parametrów mogą być w funkcji zmieniane. Od strony technicznej takie zachowanie jest uzyskiwane poprzez kopiowanie wartości argumentów do parametrów. Innymi słowy przekazanie przez wartość jest równoważne przypisaniu wartości argumentu parametrowi. Ten sposób przekazania można łączyć z pozostałymi przedstawionymi na wykładzie.

Przekazanie przez wartość

Przykład

```c
#include <stdio.h>

void f5(int x, int y)
{
    puts("W funkcji:");
    printf("Wartość parametru \"x\" przed zmianą: %d\n", x);
    x+=1;
    printf("Wartość parametru \"x\" po zmianie: %d\n", x);
    printf("Wartość parametru \"y\": %d\n",y);
}

int main(void)
{
    int a=3;
    printf("Wartość zmiennej \"a\" przed przekazaniem do funkcji f5(): %d\n",a);
    f5(a,2*a);
    printf("Wartość zmiennej \"a\" po zakończeniu funkcji f5(): %d\n",a);
    return 0;
}
```

Przekazanie przez wartość

Komentarz do przykładu

Po uruchomieniu przykładu można przekonać się, czytając komunikaty programu, że zmiana wartości parametru „x" nie wpływa na wartość zmiennej „a", która została pod niego podstawiona. Parametry i argumenty, które są pod nie podstawiane mogą mieć takie same nazwy. Przekazanie, które jest zrealizowane w przykładzie jest równoważne następującym instrukcjom przypisania:

```
int x = a;
int y = 2*a;
```

Przekazanie przez stałą

Jeśli z jakiś powodów chcemy uniemożliwić zmianę wartości parametru wewnątrz funkcji, to możemy zastosować przekazanie przez stałą. Deklaracja takiego parametru jest poprzedzona słowem kluczowym `const`. Podobnie jak w przypadku przekazania przez wartość pod takie parametry można podstawiać wartości, stałe, zmienne i całe wyrażenia. Ten sposób przekazania również może być łączony z pozostałymi przedstawionymi na wykładzie.

Przekazanie przez stałą

Przykład

```c
#include <stdio.h>

void f6(const int x)
{
    printf("Wartość parametru \"x\": %d\n",x);
    printf("Wartość wyrażenia z parametrem \"x\": %d\n",x+1);
    /* x+=1; */ // Tak nie można, nie skompiluje się.
}

int a = 3;

int main(void)
{
    printf("Wartość zmiennej \"a\" przed wywołaniem funkcji: %d\n",a);
    f6(a);
    printf("Wartość zmiennej \"a\" po wywołaniu funkcji: %d\n",a);
    return 0;
}
```

Przekazanie przez stałą

Komentarz do przykładu

Jak można zauważyć analizując przykład, wartość parametru przy przekazaniu przez stałą nie ulega zmianie. Ten rodzaj przekazania odpowiada następującemu przypisaniu:

```
const int x = a;
```

Wprowadzenie do wskaźników

Zanim zostanie przedstawiony kolejny sposób przekazywania przez parametry musimy wprowadzić nowy typ zmiennej. Zmienna ta nazywa się *zmienną wskaźnikową* lub krótko: *wskaźnikiem* (ang. *pointer*). Wzorzec deklaracji takiej zmiennej jest następujący:

```
typ_zmiennej *nazwa_zmiennej;
```

Od deklaracji zwykłej zmiennej różni ją jedynie znak `*` stojący przed nazwą. Gdybyśmy próbowali ustalić rozmiar takiej zmiennej przy pomocy operatora `sizeof`, to okazałoby się, że zawsze jest on taki sam, niezależnie od typu danych, który użyty jest w jej deklaracji i że w przypadku komputerów 64-bitowych wynosi on 8 bajtów, a w przypadku komputerów 32-bitowych wynosi 4 bajty. Dzieje się, tak ponieważ wskaźnik nie przechowuje wartości bezpośrednio, ale przechowuje *adres* zmiennej, która tę wartość pamięta. Ta zmienna nazywana jest *zmienną wskazywaną*. To jakiego typu zmienne może wskazywać wskaźnik jest określone podanym w jego deklaracji typem.

Wprowadzenie do wskaźników

Wartość zerowa wskaźnika jest oznaczana stałą NULL, choć nowsze wersje standardu języka C pozwalają mu przypisać bezpośrednio `0`. Aby zapisać we wskaźniku adres innej zmiennej możemy posłużyć się jednoargumentowym operatorem wyłuskania adresu, który oznaczany jest (podobnie jak dwa inne operatory) znakiem `&` (czytaj: ampersand). Aby odczytać wartość zmiennej wskazywanej za pomocą wskaźnika należy użyć innego operatora, który nazywamy operatorem dereferencji i który jest oznaczany symbolem $*$. Aby wypisać na ekran adres przechowywany we wskaźniku należy użyć w funkcji `printf()` ciągu formatującego `"%p"`. Następny slajd prezentuje prosty program stosujący wskaźniki.

Wprowadzenie do wskaźników

Przykład

```c
#include <stdio.h>

int main(void)
{
    int *pointer = NULL;
    int variable = 3;
    pointer = &variable;
    printf("Wartość wskazywanej zmiennej: %d\n",*pointer);
    printf("Adres przechowywany we wskaźniku: %p\n",pointer);
    variable++;
    printf("Wartość wskazywanej zmiennej: %d\n",*pointer);
    *pointer+=1;
    printf("Wartość wskazywanej zmiennej: %d\n",variable);
    return 0;
}
```

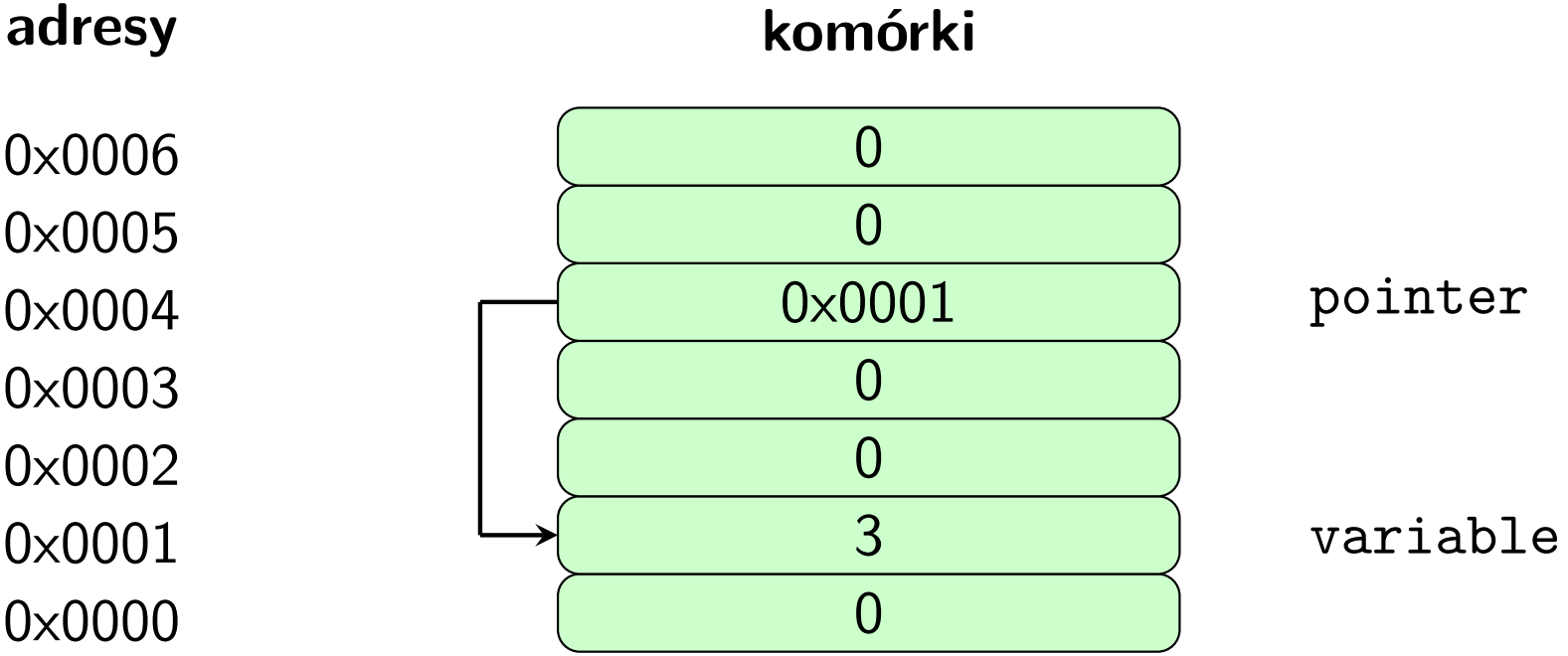
Wprowadzenie do wskaźników

Komentarz do przykładu

Śledząc wykonanie przykładowego programu możemy przekonać się, że wartość zmiennej wskazywanej można zmienić za pomocą wskaźnika, jak również zmianę wartości tej zmiennej można za jego pomocą odczytać. Proszę zwrócić uwagę na różnicę między odczytaniem wartości wskaźnika (adresu, który przechowuje), a odczytem wartości zmiennej przez niego wskazywanej.

Wprowadzenie do wskaźników

Aby lepiej zrozumieć jak działają wskaźniki, przedstawmy sobie bardzo uproszczony model pamięci, taki w którym każda zmienna ma wielkość jednej komórki. Tak mogłoby wyglądać rozmieszczenie zmiennych z przykładowego programu w tej pamięci:



Przekazanie przez wskaźnik

Wskaźniki mogą być użyte jako parametry dla funkcji. Argumentem wywołania podstawianym za taki parametr może jedynie być adres zmiennej uzyskany bezpośrednio z pomocą operatora wyłuskania lub ze wskaźnika tego samego typu co parametr. Parametr wskaźnikowy jest zarówno parametrem wejściowym, jak i *wyjściowym*. Za jego pośrednictwem można przekazać na zewnątrz funkcji wynik jej działania. Ma to zastosowanie wtedy, gdy funkcja musi zwrócić więcej niż jedną wartość.

Przekazanie przez wskaźnik

Przykład

```c
#include <stdio.h>

void f7(int *x)
{
    puts("W funkcji:");
    printf("Wartość parametru \"*x\" przed zmianą: %d\n",*x);
    *x+=1;
    printf("Wartość parametru \"*x\" po zmianie: %d\n",*x);
}

int main(void)
{
    int a = 3;
    printf("Wartość zmiennej \"a\" przed wywołaniem funkcji f7(): %d\n",a);
    f7(&a);
    printf("Wartość zmiennej \"a\" po zakończeniu funkcji f7(): %d\n",a);
    return 0;
}
```

Przekazanie przez wskaźnik

Komentarz do przykładu

Proszę zwrócić uwagę na sposób wywołania funkcji `f7()`, jak również na sposób użycia operatora dereferencji `*`. Ponieważ przykład jest prosty i ma tylko zilustrować przekazywanie przez wskaźnik, to funkcja `f7()` ma tylko jeden parametr, który z powodzeniem można by zastąpić parametrem przekazującym przez wartość, a obliczony wynik zwrócić jako wartość funkcji. Nie zawsze takie postępowanie jest możliwe. Czasem funkcja będzie zwracała więcej niż jeden wynik działania. Przekazanie przez wskaźnik można stosować wraz z innymi rodzajami przekazywania.

Czyste funkcje

W paradygmacie programowania funkcyjnego, który szerzej nie będzie omawiany w ramach tych wykładów, istnieje pojęcie *czystej* funkcji (ang. *pure function*). Jest to funkcja, która nie ma efektów ubocznych swojego działania, a jedynie zwraca wartość. Oznacza to, że taka funkcja nie zmienia wartości zmiennych globalny programu bezpośrednio, ani nawet za pomocą przekazania przez wskaźnik. Tego typu funkcje są bardzo przydatne w programowaniu współbieżnym wykorzystującym wątki, bo nie wpływają one na stan innych wątków, niż te, które je wywołały i nie wymagają dodatkowych zabiegów, aby ich działanie w programie wielowątkowym było bezpieczne.

Punkt wyjścia z funkcji

Miejsce w kodzie funkcji, gdzie kończy się jej wykonanie nazywane jest punktem wyjścia. Paradygmat programowania strukturalnego wymaga, aby funkcja (podprogram) miała tylko jeden punkt wyjścia. Oznacza, to że w jej kodzie powinniśmy tylko raz użyć słowa kluczowego `return` lub w przypadku funkcji zwracających `void` w ogóle go nie używać. Wielu programistów nie stosuje się do tej reguły, gdyż wielokrotne użycie `return` skraca zapis funkcji i sprawia, że jej kod staje się bardziej czytelny, a w niektórych przypadkach prostszy w działaniu.

Wskazówki dla tworzenia funkcji

1. Kod funkcji powinien być możliwie krótki i czytelny.
2. Funkcja powinna mieć opisową nazwę, najlepiej zawierającą czasownik.
3. Funkcja powinna mieć przynajmniej jeden parametr, z drugiej strony nie powinna mieć zbyt dużo parametrów.
4. Funkcja powinna realizować tylko jedno zadanie, opisane jej nazwą.
5. Funkcje bezparametrowe należy stosować bardzo oszczędnie.
6. Funkcja **nigdy** nie powinna używać bezpośrednio zmienny globalnych.

Programiści tworzący programy dla systemu Unix wymyślili konwencję pisania funkcji, która jest często stosowana również przez innych programistów. Polega ona na tym, że funkcja zwraca jako wartość kod poprawności swego wykonania. Jest to zwykle liczba całkowita. Zazwyczaj, jeśli jest ona równa zero, to znaczy, że funkcja się wykonała poprawnie, a jeśli jest ujemna, to znaczy, że wystąpił wyjątek, który jest sygnalizowany wartością bezwzględną tej liczby. Wynik działania funkcji może być przy stosowaniu takiej konwencji przekazywany przez wskaźnik.

Równanie kwadratowe - wersja z funkcjami

Funkcja pobierająca od użytkownika współczynniki równania

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

void get_abc_parameters(float *a, float *b, float *c)
{
    puts("Podaj współczynniki równania kwadratowego:");
    do {
        printf("a= ");
        scanf("%f",a);
        if(*a==0.0)
            puts("Wartość współczynnika 'a' nie może wynosić zero!");
    } while(*a==0.0);
    printf("b= ");
    scanf("%f",b);
    printf("c= ");
    scanf("%f",c);
}
```

Równanie kwadratowe - wersja z funkcjami

Komentarz do funkcji

Funkcja pobiera wprowadzone przez użytkownika przy pomocy klawiatury wartości współczynników równania kwadratowego, a więc realizuje tylko jedno zadanie. Proszę zwrócić uwagę, że przed drugimi argumentami wywołań funkcji `scanf()` nie użyto operatora wyłuskania `&`. Dzieje się tak dlatego, że jako drugi argument ta funkcja przyjmuje adres zmiennej, a ponieważ parametry `a`, `b` i `c` są wskaźnikami, to znaczy, że ten adres jest w nich zawarty. Zastosowanie przed nimi operatora wyłuskania byłoby błędem, bo w ten sposób przekazalibyśmy funkcji `scanf()` nie adresy zmiennych wskazywanych, do których powinna zapisać przekazane przez użytkownika liczby, a adresy samych wskaźników.
Instrukcje włączające pliki nagłówkowe nie są częścią funkcji, ale są niezbędne do prawidłowej kompilacji i działania programu.

Równanie kwadratowe - wersja z funkcjami

Funkcja wyliczająca deltę

```
float calculate_delta(float a, float b, float c)
{
    return b*b-4*a*c;
}
```

Równanie kwadratowe - wersja z funkcjami

Funkcja implementująca funkcję matematyczną *signum*

```
int signum(float number)
{
    return (number<0.0) ? -1 : (number>0.0) ? 1 : 0;
}
```

Równanie kwadratowe - wersja z funkcjami

Funkcja licząca współczynnik q

```
float calculate_q(float b, float delta)
{
    return -0.5*(b+signum(b)*sqrt(delta));
}
```

Równanie kwadratowe - wersja z funkcjami

Funkcja licząca pojedynczy pierwiastek równania

```
float calculate_root(float a, float q)
{
    return q/a;
}
```

Równanie kwadratowe - wersja z funkcjami

Funkcja licząca dwa pierwiastki równania

```
void calculate_roots(float a, float c, float q, float *x1, float *x2)
{
    *x1=q/a;
    *x2=c/q;
}
```

Równanie kwadratowe - wersja z funkcjami

Funkcja main()

```
int main(void)
{
    float a=0.0,b=0.0,c=0.0;
    get_abc_parameters(&a,&b,&c);
    float delta = calculate_delta(a,b,c);
    if(delta==0.0) {
        float q = calculate_q(b,delta);
        printf("Równianie ma jeden pierwiastek o wartości %.10f\n",
                calculate_root(a,q));
     }
     if(delta>0.0) {
           float q = calculate_q(b,delta);
           float x1=0.0, x2=0.0;
           calculate_roots(a,c,q,&x1,&x2);
           printf("Pierwiastki równania mają następujące wartości x1:\
           %.10f, x2: %.10f\n",x1,x2);
     }
     if(delta<0.0)
        puts("Równianie nie ma rozwiązań w dziedzinie liczby rzeczywistych");

    return 0;
}
```

Równanie kwadratowe - wersja z funkcjami

Komentarz do przykładu

Proszę zwrócić uwagę, w jaki sposób w funkcji `main()` został podzielony na dwa wiersze łańcuch znaków. Posłużył do tego znak backslash (\), który nakazuje traktowanie kompilatorowi dwóch następujących po sobie wierszy jako całości. Program napisany z użyciem funkcji jest dłuższy niż jego oryginalny odpowiednik, mimo to jest bardziej czytelny. Dodatkowo funkcja `signum()` może bez zmian zostać wykorzystana w innym programie, pozostałe są specyficzne dla rozwiązywanego problemu. Uwzględnienie rozróżnienia między przypadkiem kiedy równanie ma jeden i dwa pierwiastki również jest prostsze, kiedy stosowane są funkcje.

Podziękowania

Składam podziękowania dla dra inż. Grzegorza Łukawskiego i mgra inż. Leszka Ciopińskiego za udostępnienie materiałów, których fragmenty zostały wykorzystane w tym wykładzie.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!